

OBJECT ORIENTED PROGRAMMING

Using OOSimL

An Overview - Part 2

José M. Garrido
Department of Computer Science

Updated November 2016

College of Computing and Software Engineering
Kennesaw State University

© 2015, J. M. Garrido

1 Function Implementation

A program is normally decomposed into classes, and classes are divided into methods or functions.

As mentioned previously, data declared within a function is known only to that function—the scope of the data is *local* to the function. The data in a function has a limited lifetime, it only exists during execution of the function.

The name of the function is used when it is called or invoked by some other function. The relevant documentation of the function is described in the **description** paragraph, which ends with a star-slash (*). The OOSimL statements for defining a function are:

```

description
    . . .
    */
function < function_name > is
    constants
        . . .
    variables
        . . .
    object references
        . . .
    begin
        . . . [instructions]
    endfun < function_name >

```

The data declarations are divided into constant declarations, variable declarations, and object declarations. This is similar to the data declarations that appear in the class. These declarations define local data in the function and are optional. The instructions of the function appear between the keywords **begin** and **endfun**. The following OOSimL code shows a simple function for displaying a text message on the screen.

```

description
    This function displays a message
    on the screen. */
function show_message is

```

```
begin
  display "Computing data"
endfun show_message
```

2 Function Calls

A function starts executing when it is called by another function. A function *call* is also known as *method invocation*. The function that calls another function is known as the calling function; the second function is known as the called function. When a function calls or invokes another function, the flow of control is altered and the second function starts execution. When the called function completes execution, the flow of control is transferred back (returned) to the calling function. This function continues execution from the point after it called the second function.

After completion, the called function may or may not return a value to the calling function. From the data transfer point of view, there are three general categories of functions:

1. *Simple functions* are functions that do not return any value when they are invoked. The previous example, function *show_message*, is a simple (or void) function because it does not return any value to the function that invoked it.
2. *Value-returning* functions that return a single value after completion.
3. Functions with *parameters* are functions that require one or more data items as input values when invoked.

Functions that combine the last two categories —functions that return a value and that have parameters are also grouped in two categories, according to their purpose:

- Functions that return the value of an attribute of the object and do not change the value of any attribute(s) of the object. These are known as *accessor* functions.
- Functions that change the state of the object in some way by altering the value of one or more attributes in the object. Normally, these functions do not return any value. These are known as *mutator* functions.

Good programming practice recommends defining the functions in a class as being either accessor or mutator.

2.1 Calling Simple Functions

Simple functions do not return a value to the calling function. An example of this kind of function is *show_message*, discussed previously. There is no data transfer to or from the function. In OOSimL, the statement that calls or invokes a simple function that is referenced by an object reference variable is:

```
call < function_name > of < object_reference >
```

In the following example, function *show_message* that belongs to an object referenced by *carobj*, is invoked from function *main*, the call statement is:

```
call show_message of carobj
```

Figure 1 shows the calling function and the called function, *show_message*. After completing its execution, the called function returns the flow of control to the calling function.

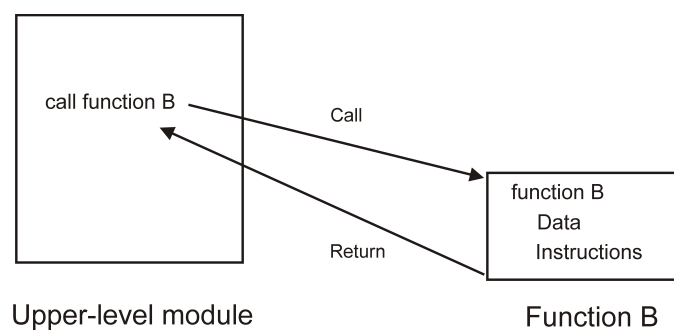


Figure 1: The calling and called functions

Data can be transferred between the calling and the called function. This data transfer may occur from the calling function to the called function, from the called function to the calling function, or in both directions.

2.2 Value-Returning Functions

Value-returning functions transfer data from the called function to the calling function. In these functions, a single value is calculated or assigned to a variable and is returned to the calling function.

The called function is defined with a type, which is the type of the return value. The function *return type* may be a simple type or a class. After the execution of the called function completes, control is returned to the calling function with a single value. The OOSimL statements that define the form of a value-returning function are:

```

description
    . . .
*/
function < function_name > of type < return_type > is
    . . .
    return < return_value >
endfun < function_name >

```

The value in the return statement can be any valid expression, following the **return** keyword. The expression can include constants, variables, object references, or a combination of these.

The following example defines a function, *get_name* that returns the value of the attribute *emp_name* of an object of class Employee. This value is returned to the calling function. In the header of the function, the type of the value returned is indicated as **string**. The code for this function definition is:

```

description
    This function returns the name of the employee object.
*/
function get_name return type string is
begin
    return name
endfun get_name

```

The value-returning function can be called and the value returned can be used in several ways:

- Call the function in a simple assignment statement
- Call the function in an assignment with an arithmetic expression
- Call the function in a more complete *call* statement

The value returned by the called function is used by the calling function by assigning this returned value to another variable. The following example

shows a function calling function *get_salary*. The calling function assigns the value returned to variable *sal*. There are several ways to write this code and the statements are:

```
set sal = call get_salary
call get_salary value to sal
set sal = get_salary()
```

The value returned can be used in an assignment with an arithmetic expression after calling a function. For example, after calling function *get_salary*, the value returned is assigned to variable *sal*. This variable is then used in an arithmetic expression that multiplies *sal* by variable *percent*. The value that results from evaluating this expression is assigned to variable *upd_sal*. This assignment statement is:

```
set upd_sal = sal * percent
```

2.3 Function Calls with Arguments

The called function can receive data values when called from another function. The functions definition for functions to be called should include data definitions known as *parameter declarations*. These parameter definitions are similar to local data declarations and have local scope and persistence. The called function may return a value, although it is not recommended to define the called function for this.

The data values sent to the called function by calling function are known as *arguments* and can be actual values (constants) or names of data items. When there are two or more argument values in the function call, the argument list consists of the data items separated by commas. The argument list should appear after the keyword **using** in the **call** statement. The OOSimL **call** statement for a function call with arguments is:

```
call < function_name > of < object_ref >
      using < argument_list >
```

The following example is a call to function *cal_area* that belongs to an object referenced by *myobj*. The function calculates and displays the area of a triangle given two arguments *x* and *y*. This **call** statement is:

```
call cal_area of myobj using x, y
```

The declaration of parameters in a function defines for every parameter a type and a parameter name. The general structure of a function with parameter declaration is:

```

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

```

In the following example, function *cal_area* computes the area of a triangle given the base and altitude. The code for the function in OOSimL is:

```

description
    This function calculates the area of
    A triangle given the base b and the height h
    it then prints the result on the screen.
*/
function cal_area
parameters b of type float,
           h of type float
is
variables
    define area of type float    // local variable
begin
    set area = b * h * 0.5
    display "Area of triangle is: ", area
    return                                     // empty return
endfun cal_area

```

In the example, function *cal_area* declares the parameters *b* and *h*. The parameters are used as placeholders for the corresponding argument values transferred from the calling function. Figure 2 illustrates the call of function *cal_area* with argument values from the calling function.

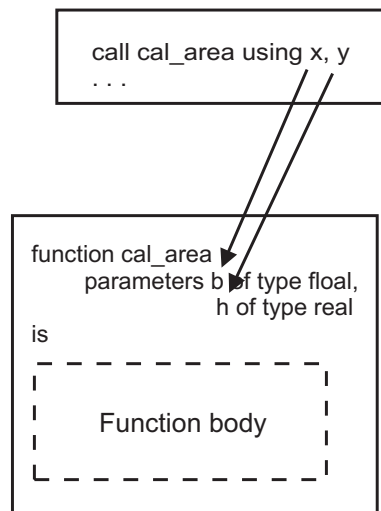


Figure 2: Using arguments in a function call

3 Constructors

One or more *constructors* functions, also known as *initializer* functions, will normally appear in a class definition. These are special functions that are called when creating objects of the enclosing class. The main purpose of an initializer function is to set the object created to an appropriate initial state. The attributes of the corresponding object are assigned to some initial values.

In the following example, class *Employee* include the attribute declaration and the definition of the initializer function.

```
class Employee is
  private
  variables
    define salary of type double
    define age of type integer
    define name of type string
    define sal_increase of type double
  object references
    define employ_date of class EmpDate
  public
  description
    This is the constructor, it initializes an Employee
    object on creation.    */
```



```

function initializer
parameters iname of type string,
           isalary of type double,
           iage of type integer
is
begin
  set salary = isalary
  set age = iage
  set name = iname
  set sal_increase = 0.0
  set employ_date = null
endfun initializer
. . . [other functions in the class]
endclass Employee

```

It is recommended to define at least a default initializer function. When no initializer function appears in the class definition, the default values are (zero and empty) assigned to the attributes.

In class *Employee* the default values for the *sal_increase* attribute is set to 0.0.

An initializer function can set the value of the attributes to the values given when called. These values are given as arguments in the statement that creates the object. The general statement to create an object with given values for one or more attributes is:

```

create < object_ref_name > of class < class_name >
      using < argument_list >

```

The following example uses an object reference *emp_obj* of class *Employee* to create an object with initial values of "John Doe" for *name*, 35876.00 for *salary*, and 45 for *age*.

```

create emp_obj of class Employee
      using "John Doe", 35876.00, 45

```

Overloading is the programming facility of defining two or more functions with the same name. This facility of the programming language allows any function in a class to be overloaded. These functions are defined with the same name, but with a different number of and types of parameters. In this way, there is more than one way to initialize an object of the class.

4 Complete Sample Program

The following example is a complete program that consists of two classes: class *Employeeem* and class *Comp_salary*. The definition of class *Employee* is as follows:

```
description
  This program computes the salary increase for an employee;
  if his/her salary is greater than $45,000 the salary
  increase is 4.5%, otherwise the salary increase is 5%.
  This is the class for employees. The main attributes are
  salary, age, and name.  */
class Employeeem is
  private
  variables                // variable data declarations
    define salary of type float
    define age of type integer
    define obj_name of type string
    define salincrease of type float  // salary increase
  public
  description
    This is the constructor, it initializes an object on
    creation.      */
  function initializer
  parameters iname of type string,
             isalary of type float,
             iage of type integer
  is
  begin
    set salary = isalary
    set age = iage
    set obj_name = iname
  endfun initializer
```

```

description
    This funtion gets the salary of the employee object. */
function get_salary return type float is
    begin
        return salary
    endfun get_salary
//
description
    This function returns the name of the employee object. */
function get_name return type string is
    begin
        return obj_name
    endfun get_name
//
description
    This function computes the salary increase and
    updates the salary of an employee.      */
function sal_increase is
    constants    // constant data declarations
        define percent1 = 0.045 of type float
        define percent2 = 0.050 of type float
    begin        // body of function starts here
        if salary > 45000.00 then
            set salincrease = salary * percent1
        else
            set salincrease = salary * percent2
        endif
        add salincrease to salary    // update salary
    endfun sal_increase
//
description
    This function returns the salary increase for
    the object. */
function get_increase return type float is
    begin
        return salincrease
    endfun get_increase
endclass Employeem

```

The other class, *Comp_salary*, includes function *main* that starts and controls the execution of the entire program. The following code is the implementation of class *Comp_salary*.

```

description
    This program computes the salary increase for an employee;
    if his/her salary is greater than $45,000 the salary
    increase is 4.5%, otherwise the salary increase is 5%.
    This class creates and manipulates the objects of
    class Employeeem. */
class Comp_salary is
    public
    description
        This is the main function of the application.
        */
    function main is
        variables
            define increase of type float
            define salary of type float
            define age of type integer
            define empname of type string
        object references
            define emp_obj of class Employeeem
        begin
            display "Enter salary: "
            read salary
            display "Enter age: "
            read age
            display "Enter employee name: "
            read empname
            create emp_obj of class Employeeem using
                empname, salary, age
            call sal_increase of emp_obj
            set increase = call get_increase of emp_obj
            set salary = get_salary() of emp_obj // updated salary
            display "Employee name: ", empname
            display "increase: ", increase, " new salary: ", salary
        endfun main
    endclass Comp_salary

```

5 Static Features

Functions or attributes in a class may not be associated with any object of the class. These features are known as *static* features and do not belong to an object, although they are defined in a class. To access these features, the class name followed by a dot and then the name of the feature is used. The

following example invokes method *cos* of class *Math*:

```
set angle = Math.cos(x) * delta
```

To define a static function, the keyword **static** is written before the name of the function. The following example is the header for the definition of method *compute_h*:

```
static function compute_h is
. . .
endfun compute_h
```

In a similar manner, the declaration of a static attribute is written with the keyword **static** before the variable declaration. These variables should normally be private and should be accessed by (public) accessor functions. When the program executes, for every declaration of a static variable, there is only one copy of its value shared by all objects of the class. Often, static variables are also known as class variables, and static methods as class methods.

The following example declares a static variable named *count_pins* and initializes it to value 20.

```
static count_pins = 20 of type integer
```

6 Design Notations

Designing a solution to a problem involves design of an algorithm, which will be as general as possible to solve a family or group of similar problems. To describe an algorithm, several notations are used, such as informal English, flowcharts, and pseudo-code.

An algorithm can be described at several levels of abstraction. Starting from a very high-level and general level of description of a preliminary design, to a much lower-level that has more detailed description of the design.

A notation is a more informal and higher level than a programming language. It is set of rules used for the informal description of an algorithm. Two widely-used design notations are:

- Flowcharts
- Pseudo-code

6.1 Flowcharts

Flowcharts consist of a set of symbol blocks connected by arrows. A flowchart is a visual representation for describing a sequence of design or action steps of the solution to a problem. The arrows that connect the blocks show the order in which the action steps are to be carried out. The arrows also show the flow of data.

Several simple flowchart blocks are shown in Figure 3. A flowchart always begins with a *start* symbol, which has an arrow pointing from it. A flowchart ends with a *stop* symbol, which has one arrow pointing to it.

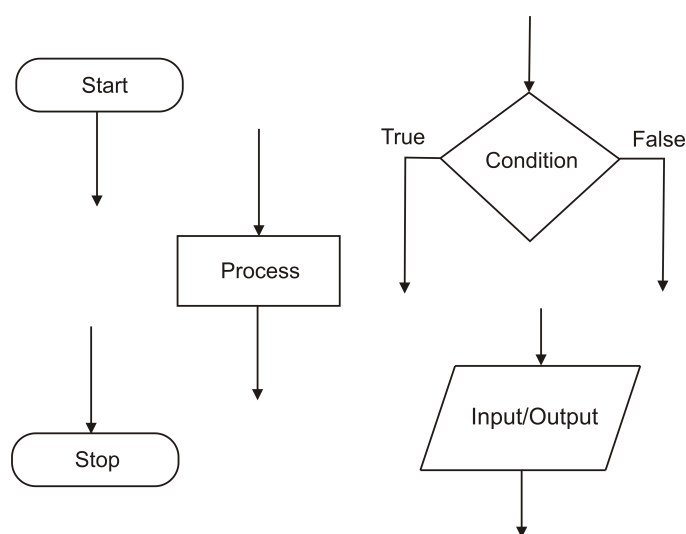


Figure 3: Simple flowchart symbols

The *process* symbol or *transformation* symbol is the most common and general symbol; it represents a design step. The symbol is shown as a rectangular box and represents any computation or sequence of computations carried out on some data. There is one arrow pointing to it and one arrow point from it.

Another flowchart symbol has the shape of a vertical diamond, and it represents a selection of alternate paths in the sequence of design steps. This symbol is also shown in Figure 3. This symbol is known as a decision block or a conditional block because the sequence of instructions can take one of two directions in the flowchart.

The flowchart symbol for a data input or output operation is shown in

Figure 4. There is one arrow pointing into the block and one arrow pointing out from the block.

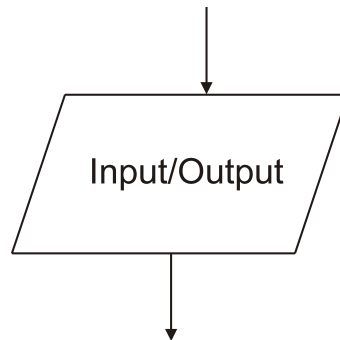


Figure 4: Flowchart data input/output symbol

For larger or more complex algorithms, flowcharts are used for the high-level description of the algorithms.

A simple flowchart with several basic symbols is shown in Figure 5.

6.2 Pseudo-code

Pseudo-code is a notation that uses English description for describing the algorithm that defines a problem solution. It can be used to describe relatively large and complex algorithms. Pseudo-code is much easier to understand and use than any programming language. It is relatively easy to convert the pseudo-code description of an algorithm to a high-level programming language.

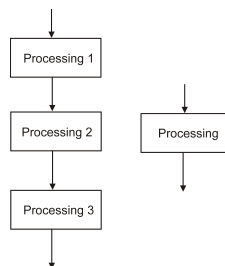


Figure 5: A simple flowchart

7 Algorithmic Structures

There are four design structures with which any algorithm can be described. These can be used with flowcharts and with pseudo-code notations. The basic design structures are:

- *Sequence*, any task can be broken down into a sequence of steps.
- *Selection*, part of the algorithm takes a decision and select one of several alternate paths of flow of actions. This structure is also known as alternation or conditional branch.
- *Repetition*, this part pf the algorithm has a set of steps that are to be executed zero, one, or more times. This structure is also known as looping.
- *Input-output*, the values of variables are read from an input device (such as the keyboard) or the values of the variables (results) are written to an output device (such as the screen)

7.1 Sequence

Figure 5 illustrates the sequence structure. A simple sequence of flowchart blocks is shown. The sequence structure is the most common and basic structure used in algorithmic design.

7.2 Selection

With the selection structure, one of several alternate paths will be followed based on the evaluation of a condition. Figure 6 illustrates the selection structure. In the figure, the actions in *Action step1* are executed when the condition is true. The instructions in *Action step2* are executed when the condition is false.

A flowchart example of the selection structure is shown in Figure 7. The condition of the selection structure is $X > 0$ and when this condition evaluates to true, the block with the action **increment j** will execute. Otherwise, the block with the action **decrement j** is executed.

A variation of the selection structure has multiple alternate paths, each one depends on the value of a variable. This structure is known as the *case* construct. Figure 8 illustrates this construct.

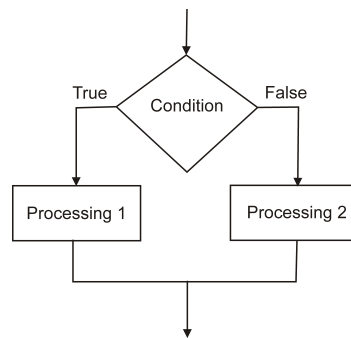


Figure 6: Selection structure in flowchart form

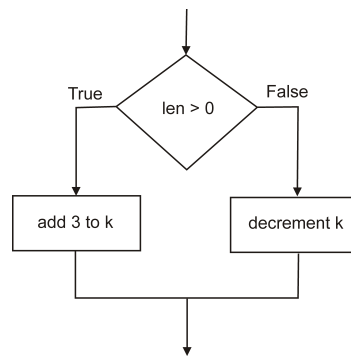


Figure 7: An example of the selection structure

7.3 Repetition

The repetition structure will cause a set of action steps to be repeated several times. Figure 9 shows this structure. The execution of the actions in *Action step1* are repeated while the condition is true. This structure is also known as the *while loop*.

A variation of the repetition structure is shown in Figure 10. The actions in *Action step1* are repeated until the condition becomes true. This structure is also known as the *repeat-until* loop.

8 Programming Statements

Programming languages are used to implement the solution that has been designed; this includes the data descriptions and the algorithm. A programming language has language *statements* that are used for data declarations

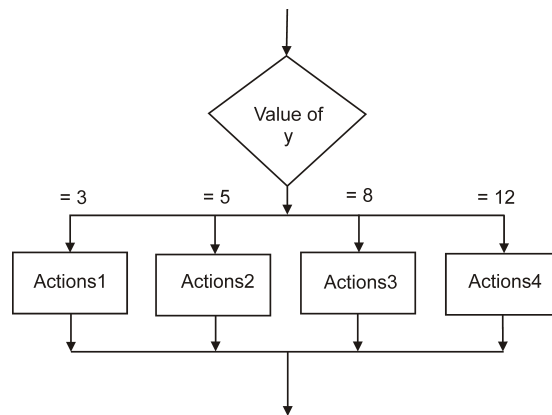
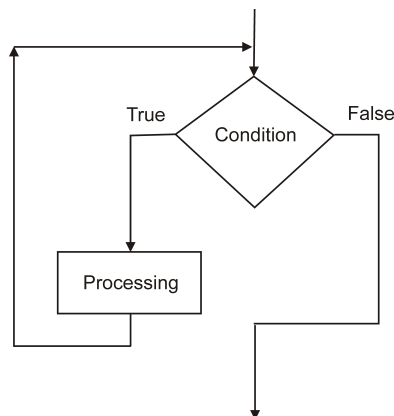
Figure 8: The *case* construct

Figure 9: Repetition structure in flowchart form

and for writing the instructions.

This section mainly deals with simple OOSimL statements, such as the assignment and I/O statements.

8.1 Assignment and Arithmetic Expressions

The most used statement is the *assignment* statement and is used to assign a value to a variable. The statement is written with the keyword **set**. This variable must be written on the left-hand side of the equal sign, which is the assignment operator.

The value to assign can be directly copied from another variable, from a

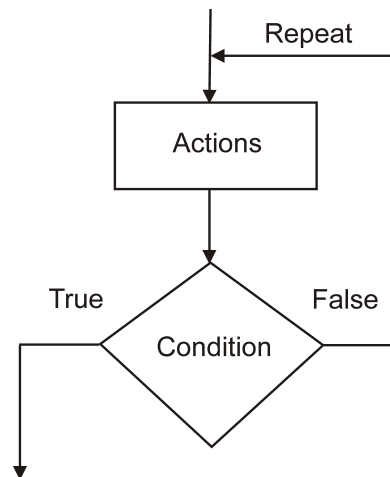


Figure 10: Repeat-until loop of the repetition structure

constant value, or from the value that results after evaluating an expression.

In the following example, the constant value 1435.5 is assigned to variable h and the value 100 to variable j .

```

variables
    float h
    float p
    float q
    integer j
begin
    . . .
    set h = 1435.5
    set j = 100
    set p = q + 14.5 * h
  
```

The third assignment statement in the example, variable p is assigned the result of evaluating the expression, $q + 14.5 * h$.

8.2 Simple Numeric Computing

Arithmetic expressions are normally used in an assignment statement. For example add 5 to the value of variable j , subtract the value of variable

h from variable p , the assignment statements with the simple arithmetic expressions are:

```
set j = j + 5
set p = p - h
```

In the example, in the first assignment, the new value of j is assigned by adding 5 to the previous value of j . In the second assignment, the new value of variable p is assigned the value that results after subtract the value of h from the previous value of p .

The OOSimL language provides the **add** and the **subtract** statements that are equivalent to the assignments of in the example. The following two statements show the same computation.

```
add 5 to j
subtract h from p
```

To add or subtract 1, the language provides the **increment** and the **decrement** statement. For example, **increment j**, adds the constant 1 to the value of variable j . **decrement k** subtracts the constant value 1 from the value of variable k .

The arithmetic expression used in the assignment statements discussed previously were very simple; only basic arithmetic operations appeared in the expressions. These are addition, subtraction, multiplication, and division.

To carry out more complex calculations, the language class library provides various numerical functions. Most of these functions are static, so to use a particular function of the *Math* library class, the name of the class must appear followed by a dot and then the name of the particular function invoked. Class *Math* is part of the class library supplied with the Java compiler. This class provides several mathematical functions, such as square root, exponentiation, trigonometric.

For example, consider the value of the expression $\cos x + y$ assigned to variable z and $\sqrt{t} - p$

The expression in the assignment statement use the mathematical functions *cos* and *sqrt* in class **Math**. The OOSimL statements are:

```
set z = Math.cos(x) + y
set y = Math.sqrt(t) - p
```

The value of the mathematical expression b^2 assigned to variable *area*, is coded as:

```
set area = Math.pow(b, 2)
```

8.3 Console I/O

The programming language OOSimL provides two basic statements for console input/output. The input statement reads a value of a variable from the input device (e.g., the keyboard). This statement is written with the keywords **read**, for input of a single variable. The output statement can be used for the output of a list of variables and literals, it is written with the keyword **display**. The general form of the input statement is:

```
read < variable_name >
```

The following example uses the **read** statement to read a value of variable *h*:

```
read h
```

The input statement implies an assignment statement for the variable *h*, because the variable changes its value to the new value that is read from the input device.

The output statement writes the value of one or more variables to the output device. The variable does not change their values. The general form of the output statement is:

```
display < data_list >
```

With the output statement, a list of data items can be displayed on the console. The following example displays on the console: a string literal, followed by the value of variable h , followed by another string literal, and followed by the value of variable q .

```
display "value of h: ", h, " value of q: ", q
```

9 Computing Area and Circumference

This problem computes the area and circumference of a circle, given its radius. The high-level algorithm description in informal pseudo-code notation is:

1. Read the value of the radius of a circle, from the input device.
2. Compute the area of the circle.
3. Compute the circumference of the circle.
4. Print or display the value of the area of the circle to the output device.
5. Print or display the value of the circumference of the circle to the output device.

A more detailed algorithm description follows:

1. Read the value of the radius r of a circle, from the input device.
2. Establish the constant π with value 3.14159.
3. Compute the area of the circle, $area = \pi \times r^2$.
4. Compute the circumference of the circle $cir = 2 \times \pi \times r$.
5. Print or display the value of $area$ of the circle to the output device.
6. Print or display the value of cir of the circle to the output device.

The following OOSimL program implements the data description and the algorithm for calculating the area of a circle. The example has a single class that includes function *main*. This class calls a function, *pow* of the library class, *Math*.

```

description
    This class computes the area and circumference of
    A circle, given its radius. This value is read
    using console I/O.
    */
class Circle_comp is
public
    function main is
        // data descriptions
    constants
        define PI = 3.14159 of type double
    variables
        define r of type float
        define area of type double
        define cir of type double
    begin
        // instructions starts here
        display "enter value of radius: "
        read r
        // compute area of circle
        set area = PI * Math.pow(r, 2)
        // compute circumference of circle
        set cir = 2.0 * PI * r
        // now print the results
        display "Area of circle is: ", area
        display "Circumference of circle is: ", cir
    endfun main
endclass Circle_comp

```

After compiling the program is executed and produces the following results on the console:

```

Enter value of radius: 2.75
Area of circle is: 23.758274375
Circumference of circle is: 17.278745

```

10 Selection Structure

The selection design structure is also known as alternation, because alternate paths are considered based on a condition.

10.1 Flowchart of Selection Structure

The selection structure provides the capability for decision-making. Because the selection design structure is better understood using a flowchart, Figure 11 is repeated here. The figure shows two possible paths for the execution flow. The condition is evaluated, and one of the paths is selected. If the condition is true, then the left path is selected and *Action step1* is performed. If the condition is false, the other path is selected and *Action step2* is performed.

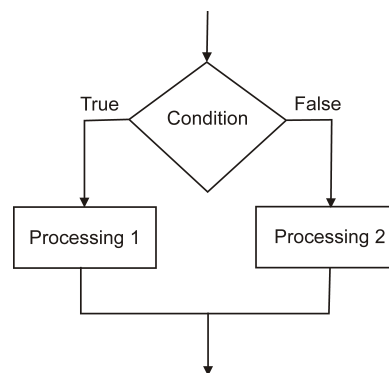


Figure 11: Flowchart of the Selection structure

10.2 IF Statement

The selection structure is written with an **if** statement, also known as an if-then-else statement. This statement includes three sections: the condition, the then-section, and the else-section. The else-section is optional. Several keywords are used in this statement: **if**, **then**, **else**, and **endif**. The general form of the **if** statement is:

```

if < condition >
  then
    < first sequence of statements >
  else
    < alternate sequence of statements >
endif
  
```

The **if** statement that corresponds to the selection structure in Figure 11 is:


```

if condition is true
  then
    perform instructions in Action step1
  else
    perform instructions in Action step2
endif

```

When the **if** statement executes, the condition is evaluated and only one of the two alternatives will be carried out: the one with the statements in *Action step1* or the one with the statements in *Action step2*.

10.3 Boolean Expressions

Boolean expressions are written to form conditions. A condition consists of an expression that evaluates to a truth-value, **true** or **false**.

A simple Boolean expression can be formed with the values of two data items and a relational operator. The following list of relational operators can appear in a Boolean expression:

- Less than, <
- Less or equal to, <=
- Greater than, >
- Greater or equal to, >=
- Equal, ==
- Not equal, !=

Examples of simple conditions are:

```

height > 23.75
a <= b
p == q

```

OOSimL supports not only conditions with the syntax in the previous example, but also provides additional keywords can be used for the relational operators. For example, the previous Boolean expressions can also be written as:

```

height greater than 23.75
a less or equal to b
p equal to q

```

10.4 Example of Selection

In this example, the condition to be evaluated is: $p \leq q$, and a decision is taken on how to update variable p . Figure 12 shows the flowchart for part of the algorithm that includes this selection structure.

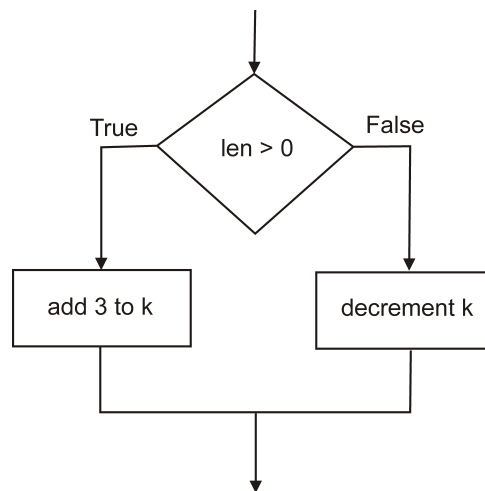


Figure 12: Example of selection structure

The OOSimL code for this example is:

```

if p <= q
then
    set p = q * x + 24.5
else
    set p = q
    set q = 12.75
endif

```

11 Example Program

This example computes the gross pay, taxes, and net pay of employees, given the hours worked and the hourly rate.

11.1 High-level Design

The overall description of the data and algorithm is shown in Figure 13. The flow of control in the selection structure depends on the condition: *gpay* **greater than** *TAXB*.

The high-level description of algorithm in pseudo-code follows:

1. Read values of employee name, employee number, hours worked, and hourly rate.
2. Compute gross pay: $\text{hours worked} \times \text{hourly rate}$
3. if gross pay > tax bracket then
 Compute tax amount: $(\text{gross pay} - \text{tax bracket}) \times \text{tax rate}$
 Compute net pay: $\text{gross pay} - \text{tax amount}$
 else compute net pay = gross pay
4. Display employee name, tax amount, net pay

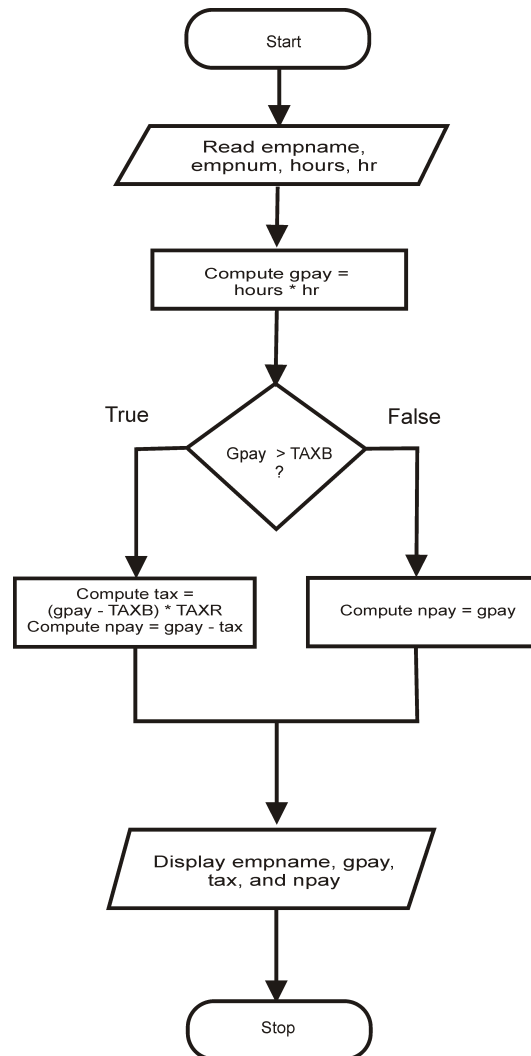


Figure 13: General problem description

11.2 Implementation

The implementation of the employee payment problem consists of two classes: *PayEmployee* and *Comp-pay*. The most interesting part of the solution for this problem includes the selection statement and is found in function *compute_pay* of class *PayEmployee*.

```

description
    This program computes the amount to pay an employee weekly
    after deducting taxes.
    The gross pay is computed from the number of hours worked
    by the employee, and the hourly rate.
    The income tax is computed as 14.75% of the amount earned
    that exceeds the tax bracket, $110.00
    This is the class for employees. The main attributes are
        emp_number, name, hours worked, and hourly rate.
    */
class PayEmployee is
    private
    constants

        define TAXR = 0.1475 of type float    // tax rate
        define TAXB = 110.00 of type float    // tax bracket
    variables
        define emp_name of type string
        define emp_number of type string
        define hours of type integer          // hours worked
        define hr of type float              // hourly rate
        define gpay of type float            // gross pay
        define tax of type float             // taxes
        define npay of type float            // net pay
    public
        //
    description
        This is the constructor, it initializes an object
        on creation.    */
    function initializer
    parameters iname of type string,
                inumb of type string,
                ihours of type integer,
                ihr of type float
    is
    begin
        set emp_number = inumb
        set hr = ihr
        set hours = ihours
        set emp_name = iname
    endfun initializer
    //
    description
        This funtion gets the tax amount paid by the
        employee object.    */

```

```

function get_tax return type float is
    begin
        return tax
    endfun get_tax
//
description
    This function returns the name of the employee object.
    */
function get_name return type string is
    begin
        return emp_name
    endfun get_name
//
description
    This function computes the gross pay, the taxes,
    and the net pay of an employee.
    */
function compute_pay is
    begin
        // compute gross pay
        set gpay = hours * hr
        //
        // compute tax and net pay
        if gpay > TAXB then
            set tax = (gpay - TAXB) * TAXR
            set npay = gpay - tax    // net pay
        else
            set npay = gpay
        endif
    endfun compute_pay
//
description
    This function returns the net pay for the
    PayEmployee object.    */
function get_npay return type float is
    begin
        return npay
    endfun get_npay
endclass PayEmployee
//

```

The following class, *Comp_pay* includes the function *main*, which creates and manipulates objects of class *PayEmployee*.

```

description
    This program computes the gross pay, tax amount,
    and net pay an employee;
    */
class Comp_pay is
    public

    description
        This is the main function of the application.
        */
    function main is
        variables
            define net_pay of type float
            define taxes of type float
            define empname of type string
            define empnum of type string
            define hours of type integer
            define hrate of type float
        object references
            define emp_obj of class PayEmployee
        begin

            display "Enter employee name: "
            read empname
            display "Enter employee number: "
            read empnum
            display "Enter hours worked: "
            read hours
            display "Enter hourly rate: "
            read hrate
            create emp_obj of class PayEmployee using empname,
                empnum, hours, hrate
            call compute_pay of emp_obj
            set net_pay = call get_npay of emp_obj
            set taxes = get_tax() of emp_obj // get tax amount paid
            display "Employee name: ", empname, " Net pay: ",
                net_pay, " Tax amount: ", taxes
        endfun main
    endclass Comp_pay

```

12 If Statement with Multiple Paths

The **if** statement with multiple paths is used to implement decisions involving more than two alternatives. The additional **elseif** clause is used to expand the number of alternatives. The general form of the *if* statement with k alternatives is:

```

if  $\langle \textit{condition} \rangle$ 
  then
     $\langle \textit{sequence1 of statements} \rangle$ 
  elseif  $\langle \textit{condition2} \rangle$ 
  then
     $\langle \textit{sequence2 of statements} \rangle$ 
  elseif  $\langle \textit{condition3} \rangle$ 
  then
     $\langle \textit{sequence3 of statements} \rangle$ 
    . . .
  else
     $\langle \textit{sequencek of statements} \rangle$ 
endif

```

The conditions in a multiple-path **if** statement are evaluated from top to bottom until one of the conditions evaluates to true. The following example applies the **if** statement with four paths.

```

if height > 6.30
then increment group1
elseif height > 6.15
then increment group2
elseif height > 5.85
then increment group3
elseif height > 5.60
then increment group4
else increment group5
endif

```

13 Using Logical Operators

With the logical operators, complex Boolean expressions can be constructed. The logical operators are: **and**, **or**, and **not**. These logical operators help to

construct complex (or compound) conditions from simpler conditions. The general form of a complex conditions using the **or** operator and two simple conditions, *cond1* and *cond2*, is:

```
cond1 and cond2
cond1 or cond2
not cond1
```

The following example includes the **and** logical operator:

```
if x >= y and p == q
then
    < statements_1 >
else
    < statements_2 >
endif
```

The condition can also be written in the following manner:

```
if x greater or equal to y and p is equal to q
...
endif
```

The following example applies the **not** operator:

```
not (p > q)
```

14 The Case Statement

The case structure is a simplified version of the selection structure with multiple paths. The **case** statement evaluates the value of a single variable or simple expression of type **integer** or of type **character** and selects the appropriate path. The case statement also supports compound statements, that is, multiple statements instead of a single statement in one or more of the selection options. The general form of this statement is:

```
case < selector_variable > of
    value variable_value : < statements >
    ...
endcase
```

In the following example, the temperature status of a boiler is monitored and the temperature status is stored in variable *temp_status*. The following case statement first evaluates the temperature status in variable *temp_status* and then assigns an appropriate string literal to variable *mess*, finally this variable is displayed on the console.

The example assumes that the variables involved have an appropriate declaration, such as:

```
variables
  define temp_status of type character
  define mess of type string
  ...
  case temp_status of
    value 'E': mess = "Extremely dangerous"
    value 'D': mess = "Dangerous"
    value 'H': mess = "High"
    value 'N': mess = "Normal"
    value 'B': mess = "Below normal"
  endcase
  . . .
  display "Temperature status: ", mess
```

The default option of the **case** statement can also be used by writing the keywords **default** or **otherwise** in the last case of the selector variable. For example, the previous example can be enhanced by including the default option in the case statement:

```
case temp_status of
  value 'E': mess = "Extremely dangerous"
  value 'D': mess = "Dangerous"
  value 'H': mess = "High"
  value 'N': mess = "Normal"
  value 'B': mess = "Below normal"
  otherwise
    mess = "Temp rising"
endcase
. . .
display "Temperature status: ", mess
```

15 The While Loop Structure

The *while* loop structure consists of a block of code and a condition. The condition is first evaluated – if the condition is true the code within the block is then executed. This repeats until the condition evaluates to false. The while-loop structure checks the condition before the block of code is executed

15.1 While-Loop Flowchart

A flowchart with the *while* loop structure is shown in Figure 14. The block of code consists of a sequence of actions.

The actions in the code block are performed while the condition is true. After the actions are performed, the condition is again evaluated, and the actions are again performed if the condition is still true. This continues until the condition changes to false, at which point the loop terminates.

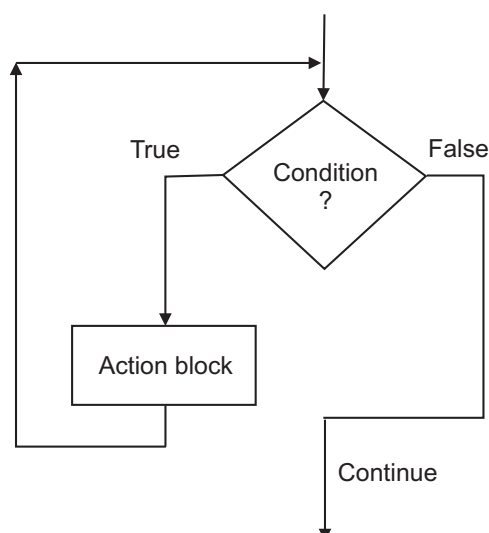


Figure 14: A flowchart of the while-loop

The condition is tested first, and then the block of code is performed. If this condition is initially false, the actions in the code block are not performed.

The number of times that the loop is performed is normally a finite integer value. A well-defined loop will eventually terminate. The condition

is sometimes called the *loop condition*, and it determines when the loop terminates. A non-terminating loop is defined in special cases and will repeat the actions forever.

A counter variable has the purpose of storing the number of times (iterations) that the actions are repeated. The counter variable is of type integer and is incremented (or decremented) on each repetition. The counter variable must be initialized to a given value.

15.2 The While Statement

The **while** statement includes the condition (a Boolean expression), the statements in the code block, and the keywords **while**, **do**, and **endwhile**. The code block is placed after the **do** keyword and before the **endwhile** keyword. The following lines of code show the general form of the while-loop statement that corresponds to the portion of flowchart shown in Figure 14.

```

while < condition > do
    < statements in code block >
endwhile

```

In the following example, a **while** statement appears with a counter variable, *j*. This counter variable is used to control the number of times the repeat group will be performed.

```

constants
    define MAX_NUM = 15 of type integer    // maximum number
                                           // of times through the loop

variables
    define j of type integer              // loop counter
    define sum of type float
    ...
begin
    set j = 1    // initial value
    while j <= MAX_NUM do
        set sum = sum + 12.5
        increment j
    endwhile
    display "Value of sum: ", sum
    ...

```

16 Employee Pay Problem with Repetition

This problem uses repetition to compute the gross pay and net pay to a given number of employees, *num*, which is read from the console. The processing for each employee has been discussed previously.

In the flowchart, the actions performed to compute gross and net pay for every employee is included in the code block and placed in a while loop. In the program, the **while** statement is used. The condition compares the loop counter, *j* with the given number of employees read, *num*. The condition for the repetition (loop) is: $j \leq Num$.

The value of variable *num* represents the number of employees to process. Figure 15 shows the flowchart for the algorithm for the problem with repetition.

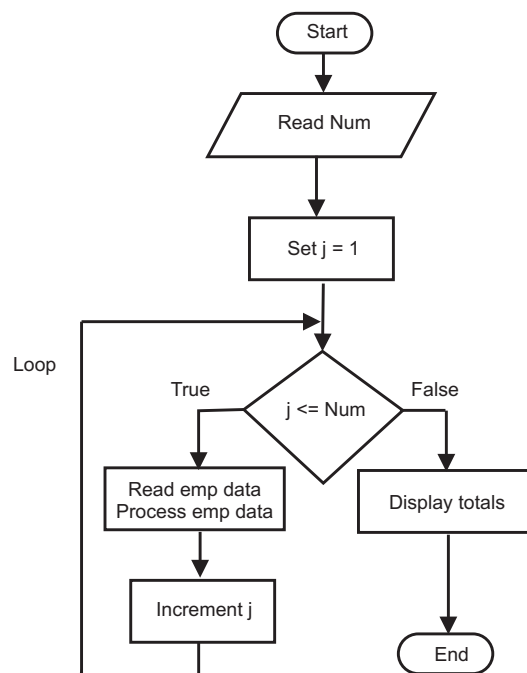


Figure 15: Employee pay problem with repetition

The following class includes the modified function *main*, which has the while loop. The other class, *PayEmployee*, has not been changed.

description

```

This program processes the net pay for a given
number of employees. For every employee,
it computes the gross pay, tax amount, and net
pay an employee.
*/
class Comp_pay is
public
description
    This is the main function of the application.
    */
function main is
variables
    define num of type integer
    define j of type integer
    //
    define net_pay of type float
    define taxes of type float
    define empname of type string
    define empnum of type string
    define hours of type integer
    define hrate of type float
    define total_taxes of type float
    define total_npay of type float
object references
    define emp_obj of class PayEmployee
begin
    display "Enter number of employees to process"
    read num
    set j = 1
    set total_taxes = 0.0
    set total_npay = 0.0
    while j <= num do
        display "Enter employee name: "
        read empname
        display "Enter employee number: "
        read empnum
        display "Enter hours worked: "
        read hours
        display "Enter hourly rate: "
        read hrate
        create emp_obj of class PayEmployee
            using empname, empnum, hours, hrate
        call compute_pay of emp_obj
        set net_pay = call get_npay of emp_obj
        set taxes = get_tax() of emp_obj    // get tax amount
    end while
end function

```

```

        display "Employee name: ", empname,
        " Net pay: ", net_pay, " Tax amount: ", taxes
        increment j
        add taxes to total_taxes
        add net_pay to total_npay
    endwhile
    display "Total taxes: ", total_taxes
    display "total net pay: ", total_npay
endfun main
endclass Comp_pay

```

Variables *total_taxes* and *total_npay* accumulate on every iteration in the while loop, the tax amount and the net pay respectively. These variables are often known as accumulator variables.

17 Loop Until

The loop *until* structure is a control flow statement that allows code to be executed repeatedly based on a given condition. The structure consists of a block of code and the condition.

The actions within the block are executed first, and then the condition is evaluated. If the condition is not true the actions within the block are executed again. This repeats until the condition becomes true.

Loop until structures check the condition after the block is executed, this is an important difference with while loop, which tests the condition before the actions within the block are executed. Figure 16 shows the flowchart for the loop-until structure.

The OOSimL statement of the loop-until structure, corresponds directly with the flowchart in Figure 16 and uses the keywords **repeat**, **until**, and **endrepeat**. The following portion of code shows the general form of the loop-until statement.

```

repeat
    < statements in Action block >
until < condition >
endrepeat

```

The following example shows the code of a loop until statement for the pay employee problem.

```

set j = 0

```

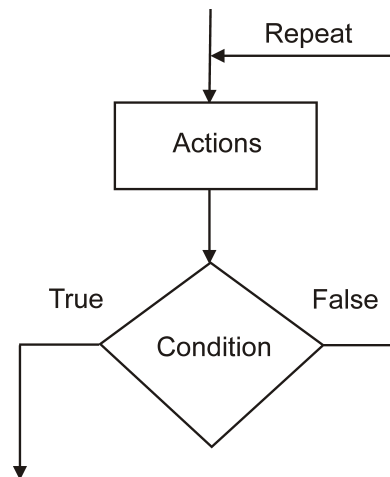


Figure 16: Loop-until structure

```

repeat
  display "Enter employee name: "
  read empname
  display "Enter employee number: "
  read empnum
  display "Enter hours worked: "
  read hours
  display "Enter hourly rate: "
  read hrate
  create emp_obj of class PayEmployee
    using empname, empnum, hours, hrate
  call compute_pay of emp_obj
  set net_pay = call get_npay of emp_obj
  set taxes = get_tax() of emp_obj // get tax amount
  display "Employee name: ", empname,
    " Net pay: ", net_pay, " Tax amount: ", taxes
  increment j
  add taxes to total_taxes
  add net_pay to total_npay
until j > num
endrepeat

```


18 For Loop Structure

The third type of loop structure is the *for* loop. It explicitly uses a loop counter; the initial value and the final value of the loop counter have to be indicated. The *for* loop is most useful when the number of times that the loop is carried out is known in advance. The **for** statement has the keywords: **for**, **to**, **downto**, **do**, and **endfor**. The **for** statement has the general form:

```

for  $\langle counter \rangle = \langle initial\_value \rangle$  to  $\langle final\_value \rangle$ 
  do
    Action block
  endfor

```

On every iteration, the loop counter is automatically incremented. The last time through the loop, the loop counter has its final value allowed. In other words, when the loop counter reaches its final value, the loop terminates. The *for* loop is similar to the *while* loop in that the condition is evaluated before carrying out the operations in the repeat loop.

The following portion of code uses a **for** statement for the repetition part of the employee pay problem.

```

for j = 1 to num do
  display "Enter employee name: "
  read empname
  display "Enter employee number: "
  read empnum
  display "Enter hours worked: "
  read hours
  display "Enter hourly rate: "
  read hrate
  create emp_obj of class PayEmployee
    using empname, empnum, hours, hrate
  call compute_pay of emp_obj
  set net_pay = call get_npay of emp_obj
  set taxes = get_tax() of emp_obj // get tax amount
  display "Employee name: ", empname,
    " Net pay: ", net_pay, " Tax amount: ", taxes
  increment j
  add taxes to total_taxes
  add net_pay to total_npay
endfor

```