



PSEUDOCODE STANDARDS

Pseudocode is a kind of structured English for describing algorithms. It allows the designer to focus on the logic of the algorithm without being distracted by details of language syntax. At the same time, the pseudocode needs to be complete. It describes the entire logic of the algorithm so that implementation becomes a rote mechanical task of translating logic into source code. A line of pseudocode translates to one or many lines of actual code (normally, it is the latter than former).

In general the vocabulary used in the pseudocode should be the vocabulary of the problem domain, not of the implementation domain. The pseudocode is a narrative for someone who knows the requirements (problem domain) and is trying to learn how the solution is organized. E.g.,

Extract the next word from the line (good)
set word to get next token (poor)

Append the file extension to the name (good)
name = name + extension (poor)

FOR all the characters in the name (good)
FOR character = first to last (ok)

Note that the logic must be decomposed to the level of a single loop or decision. Thus "Search the list and find the customer with highest balance" is too vague because it takes a loop AND a nested decision to implement it. It's okay to use "Find" or "Lookup" if there's a predefined function for it such as `String.indexOf()`.

Each textbook and each individual designer may have their own personal style of pseudocode. Pseudocode is not a rigorous notation, since it is read by other people, not by the computer. There is no universal "standard" for the industry, but for instructional purposes it is helpful if we all follow a similar style. The format below is recommended for expressing your solutions in our class.

The "structured" part of pseudocode is a notation for representing **six** specific structured programming constructs: SEQUENCE, WHILE, IF-THEN-ELSE, REPEAT-UNTIL, FOR (or FOR EACH), and CASE. Each of these constructs can be embedded inside any other construct. These constructs represent the logic, or flow of control in an algorithm.

It has been proven that three basic constructs for flow of control are sufficient to implement any "proper" algorithm.

SEQUENCE is a linear progression where one task is performed sequentially after another.

WHILE is a loop (repetition) with a simple conditional test at its beginning.

IF-THEN-ELSE is a decision (selection) in which a choice is made between two alternative courses of action.

Although these constructs are sufficient, it is often useful to include three more constructs:

REPEAT-UNTIL is a loop with a simple conditional test at the bottom.

CASE is a multiway branch (decision) based on the value of an expression. CASE is a generalization of IF-THEN-ELSE.

FOR (or FOR EACH) is a "counting" loop.

SEQUENCE

Sequential control is indicated by writing one action after another, each action on a line by itself, and all actions aligned with the same indent. The actions are performed in the sequence (top to bottom) that they are written.

Example (non-computer)

Brush teeth
Wash face
Comb hair
Smile in mirror

Example

READ height of rectangle
READ width of rectangle
COMPUTE area as height times width

Common Action Keywords

Several keywords are often used to indicate common input, output, and processing operations.

Input: READ, OBTAIN, GET

Output: PRINT, DISPLAY, SHOW

Compute: COMPUTE, CALCULATE, DETERMINE

Initialize: SET, INIT

Add one: INCREMENT, BUMP

List manipulation: ADD, REMOVE, PUSH, POP

IF-THEN-ELSE

Binary choice on a given Boolean condition is indicated by the use of four keywords: IF, THEN, ELSE, and ENDIF. The general form is:

```
IF condition THEN
    sequence 1
ELSE
    sequence 2
ENDIF
```

The ELSE keyword and "sequence 2" are optional. If the condition is true, sequence 1 is performed, otherwise sequence 2 is performed.

Example

```
IF HoursWorked > NormalMax THEN
    Display overtime message
ELSE
    Display regular time message
ENDIF
```

WHILE

The WHILE construct is used to specify a loop with a test at the top. The beginning and ending of the loop are indicated by two keywords WHILE and ENDWHILE. The general form is:

```
WHILE condition
    sequence
ENDWHILE
```

The loop is entered only if the condition is true. The "sequence" is performed for each iteration. At the conclusion of each iteration, the condition is evaluated and the loop continues as long as the condition is true.

Example

```
WHILE Population < Limit
    Compute Population as Population + Births - Deaths
ENDWHILE
```

Example

```
WHILE employee.type NOT EQUAL manager AND personCount < numEmployees
    INCREMENT personCount
    CALL employeeList.getPerson with personCount RETURNING employee
ENDWHILE
```

CASE

A CASE construct indicates a multiway branch based on conditions that are mutually exclusive. Four keywords, CASE, OF, OTHERS, and ENDCASE, and conditions are used to indicate the various alternatives. The general form is:

```
CASE expression OF
    condition 1 :
        sequence 1
    condition 2 :
        sequence 2
    ...
    condition n :
        sequence n
    OTHERS:
        default sequence
ENDCASE
```

The OTHERS clause with its default sequence is optional. Conditions are normally numbers or characters

indicating the value of "expression", but they can be English statements or some other notation that specifies the condition under which the given sequence is to be performed. A certain sequence may be associated with more than one condition.

Example

```
CASE Title OF
  Mr   : Print "Mister"
  Mrs  : Print "Missus"
  Miss : Print "Miss"
  Ms   : Print "Mizz"
  Dr   : Print "Doctor"
ENDCASE
```

Example

```
CASE grade OF
  A   : points = 4
  B   : points = 3
  C   : points = 2
  D   : points = 1
  F   : points = 0
ENDCASE
```

REPEAT-UNTIL

This loop is similar to the WHILE loop except that the test is performed at the bottom of the loop instead of at the top. Two keywords, REPEAT and UNTIL are used. The general form is:

```
REPEAT
  sequence
UNTIL condition
```

The "sequence" in this type of loop is always performed at least once, because the test is performed after the sequence is executed. At the conclusion of each iteration, the condition is evaluated, and the loop repeats if the condition is false. The loop terminates when the condition becomes true.

FOR (or FOR EACH)

This loop is a specialized construct for iterating a specific number of times, often called a "counting" loop. Two keywords, FOR and ENDFOR are used. The general form is:

```
FOR iteration bounds
  sequence
ENDFOR
```

or

```
FOR EACH item IN collection
```

```
        sequence with item
    ENDFOR
```

In cases where the loop constraints can be obviously inferred it is best to describe the loop using problem domain vocabulary.

Example

```
FOR each month of the year (good)
FOR month = 1 to 12 (ok)
```

```
FOR each employee in the list (good)
FOR empno = 1 to listsize (ok)
```

NESTED CONSTRUCTS

The constructs can be embedded within each other, and this is made clear by use of indenting. Nested constructs should be clearly indented from their surrounding constructs.

Example

```
SET total to zero
REPEAT
    READ Temperature
    IF Temperature > Freezing THEN
        INCREMENT total
    END IF
UNTIL Temperature < zero
Print total
```

In the above example, the IF construct is nested within the REPEAT construct, and therefore is indented.

INVOKING SUBPROCEDURES

Use the `CALL` keyword. For example:

```
CALL AvgAge with StudentAges
CALL Swap with CurrentItem and TargetItem
CALL Account.debit with CheckAmount
CALL getBalance RETURNING aBalance
CALL SquareRoot with orbitHeight RETURNING nominalOrbit
```

EXCEPTION HANDLING

```
BEGIN
    statements
EXCEPTION
    WHEN exception type
        statements to handle exception
    WHEN another exception type
        statements to handle exception
```

END

Sample Pseudocode

"Adequate"

```
FOR X = 1 to 10
  FOR Y = 1 to 10
    IF gameBoard[X][Y] = 0
      Do nothing
    ELSE
      CALL theCall(X, Y) (recursive method)
      increment counter
    END IF
  END FOR
END FOR
```

"Better"

```
Set moveCount to 1
FOR each row on the board
  FOR each column on the board
    IF gameBoard position (row, column) is occupied THEN
      CALL findAdjacentTiles with row, column
      INCREMENT moveCount
    END IF
  END FOR
END FOR
```

(Note: the logic is restructured to omit the "do nothing" clause)

"Not So Good"

```
FOR all the number at the back of the array
  SET Temp equal the addition of each number
  IF > 9 THEN
    get the remainder of the number divided by 10 to that index
    and carry the "1"
  Decrement one
Do it again for numbers before the decimal
```

"Good Enough (not perfect)"

```
SET Carry to 0
FOR each DigitPosition in Number from least significant to most significant
```

```

    COMPUTE Total as sum of FirstNum[DigitPosition] and SecondNum[DigitPosition] and
    Carry

    IF Total > 10 THEN
        SET Carry to 1
        SUBTRACT 10 from Total
    ELSE
        SET Carry to 0
    END IF

    STORE Total in Result[DigitPosition]

END LOOP

IF Carry = 1 THEN
    RAISE Overflow exception
END IF

```

"Pretty Good" This example shows how pseudocode is written as comments in the source file. Note that the double slashes are indented.

```

public boolean moveRobot (Robot aRobot)
{
    //IF robot has no obstacle in front THEN
        // Call Move robot
        // Add the move command to the command history
        // RETURN true
    //ELSE
        // RETURN false without moving the robot
    //END IF
}

```

Example Java Implementation

- source code statements are interleaved with pseudocode.
- comments that correspond exactly to source code are removed during coding.

```

public boolean moveRobot (Robot aRobot)
{
    //IF robot has no obstacle in front THEN
    if (aRobot.isFrontClear())
    {
        // Call Move robot
        aRobot.move();
        // Add the move command to the command history
        cmdHistory.add(RobotAction.MOVE);
        return true;
    }
    else // don't move the robot

```

```
{  
    return false;  
} //END IF  
}
```