

Pseudocode: An Introduction

Flowcharts were the first design tool to be widely used, but unfortunately they do not very well reflect some of the concepts of structured programming. Pseudocode, on the other hand, is a newer tool and has features that make it more reflective of the structured concepts. Unfortunately, the narrative presentation is not as easy to understand and follow.

RULES FOR PSEUDOCODE

1. Write only one stmt per line

Each stmt in your pseudocode should express just one action for the computer. If the task list is properly drawn, then in most cases each task will correspond to one line of pseudocode.

EX: TASK LIST:

Read name, hourly rate, hours worked, deduction rate

Perform calculations

gross = hourlyRate * hoursWorked

deduction = grossPay * deductionRate

net pay = grossPay – deduction

Write name, gross, deduction, net pay

PSEUDOCODE:

READ name, hourlyRate, hoursWorked, deductionRate

grossPay = hourlyRate * hoursWorked

deduction = grossPay * deductionRate

netPay = grossPay – deduction

WRITE name, grossPay, deduction, netPay

2. Capitalize initial keyword

In the example above, **READ** and **WRITE** are in caps. There are just a few keywords we will use:

READ, WRITE, IF, ELSE, ENDIF, WHILE, ENDWHILE, REPEAT, UNTIL

3. Indent to show hierarchy

We will use a particular indentation pattern in each of the design structures:

SEQUENCE: keep statements that are “stacked” in sequence all starting in the same column.

SELECTION: indent the statements that fall inside the selection structure, but not the keywords that form the selection

LOOPING: indent the statements that fall inside the loop, but not the keywords that form the loop

EX: In the example above, employees whose *grossPay* is less than 100 do not have any deduction.

TASK LIST:

Read name, hourly rate, hours worked, deduction rate

Compute gross, deduction, net pay

 Is gross \geq 100?

 YES: calculate deduction

 NO: no deduction

Write name, gross, deduction, net pay

PSEUDOCODE:

READ name, hourlyRate, hoursWorked

grossPay = hourlyRate * hoursWorked

IF grossPay \geq 100

 deduction = grossPay * deductionRate

ELSE

 deduction = 0

ENDIF

netPay = grossPay – deduction

WRITE name, grossPay, deduction, netPay

4. End multiline structures

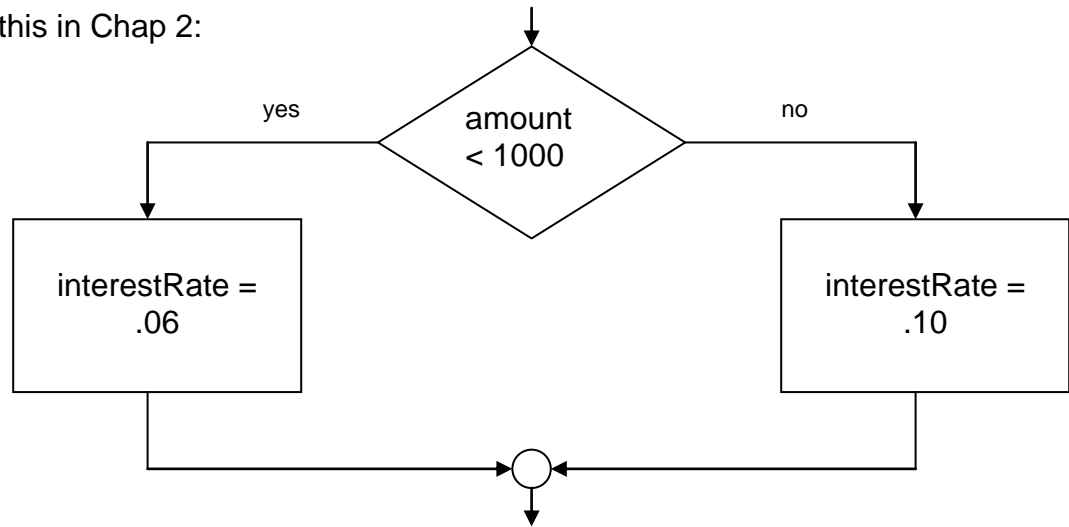
See how the IF/ELSE/ENDIF is constructed above. The ENDIF (or END whatever) always is in line with the IF (or whatever starts the structure).

5. Keep stmts language independent

Resist the urge to write in whatever language you are most comfortable with. In the long run, you will save time! There may be special features available in the language you plan to eventually write the program in; if you are SURE it will be written in that language, then you can use the features. If not, then avoid using the special features.

SELECTION STRUCTURE

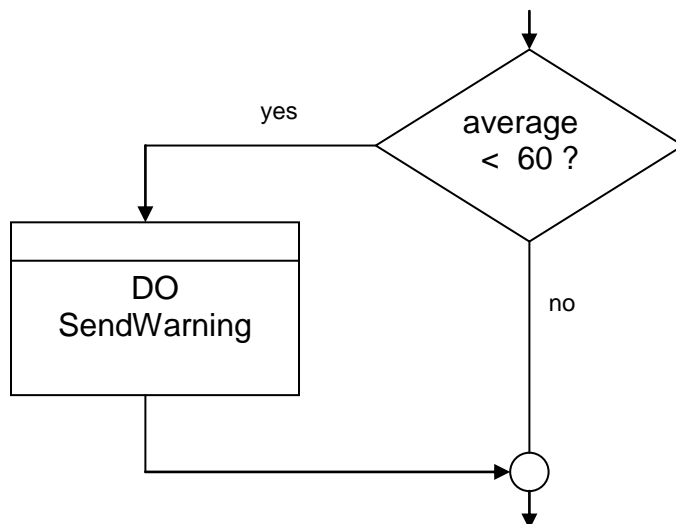
We looked at this in Chap 2:



The **pseudocode** for this would be:

```
IF amount < 1000
    interestRate = .06           // the "yes" or "true" action
ELSE
    interestRate = .10         // the "no" or "false" action
ENDIF
```

Some selections are of the "do it or don't" (one sided) variety. For example:



The pseudocode for this is:

```
IF average < 60
    DO SendWarning
ENDIF
```

It is considered **poor form** to have a 1-sided IF stmt where the action is on the “no” or ELSE side. Consider this code:

```
IF average < 60
    NULL
ELSE
    DO GivePassingGrade
ENDIF
```

This could (**and should**) be rewritten to eliminate the NULL “yes” part. To do that, we change the < to its opposite: >= as follows:

```
IF average >= 60
    DO GivePassingGrade
ENDIF
```

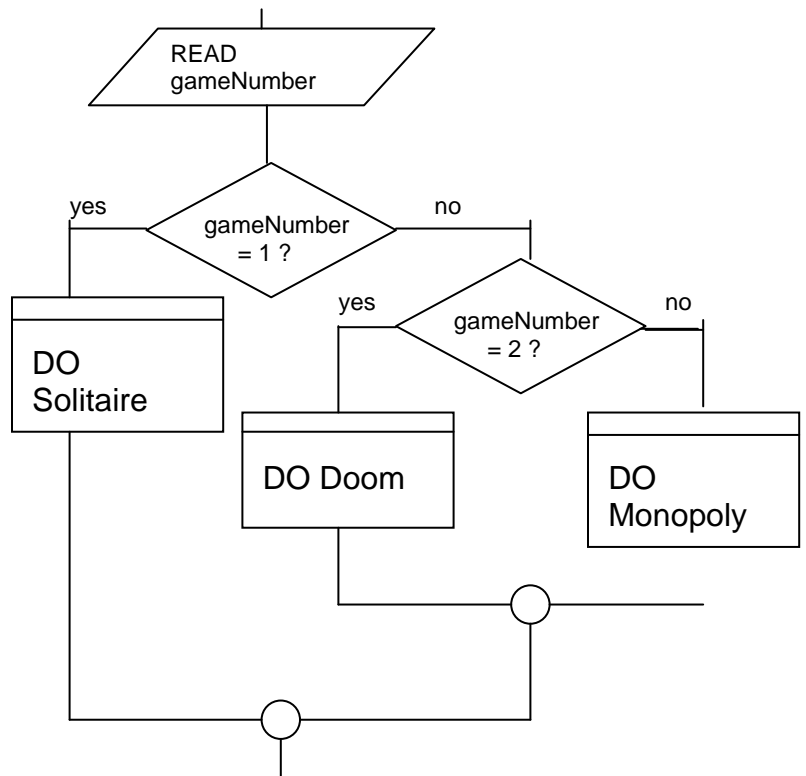
NESTING IF STATEMENTS

What if we wanted to put a little menu up on the screen:

1. Solitaire
2. Doom
3. Monopoly

and have the user select which game to play. How would we activate the correct game?

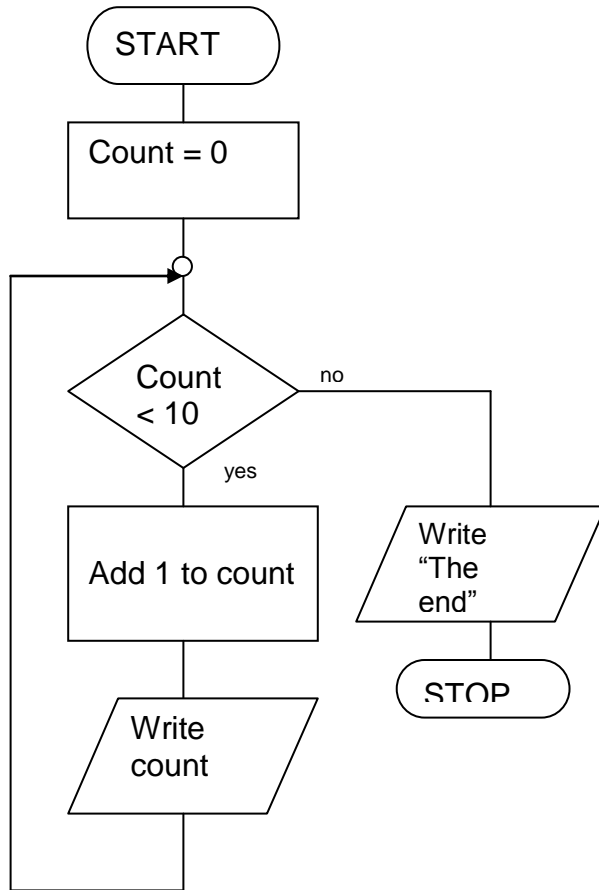
```
READ gameNumber
IF gameNumber = 1
    DO Solitaire
ELSE
    IF gameNumber = 2
        DO Doom
    ELSE
        DO Monopoly
    ENDIF
ENDIF
ENDIF
```



We must pay particular attention to where the IFs end. The nested IF must be completely contained in either the IF or the ELSE part of the containing IF. Watch for and line up the matching ENDIF.

LOOPING STRUCTURES

One of the most confusing things for students first seeing a flowchart is telling the loops apart from the selections. This is because both use the diamond shape as their control symbol. In pseudocode this confusion is eliminated. To mark our loops we will use these pairs: **WHILE / ENDWHILE** **REPEAT / UNTIL**



The loop shown here (from the last chapter) will have the following pseudocode:

```

count = 0
WHILE count < 10
    ADD 1 to count
    WRITE count
ENDWHILE
WRITE "The end"
  
```

Notice that the connector and test at the top of the loop in the flowchart become the WHILE stmt in pseudocode. The end of the loop is marked by ENDWHILE.

What statement do we execute when the loop is over? The one that follows the ENDWHILE.

Sometimes it is desirable to put the steps that are inside the loop into a separate module. Then the pseudocode might be this:

Mainline

```

count = 0
WHILE count < 10
    DO Process
ENDWHILE
WRITE "The end"
  
```

We often use this name for the first module. Initialization comes first

The processing loop uses this module

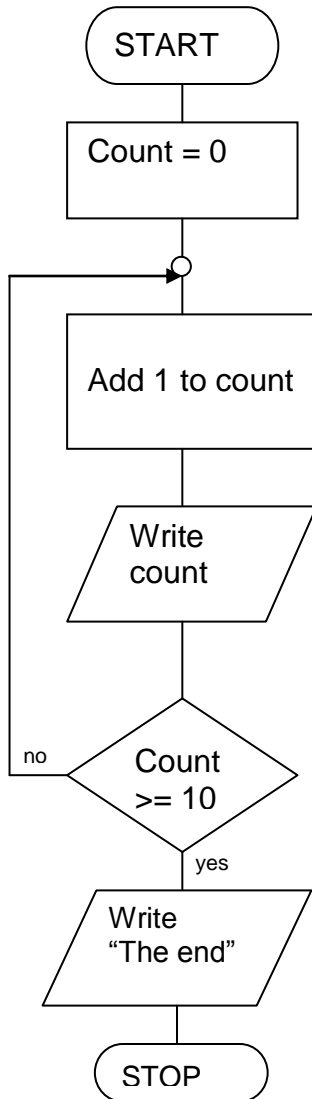
Termination does clean-up

Process

```

ADD 1 to count
WRITE count
  
```

Go thru these steps and then return to the module that sent you here (Mainline)



This time we will see how to write pseudocode for an UNTIL loop:

```
count = 0
REPEAT
    ADD 1 to count
    WRITE count
UNTIL count >= 10
WRITE "The end"
```

Notice how the connector at the top of the loop corresponds to the REPEAT keyword, while the test at the bottom corresponds to the UNTIL stmt. When the loop is over, control goes to the stmt following the UNTIL.

ADVANTAGES AND DISADVANTAGES

Pseudocode Disadvantages

- It's not visual
- There is no accepted standard, so it varies widely from company to company

Pseudocode Advantages

- Can be done easily on a word processor
- Easily modified
- Implements structured concepts well

Flowchart Disadvantages

- Hard to modify
- Need special software (not so much now!)
- Structured design elements not all implemented

Flowchart Advantages

- Standardized: all pretty much agree on the symbols and their meaning
- Visual (but this does bring some limitations)

HOW WE STORE AND ACCESS DATA

What happens when we execute READ stmt?

 READ name, hoursWorked, hourlyRate, deductionRate

The computer stores all program data into memory locations. It knows these location by their addresses. It is perfectly able to understand code like:

 READ 19087, 80976, 10943, 80764

but we would have a hard time with it. So we name our storage locations using words that are descriptive to us. Every language has its own (different) set of rules about how these names are formed. **We will use a simple style:**

- variable names will start with a lowercase letter
- they will have no spaces in them
- additional words in them will start with a capital
- names must be unique within the program
- consistent use of names

The READ statement tells the computer to get a value from the input device (keyboard, file, ...) and store it in the names memory location.

When we need to compute a value in a program (like *grossPay*) we will use what is called an assignment stmt.

 variable = expression

Be careful to understand the difference between these two stmts:

 num1 = num2

 num2 = num1

The WRITE stmt is used to display information on the output device (screen, printer). To display words, enclose them in quotes. A variable's value will be displayed. So if the variable *name* currently contains John Smith, then the stmt

WRITE "Employee name: ", name
will output like this:
Employee name: John Smith

CALCULATION SYMBOLS

We will often have to represent an expression like the one that computes *grossPay*. To symbolize the arithmetic operators we use these symbols

grouping	()
exponent	** or ^
multiply	*
divide	/
add	+
subtract	-

There is a precedence or hierarchy implied in these symbols.

ORDER OF EXECUTION

()	equations in parenthesis
**	exponentiation
/ *	division and multiplication
+ -	addition and subtraction

Please Excuse My Dear Aunt Sally

Note: when operators of equal value are to be executed, the order of execution is left to right.

Examples:

$$\text{AREA} = R^2$$

$$\text{SUM} = A^2 + B^2$$

$$\text{PERIM} = 2(L + W)$$

$$\frac{A}{B+C} \quad \frac{A}{B} + C \quad \frac{A}{B}C \quad \frac{A}{BC} \quad \frac{D}{B^2} + C \quad \frac{D^2}{B+C}$$

$$\text{value} = 100 * 2 / 5 - 3 = 200 / 5 - 3 = 40 - 3 = 37$$

$$\text{value} = 100 * 2 / (5 - 3) = 100 * 2 / 2 = 200 / 2 = 100$$

$$\text{value} = 100 * ((2/5) - 3) = 100 * (.4 - 3) = 100 * -2.6 = -260$$

SELECTION

When we have to make a choice between actions, we almost always base that choice on a test. The test uses phrases like “is less than” or “is equal to”. There is a universally accepted set of symbols used to represent these phrases:

- > (greater than)
- < (less than)
- >= (greater than or equal to)
- <= (less than or
- = (equal to)
- <> (not equal to)

It is interesting to notice that these can be paired up:

SYMBOL	IS OPPOSITE TO
>	<=
<	>=
=	<>

LOGICAL OPERATORS: AND, OR, NOT

AND: if any of the conditions are false, the whole expression is false.

```
ex:  IF day = "Saturday" AND weather = "sunny"  
      WRITE "Let's go to the beach!"  
      ENDIF
```

OR: if any of the conditions are true, the whole expression is true

```
ex:  IF month = "June" OR month = "July" OR month = "August"  
      WRITE "Yahoo! Summer vacation!"  
      ENDIF
```

NOT: reverses the outcome of the expression; true becomes false, false becomes true.

```
ex:  IF day <> "Saturday" AND day <> "Sunday"  
      WRITE "Yuk, another work day"  
      ENDIF
```