

OBJECT ORIENTED PROGRAMMING

Using OOSimL

An Overview - Part 3

José M. Garrido
Department of Computer Science

Updated November 2016

College of Computing and Software Engineering
Kennesaw State University

© 2015, J. M. Garrido

1 Introduction to Arrays

An array is a static data structure that can store multiple values of the same type. These values are stored using contiguous memory locations with the same name. The values in the array are known as *elements*. Arrays can be used to store and manipulate a large number of values of the same type. This mechanism provides the ability to handle large number of values in a single collection and to refer to each value with an index.

An integer value or variable known as *index* is used to access a particular element of the array. The values of the index start from zero and it represents the relative position of the element in the array. Figure 1 shows an array with 10 elements.

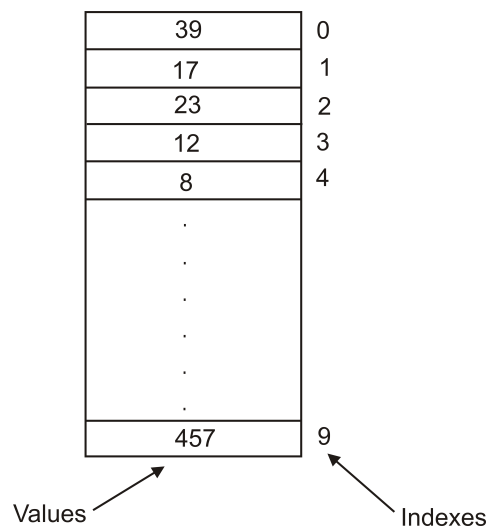


Figure 1: An array with 10 elements

An array is a static data structure because once the array is declared (and created), its size or capacity cannot change. An array is declared to hold 50 elements cannot be changed to hold a larger number of elements. The programmer will normally carry out the following sequence of steps to use an array:

1. Declaring the array with appropriate name and type

2. Optionally assigning initial values to the array elements
3. Manipulating the individual elements of the array

2 Array Declaration

To declare an array, an identifier is used for the name of the array. The type of the array is a simple (primitive) type or a class. The general statement for declaring an array of a simple type is:

```
define < array_name > array [ ] of type < array_type >
```

Arrays of simple types must be declared in the *variables* section of data definitions. In the following example an array *width* of type *float* is declared.

```
variables
  define width array [] of type float
  . . .
```

The statement for declaring an array of object references has the general form:

```
define < array_name > array [ ] of class < class_name >
```

Arrays of these types are declared in the *object references* section of data declarations. The following example declares an array *points* of class *Point*.

```
object references
  define points array [] of class Point
  . . .
```

3 Creating Arrays

After an array has been declared, it must be created with an appropriate capacity or size, and then values can be assigned to specific elements. The capacity of the array is the number of elements it can hold. The general form of the statement for creating an array is:

```
create < array_name > array [< capacity >]
```

of type $\langle \text{array_type} \rangle$

The following statement creates array, *width*, with capacity of 50 elements. This array was already declared.

```
create temp array [50] of type float
. . .
```

The general form of the statement to create an array of object references is:

```
create  $\langle \text{array\_name} \rangle$  array [ $\langle \text{capacity} \rangle$ ]
of class  $\langle \text{class\_name} \rangle$ 
```

An identifier constant is normally used to specify the capacity of the array. For example, given the constant *MAX_NUM* has a value 50 and *NUM_OBJECTS* a value of 25, the statements for declaring and creating the array *width* and array *points* are:

```
constants
  define MAX_NUM = 50 of type integer
  define NUM_OBJECTS = 25 of type integer
variables
  define width array [] of type float
object references
  define points array [] of class Point
. . .
begin
  . . .
  create width array [MAX_NUM] of type float
  create points array [NUM_OBJECTS] of class Point
```

The actual number of elements in the array is usually less than the total number or capacity of the array. For example, an array is declared with a capacity of 50 elements but only the first 20 elements of the array are used. An array is a static data structure, therefore elements cannot be inserted or deleted from the array.

4 Manipulating an Array

Manipulating an array involve accessing the individual elements of the array and performing some computing. To access an individual element of an

array, an integer value, known as the *index*, is used with the name of the array. The range of index values is 0 to the capacity of the array minus 1.

To access a particular element of an array a statement uses the name of the array followed by the index value in rectangular brackets. The following **set** statement assigns a value of 33.55 to element 3 of array *width*.

```
set width[2] = 33.55
```

The index value used is an integer constant, a constant identifier, or an integer variable. The following statement uses a reference to element 3 of array *width* with variable *j* as the index.

```
constants
  define MAX_NUM = 50 of type integer
variables
  define width array [] of type float
  define j of type integer
  . . .
  create width array [MAX_NUM] of type float
  set j = 2
  set temp[j] = 33.55
```

An array of object references is not an array of objects since the objects are stored elsewhere in memory. The following statement creates an object of class *Point* and assigns this to element with index *j* of array *points* (of class *Point*):

```
create points[j] of class Point
```

Element *j* of array *points* now refers to the newly created object of class *Point*. To invoke a public function of an object referenced in an element of an array, the call statement is used followed by the name of the array and the index value enclosed in brackets.

The following statement is used to invoke function *get_xcoord* of the object referenced by element with index *j* of array *points*.

```
variables
  define x of type float
  define j of type integer
```

```

    . . .
begin
    . . .
    set x = call get_xcoord of points[j]

```

5 Array Parameters

An array may be defined as a parameter in the header of a function definition. When invoking this function, an array of the same or similar type can then be passed as argument. A reference to the array is passed in the function call.

A function with an array parameter declares the array in the header of the function definition after the **parameters** keyword. The general form of a function header with a parameter definition is:

```

description
    . . .
    */
function < function_name >
parameters < parameter_list > is
    . . .
endfun < function_name >

```

The data declarations in *parameter_list* section of the function header may have zero or more array declarations. The general form of these parameter declarations is:

```

< array_name > array []
                    of type < simple type or class >

```

The following example defines a function that finds the maximum value in an array of type *float*. The function defines the array parameter and an integer parameter. This second parameter is the number of elements in the array. The function returns the maximum value found.

```

description
    This function calculates the maximum value of
    an array parameter myarray, it then returns the
    result.

```

```

*/
function maximum return type float
parameters myarray array [] of type float,
           num of type integer is
variables
  define max of type float      // local variable
  define j of type integer
begin
  set max = myarray[0]
  for j = 1 to num - 1 do
    if max < myarray[j] then

      set max = myarray[j]
    endif
  endfor
  return max
endfun maximum

```

In the call to a function and passing an array as an argument, only the name of the array is used. The following code calls function *maximum* and pass array *marr* and constant *NUM* as arguments.

```

constants
  define NUM = 100 of type integer
variables
  define marr array [] of type float
  define maxval of type float
  . . .
  create marr array [NUM] of type float
  . . .
  set maxval = call maximum using marr, NUM
  . . .

```

A function may also return the reference to an array. In the following example, a function creates an array *h* then returns a reference to this array.

```

create h array [NUM] of type float
. . .
return h

```

6 Array Return Types

The return type of a function can be specified to be of an array of a simple type or an array of a class. The following statement defines the general form of the return type in a class header.

```
function  $\langle$  function_name  $\rangle$ 
    return  $\langle$  type_class  $\rangle$  array []  $\langle$  type_class name  $\rangle$ 
```

In the following examples, two function definitions appear. The first example defines the return type of a function as an array of integer and the second example defines the return type of a function as an array of class Person.

```
function arrayfun1 return type array [] integer
parameters myarray array [] of class Person,
    num of type integer
is
. . .
endfun arrayfun1
function arrayfun2 return class array [] Person
parameters myarray array [] of class Point
is
. . .
endfun arrayfun2
```

7 Arrays with Multiple Dimensions

More than one dimension can be defined with arrays. Two-dimension arrays, also known as matrices, are mathematical structures with values arranged in columns and rows. Two index values are required, one for the rows and one for the columns.

With two-dimensional arrays, two numbers are defined each in a pair of brackets. The first number defines the range of values for the first index (for rows) and the second number defines the range of values for the second index (for the columns).

The following statements declare and create a two-dimensional array named *tmatrix* with capacity 50 rows and 15 columns.


```

constants
    define ROWS = 50 of type integer
    define COLS = 15 of type integer
variables
    define tmatrix array [][] of type float
    . . .
    create tmatrix array [ROWS][COLS] of type float

```

Two indexes are required to reference the elements of a two-dimensional array. The following statements assign all the elements of array *tmatrix* to 0.0:

```

for j = 0 to COLS - 1 do
    for i = 0 to ROWS - 1 do
        set tmatrix [i][j] = 0.0
    endfor
endfor

```

Two loop definitions are needed, an outer loop and an inner loop (this is also known nested loops). The inner loop varies the row index and outer loop varies the row index. The assignment statement sets the value 0.0 to the element at row *i* and column *j*.

8 Examples with Arrays

In this section, several simple applications of arrays are discussed. These find the maximum, minimum, average values, others to carry out search and sorting in arrays.

8.1 Finding Maximum and Minimum Values in an Array

Finding the minimum and/or maximum values stored in an array, involve accessing all the elements of the array. The algorithm for finding the maximum value in pseudo-code is:

1. Assign the current largest value found to be the value of the first element of the array.
2. Assign value zero to the index value of the first element.
3. Perform the following steps for each of the other elements of the array

- (a) Access the next element in the array.
 - (b) Examine its value, if the value of the current element is greater than the largest value so far, update the value found so far with this element value and save the index of the element.
4. The index value of the element value found to be the largest in the array is the result.

The algorithm described is implemented in function *maxf*. The function uses array *myarr* of type *float* that has *num* elements. The function returns the index value with the largest value found.

```

description
  This function of the element with the maximum
  value in the array and returns its index.
  */
function maxf return type integer
parameters num of type integer,
           myarr array [] of type float
is
  variables
    define j of type integer          // index variable
    // index of element with largest value
    define k of type integer
    define max_val of type float      // largest value
  begin
    set k = 0                        // index first element
    set max_val = myarr[0] // max value so far
    for j = 1 to num - 1 do
      if myarr[j] > max_val
      then
        set k = j

        set max_val = myarr[j]
      endif
    endfor
    return k                        // result
endfun maxf

```

8.2 The Average Value in an Array

To compute the average value in an array, all the elements have to be added to an accumulator variable, *sum*. The algorithm for computing the average

value in pseudo-code is:

1. Assign the accumulator variable, *sum*, to the value of the first element of the array.
2. For each of the other elements of the array, add the value of the next element in the array to the accumulator variable.
3. Divide the value of the accumulator variable by the number of elements in the array. This is the result value calculated.

An accumulator variable is used to store the summation of the element values in the array. In an array *x* with *n* elements. The summation of *x* with index *j* starting with *j* = 1 to *j* = *n* is expressed mathematically as:

$$sum = \sum_{j=1}^n x_j.$$

Function *averagef* implements the algorithm. The function uses array *myarr*, which is declared as an array of type *float* and has *num* elements. The function returns the average value calculated.

```
description
  This function calculates the average of the
  elements in array myarr.
*/
function averagef return type float
parameters myarr array of type float,
           num of type integer
is
  variables
    define sum of type float    // variable for summation
    define avg of type float    // average value
    define j of type integer
  begin
    set sum = 0
    for j = 0 to num - 1 do
      add myarr[j] to sum
    endfor
    set avg = sum / num
    return avg
endfun averagef
```

8.3 Searching

Searching involves examining the elements of an array for an element with particular value. Not all the elements of the array need to be examined, the search ends when and if an element of the array has a value equal to the requested value. There are two general techniques for searching: linear search and binary search.

8.3.1 Linear Search

Linear search finds the element with a specified value. The algorithm starts to compare the requested value with the value in the first element of the array and if not found, compare with the next element and so on until the last element of the array is compared with the requested value.

The result of this search is the index of the element in the array that is equal to the requested value. If the requested value is not found, the result is a negative value. The algorithm description in informal pseudo-code is:

1. Repeat for every element of the array:
 - (a) Compare the current element with the requested value. If the values are equal, store the value of the index as the result and search no more.
 - (b) If values are not equal, continue search.
2. If the value requested is not found, set the result to value -1.

Function *searcht* searches the array for an element with the requested temperature value, *t_val*. For the result, the function assigns to *t* the index value of the element with the value requested. If the requested value is not found, the function assigns a negative integer value to *t*.

```
description
  This function carries out a linear search of
  the array of temperature for the temperature
  value in parameter t_val. It sets the index
  value of the element found, or -1 if not found.
*/
function searcht return type integer
parameters t_val of type float,
           myarr array [] of type float,
           num of type integer
```

```

is
  variables
    define j of type integer
    define found = false of type boolean
  begin
    set j = 0
    while j < num and found not equal true do
      if myarr [j] == t_val
        then
          set t = j
          set found = true
        else
          increment j
        endif
      endwhile
    if found not equal true
      then
        set t = -1
      endif
    return t
  endfun searcht

```

8.3.2 Binary Search

In this type of search technique, the values to be searched have to be sorted in some given ascending order. The part of the array elements to include in the search is split into two partitions of about the same size. The middle element is compared with the requested value. If the value is not found, the search continues on only one partition. This partition is again split into two smaller partitions until the element is found or until no more splits are possible (not found). The binary search technique applied to a sorted array.

Binary search is a very efficient search technique compared to linear search because the number of comparisons is smaller. The efficiency of a search algorithm is determined by the number of relevant operations in proportion to the size of the array to search. The significant operations in this case are the comparisons of the element values with the requested value.

For an array with N elements, the average number of comparisons with linear search is $N/2$, and if the requested value is not found, the number of comparisons is N . With binary search, the number of comparisons is $\log_2 N$. The description of the algorithm in pseudo-code is:

1. Set up the lower and upper bounds of the array.
2. Repeat the search while the lower index value is less than the upper index value.
 - (a) Split the array into two partitions. Compare the middle element with the requested value.
 - (b) If the value of the middle element is the search value, the result is the index of this element, search no more.
 - (c) If the search value is less than the middle element, change the upper bound to the index of the middle element minus 1. The search will continue on the lower partition.
 - (d) If the search value is greater or equal to the middle element, change the lower bound to the index of the middle element plus 1. The search will continue on the upper partition.
3. If the search value is not found, the result is -1.

Function *bsearcht* implements the binary search algorithm using array parameter *myarr*.

description

```
This function implements a binary search of
the myarr array using search value in
parameter t_val. It sets the index
value of the element found, or -1 if not found.
*/
```

```
function bsearcht return type integer
```

```
parameters t_val of type float
```

```
myarr array [] of type float
```

```
num of type integer
```

```
is
```

```
variables
```

```
define t of type integer
```

```
define found = false of type boolean
```

```
define lower of type integer // index lower bound
```

```
define upper of type integer // index upper bound
```

```
define middle of type integer // index of middle
```

```
begin
```

```
set lower = 0
```

```
set upper = num
```

```
while lower < upper and found not equal true
```

```

do
  set middle = (lower + upper) / 2
  if t_val == myarr[middle]
  then
    set found = true
    set t = middle
  else
    if t_val < myarr[middle]
    then
      set upper = middle - 1
    else
      set lower = middle + 1
    endif
  endif
endwhile
if found not equal true
then
  set t = -1
endif
return t
endfun searcht

```

8.4 Sorting

Sorting is a technique that consists of rearranging the elements of the array in specified some order. If the values of the array to be sorted are numerical, then the two possible orders are ascending and descending. If the values of array are strings, then alphabetical order is the most practical. Some of the most widely known sorting algorithms are:

- Selection sort
- Insertion sort
- Bubble sort
- Shell sort
- Merge sort
- Quick sort

Selection sort is a very simple sorting algorithm. Given an array of a numerical type of size N , the algorithm performs several steps. First, it finds the index value of the smallest element value in the array. Second, it swaps this element with the element with index 0 (the first element. This step actually places the smallest element to the first position. Third, the first step is repeated for the part of the array with index 1 to $N - 1$, this excludes the element with index 0, which is at the proper position. The smallest element found is swapped with the element at position with index 1. This is repeated until all the elements are located in ascending order. The algorithm description in pseudo-code is:

1. Repeat the following steps for all elements with index $J = 0$ to $N-2$
2. Search for the smallest element from index J to $N-1$.
3. Swap the smallest element found with element with index J , if the smallest element is not the one with index J .

The following function *selectionsort* implements the algorithm for the selection sort.

```
description
  This function implements a selection sort of
  the array.
  */
function selectionsort
parameters myarr array [] of type float,
          num of type integer
is
  variables
    define N of type integer      // elements in array
    define Jmin of type integer  // smallest element
    define j of type integer
    define k of type integer
    define t_val of type float   // intermediate value
```



```

begin
  set N = num
  for j = 0 to N - 2 do
    // search for the smallest element
    // in the index range from j to N-1
    set Jmin = j
    for k = j+1 to N - 1 do
      if myarr[k] < myarr[Jmin]
      then
        set Jmin = k
      endif
    endfor
    if Jmin != j
    then
      // swap elements with index J and Jmin
      set t_val = myarr[j]
      set myarr[j] = myarr[Jmin]
      set myarr[Jmin] = t_val
    endif
  endfor
endfun selectionsort

```

The efficiency of the algorithm is formally expressed as $O(N^2)$. The number of element comparisons with an array size of N is $N^2/2 - N/2$. The first term ($N^2/2$) in this expression is the dominant one; the order of growth of this algorithm is N^2 .

9 Introduction to Inheritance

Inheritance is a class relationship among classes. The other basic class relationship is composition, which is a stronger form of association. These relationships are easily modeled in UML diagrams. Composition is a horizontal relationship and inheritance is a vertical relationship.

Inheritance is a facility provided by an object-oriented language for defining new classes from existing classes. The basic inheritance relationships and their applications are explained in some detail. Inheritance enhances class reuse, that is, the use of a class in more than one application.

10 Modeling Classes

A class is defined as a group of objects with common characteristics. Some classes of an application are completely independent because they do not have any relationship with other classes. The other classes in an application are related in some manner and they form a hierarchy of classes, therefore, class relationships must also be identified.

To help describe a class hierarchy, the most general class is placed at the top. This is known as the *parent* class and is also known as the *super* class (or the *base* class). A *subclass* inherits the (all attributes and operations) of its parent class. These characteristics of a class are also known as features.

A subclass can be further inherited to lower-level classes. In the UML class diagram, an arrow with an empty head points from a subclass (the derived class) to its base class to show that it is inheriting the features of its base class.

From UML diagram description of this type of class relationship, inheritance is seen as a vertical relationship between two classes. Figure 2 illustrates this class relationship. Class *Motor Vehicle* is the base class, the other three classes inherit the features of this base class.

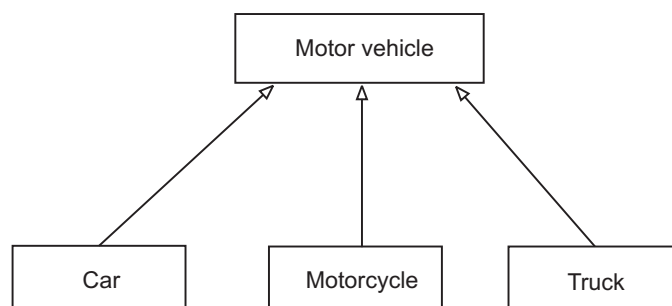


Figure 2: Class inheritance

The terminology varies somewhat in different software environments. In UML, the term *generalization* is used instead of inheritance. In the Java programming language, the term *extends* is used for a class that inherits features from a base class.

11 Inheritance

In defining a new class using inheritance, the newly defined class acquires all the non-private features of an existing parent class. This mechanism is provided by most object-oriented programming languages. The parent class is also known as the *base* class, or the *super* class. The new class being defined is known as a *derived* class or *subclass*.

The base class is a more general class than its subclasses. A derived class can be defined by adding more features or modifying some of the inherited features. Therefore, a subclass can be defined as:

- An extension of the base class, if in addition to the inherited features, it includes its own attributes and operations.
- A specialized version of the base class, if it overrides (redefines) one or more of the features inherited from its parent class
- A combination of an extension and a specialization of the base class

Using UML, the term *generalization* is defined as the association between a general class and a more specialized class or extended class. This association is also known as *inheritance*, and it is an important relationship between two classes. In the UML class diagram, the arrow that represents this relationship points from a class (the derived class) to its parent class.

The purpose of inheritance is to define a new class from an existing class and to shorten the time compared to the development of a class from scratch. Inheritance also enhances class reuse.

The ability of a class to inherit the characteristics from more than one parent class is known as *multiple inheritance*. Most object-oriented programming languages support multiple inheritance. The OOSimL and Java languages only support single inheritance.

A simple class hierarchy with inheritance is shown in Figure 2. The base class is *Motorvehicle* and the subclasses are: *Car*, *Motorcycle*, and *Truck*.

All objects of class *Truck* in Figure 2 are also objects of class *Motorvehicle*, because this is the base class for the other classes. On the contrary, not all objects of class *Motorvehicle* are objects of class *Truck*.

11.1 Defining Subclasses with OOSimL

As mentioned previously, the public features of the base class are inherited by the subclass. Protected features are inherited by the subclass, but the access is limited to classes in the same package.

The definition of a subclass in OOSimL must include the keyword **inherits** followed by the name of the base class. The general form of the OOSimL statement for the header in the definition of a subclass is:

```

description
...
class < class_name > inherits < base_class_name > is
  private
  ...
  protected
  ...
  public
  ...
endclass < class_name >

```

In UML class diagrams, a feature of the class is indicated with a plus (+) sign if it is public, with a minus (-) sign if it is private, and with a pound (#) sign if it is protected.

11.2 Inheritance and Initializer Functions

The initializer functions of a base class are the only public features that are not inherited by the subclasses. An initializer function of a subclass should initialize its own attributes (variables and object references) as well as the attributes defined in the base class. Therefore, when defining an initializer function of a subclass, it will normally invoke the initializer function of the base class.

The special name given to the initializer function of the base class is *super*. This call must be the first statement in the initializer function of the subclass. Calling function *super* may require arguments, and these must correspond to the parameters defined in the initializer function of the base class.

The statement to call or invoke an initializer function of the base class from the subclass is:

```

call super < using argument_list >

```

If the argument list is absent in the call, the initializer function invoked is the default initializer of the base class.

In the following example, a new class, *Student*, is defined that inherits the features of an existing (base) class *Person*. The attributes of class *Person*

are *pnumber*, *age*, *name*, and *address*. The initializer function of class *Person* sets initial values to these attributes.

The subclass *Student* has two other attribute, *major* and *gpa*. The initializer function of this class invokes the initializer function of the base class and sets initial values to its two attributes, *major* and *gpa*. The following code shows this part of class *Student*.

```
class Student inherits Person is
  private
  variables
    define major of type integer
    define gpa of type float
  public
  description
    This is the constructor, it initializes a Student
    Object on creation.
  */
  function initializer
  parameters
    imajor of type integer,
    igpa of type float,
    iname of type string,
    ipnum of type long,
    iage of type integer,
    iaddress of type string
  is
  begin
    // invoke the initializer of the base class
    call super using iname, ipnum, iage, iaddress
    set major = imajor
    set gpa = igpa
  endfun initializer
  . . .
endclass Student
```

The attributes of a class are usually private, so the only way to set the initial values for the attributes of the base class is to invoke its initializer function of the base class.

In the previous code, this is accomplished with the statement:

```
call super using iname, ipnum, iage, iaddress
```

11.3 Complete Example with Inheritance

The following source code shows the implementation of the base class, *Person*. The attributes of class *Person* are *pnumber*, *age*, *name*, and *address*. The initializer function of class *Person* sets initial values to these attributes.

```

description
  This class represents person data. The attributes are
    ID number, age, address, and name.
  */
class Person is
  private
  variables
    define pnumber of type long
    define age of type integer
    define name of type string
    define address of type string
  public
  description
    This is the constructor, it initializes a Person object
    on creation.
  */
  function initializer
  parameters iname of type string,
             ipnum of type long,
             iage of type integer,
             iaddress of type string

  is
  begin
    //
    set pnumber = ipnum
    set age = iage
    set name = iname
    set address = iaddress
  endfun initializer
  //
  description
    This funtion gets the person number of the
    Person object.    */
  function get_pnum return type long is
  begin
    return pnumber
  endfun get_pnum
  //
  description

```

```

        This function returns the name of the employee object.
    */
function get_name return type string is
begin
    return name
endfun get_name
//
description
    This function returns the address increase for the object.
    */
function get_address return type string is
begin
    return address
endfun get_address
//
description
    This function displays the attributes of the Person object
    */
function display_data is
begin
    display name, " ", age, " ", address
endfun display_data
endclass Person

```

The following code shows the implementation of class *Student*, which is a subclass of class *Person*.

```

description
    This class represents student objects. The attributes are
    major and GPA. The major is coded to an integer number.
    This class inherits class Person.
    */
class Student inherits Person is
private
variables
    define major of type integer
    define gpa of type float
public
description
    This is the constructor, it initializes a Student
    on creation.
    */
function initializer
parameters

```

```

        imajor of type integer,
        igpa of type float,
        iname of type string,
        ipnum of type long,
        iage of type integer,
        iaddress of type string
is
begin
    // invoke the initializer of the base class
    call super using iname, ipnum, iage, iaddress
    set major = imajor
    set gpa = igpa
endfun initializer
//
description
    This funtion gets the student major of the
    object.
    */
function get_major return type integer is
begin
    return major
endfun get_major
//
description
    This funtion sets the student major of the
    object.    */
function set_major parameters pmajor of type integer is
begin
    set major = pmajor
endfun set_major
//
description
    This function returns the GPA of the student object.
    */
function get_gpa return type float is
begin
    return gpa
endfun get_gpa
//
description
    This function sets the GPA of the student object.
    */
function set_gpa parameters pgpa of type float is
begin
    set gpa = pgpa

```



```

endfun set_gpa
//
description
    This function displays the attributes of the
    student object. it overrides the function of
    the base class
*/
function display_data is
begin
    call super.display_data
    display major, " ", gpa
endfun display_data
endclass Student

```

11.4 Function Overriding

A subclass is a *specialization* of the base class if one or more functions of the base class are *redefined* (or overridden) in the subclass. The subclass is said to reimplement one or more functions of the base class.

In class *Student*, which was shown previously, function *display_data* is a redefinition of the function with the same name in the base class. Class *Student* inherits class *Person* and overrides function *display_data*. Therefore, class *Student* is an extension and also a specialized version of class *Person*.

12 Abstract Classes

An abstract class has one or more *abstract functions*. An abstract function has only its declaration (also called its specification) and no implementation is included. A pure abstract class has all its functions declared as abstract.

Because an abstract class has one or more functions without their implementations, the class cannot be instantiated and it is generally used as a base class. The subclasses override the abstract functions inherited from the abstract base class and provide implementation for these functions.

12.1 Definition of Abstract Classes

In OOSimL, the keyword **abstract** is used before the keyword **class** when defining an abstract class. The definitions of abstract functions are also preceded by the keyword **abstract**. The OOSimL statements for abstract class definition has the following general form:

description

```

      . . .
abstract class < class_name > is
  private
      . . .
  protected
      . . .
  public
      . . .
endclass < class_name >

```

As mentioned previously, an abstract class is normally defined as a base class. Base classes provide single general descriptions for the common functionality and structure of its subclasses. Therefore, an abstract class is a foundation on which to define subclasses. Classes that are not abstract classes are known as *concrete classes*.

For example, a class that represents basic geometric figures is defined as an abstract class. Two functions *area* and *perimeter* are declared in the class. These functions compute the area and the perimeter of a figure. Class *Genfig* is defined in the following code:

```

description
  This is an abstract class that defines a generic
  geometric figure
  */
abstract class Genfig is
public
  // compute are of geometric figure
  abstract function area return type double
  //
  // compute perimeter of geometric figure
  abstract function perimeter return type double
endclass Genfig

```

The base class *Genfig* is defined as an abstract class because it does not provide the implementation of the functions *area* and *perimeter*.

Figure 3 shows the class hierarchy with base class *Genfig* and three subclasses: *Rectangle*, *Triangle*, and *Circle*. The calculations of area and perimeter are different in each of these subclasses and the base class *Genfig*

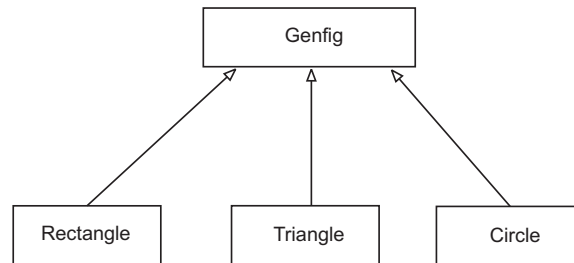


Figure 3: A class hierarchy of geometric figures

cannot include the implementations for these functions. The base class only provides the declaration (prototypes) for these functions.

Because the abstract class *Genfig* does not include the implementation body of the functions *area* and *perimeter*, objects of this class cannot be created. In other words, class *Genfig* cannot be instantiated.

12.2 Inheriting Abstract Classes

Because an abstract base class does not provide an implementation to one or more of its functions, the subclasses that inherit an abstract base class have to override (redefine) the functions that are defined as abstract functions in the abstract base class. The subclass, *Triangle*, for example, inherits class *Genfig* and includes the specific implementation of the functions *area*, *perimeter*, and the relevant attributes. The following code implements class *Triangle*.

```

description
    This class computes the area and perimeter
    of a triangle, given its three sides.
    */
class Triangle inherits Genfig is
    private
    variables
        define x of type double    // first side
        define y of type double    // second side
        define z of type double    // third side
    public
    description
        This function sets values for the three

```

```

        sides of the triangle.      */
function initializer parameters a of type double,
        b of type double, c of type double is
begin
    set x = a
    set y = b
    set z = c
endfun initializer
//
description
    This function computes the perimeter of a
    triangle. */
function perimeter return type double is
variables
    define lperim of type double
begin
    set lperim = x + y + z
    return lperim
endfun perimeter
//
description
    This function computes the area of a
    triangle.      */
function area return type double is
variables
    define s of type double      // intermediate result
    define r of type double
    define larea of type double
begin
    set s = 0.5 * (x + y + z)
    set r = s * (s - x)*(s - y)*(s - z)
    set larea = call Math.sqrt using r
    return larea
endfun area
endclass Triangle

```

13 Interfaces

An interface does not include constructors and cannot be instantiated. However, none of the functions of an interface include implementation, does not include attribute definitions although constant definitions are allowed. An interface is similar to a pure abstract class.

13.1 Interface Definition

The keyword **interface** is used followed by the name of the interface, in defining an interface. For the functions, the keyword **abstract** is not needed because all the functions are abstract functions. All features of an interface are implicitly public. The definition of an interface in OOSimL has the following general structure:

```
description
. . .
interface < interface_name > is
  public
  constants
. . .
  // public functions
. . .
endinterface < interface_name >
```

An interface, *IGenfig*, is shown in the following code. It defines the specification for the behavior of objects of any class that implements this interface. Note that the form of the interface is very similar to that of a pure abstract class.

```
description
  This is an interface that defines a generic
  geometric figure
  */
interface IGenfig is
  public
    // compute are of geometric figure
    function area return type double
  //
    // compute perimeter of geometric figure
    function perimeter return type double
endinterface IGenfig
```

13.2 Implementing an Interface

All methods declared in the interface must be implemented in the class that implements it. This class can define additional features. The header of the

class that implements an interface has the keyword **implements** followed by the name of the interface. The class header has the general form:

description

```

      . . .
class < cls_name > implements < interface_name > is
      . . .
endclass < cls_name >

```

Two additional and important differences between an interface and an abstract class is that multiple interfaces can be implemented by a class, whereas only one abstract base class can be inherited by a subclass, and it is mandatory for a class that implements an interface to implement all the functions defined in the interface.

14 Object Types

Abstract classes and interfaces can be used as *super types* for object references. Although they cannot be instantiated, interfaces and abstract classes are useful for declaring object references.

In the example of geometric figures, class *Genfig* is an abstract class and the other classes are subclasses. An object reference can be declared of type *Genfig*. Objects of the subclasses can be declared of each of their classes. The subclasses are considered subtypes of type *Genfig*. The following lines of code declare four object references, *gfigure*, *triangle_obj*, *circle_obj*, and *rect_obj*.

```

define gfigure of class Genfig
define triangle_obj of class Triangle
define circle_obj of class Circle
define rect_obj of class Rectangle

```

The types of these object references declared are considered subtypes in the problem domain because of the original class hierarchy represented in Figure 3.

The object reference *triangle_obj* is declared of type *Triangle*, but is also of type *Genfig* and any object reference of type *Triangle* is also of type *Genfig*.

The principle holds going upward in the class hierarchy, the opposite is not true; any object reference of type *Genfig* is not also of type *Rectangle*.

An interface can also be used as a super type, and all the classes that implement the interface are subtypes.

15 Polymorphism

Polymorphism is a runtime mechanism of an object-oriented language. This mechanism selects the appropriate version of a function to be executed depending on the actual type of the object. This function selection is based on *late binding* that occurs at execution time.

The typing principle explained previously allows an object reference of a super type to reference an object of a subtype. The following code creates objects for the object references *triangle_obj*, *circle_obj*, and *rect_obj*.

```
create triangle_obj of class Triangle using x, y, z
create circle_obj of class Circle using r
create rect_obj of class Rectangle using x, y
```

An object reference *gfigure* of class *Genfig* can be assigned to refer to any of the object of a subtype. The following code implements such an assignment.

```
set gfigure = triangle_obj
```

In this assignment, the type of object reference *triangle_obj* is a subtype of the type of object *gfigure*. It is legal to invoke a function of the abstract class *Genfig* that is implemented differently in the subclasses. The following code invokes function *perimeter*:

```
call perimeter of gfigure
```

The actual function invoked is the one implemented in class *Triangle*, because it is the actual type of the object reference *triangle_obj*. Another similar assignment can be included at some other point in the program.

The following code assigns the object reference *circle_obj* to the object reference *gfigure* and the call to function *perimeter* is the same as before, because the three subtypes represented by the three subclasses *Rectangle*, *Circle*, and *Triangle* each implement function *perimeter*.

```
set gfigure = circle_obj  
call perimeter of gfigure
```

Because the implementation for function *perimeter* is different in the three classes, the runtime language system selects the right version of the function to call.