

Cambridge Pseudocode Handout

English edition

1 Style and layout

1.1 Keywords and case

Pseudocode is the language Cambridge writes algorithms in. It is not a real programming language — no computer runs it — but the exam marks it as if it were, so the form matters.

Every keyword is written in **upper case**: IF, REPEAT, PROCEDURE, OUTPUT.

```
DECLARE Count : INTEGER
Count ← 3
IF Count > 0
    THEN
        OUTPUT "Ready"
ENDIF
```

- Write IF, never if or If.
- A keyword can never be used as a name of your own: you may not call a variable FOR.

Common mistakes

- Lower-case keywords. The mark scheme expects upper case.
- Inventing keywords. Only the words in this handout are in the syllabus.

1.2 Identifiers

An **identifier** is the name you give a variable, a constant, a procedure or a function.

```
DECLARE NumberOfPlayers : INTEGER
NumberOfPlayers ← 4
OUTPUT NumberOfPlayers
```

Four rules, all from the syllabus:

- It must **start with a capital letter**: Count, not count.
- It may hold only letters and digits — no spaces, no punctuation.
- It is **not case-sensitive**, so Count and COUNT are the same name. Pick one spelling and keep it.
- Make it mean something. NumberOfPlayers earns the reader's time; N does not.

Common mistakes

- count ← 0 — a lower-case start. 0478 requires the capital.

- Number of players — spaces are not allowed inside a name.

1.3 Comments and indentation

A **comment** starts with // and runs to the end of the line. It is for the human reader; the algorithm ignores it.

```
// Swap the values of X and Y
DECLARE X, Y, Temp : INTEGER
X ← 1
Y ← 2
Temp ← X    // temporarily store X
X ← Y
Y ← Temp
OUTPUT X, " ", Y
```

Indentation shows which statements are **inside** a structure. The syllabus asks for four spaces per level, with one exception you must know:

```
DECLARE X, Y : INTEGER
X ← 7
Y ← 3
IF X > Y
    THEN
        OUTPUT "X is bigger"
    ELSE
        OUTPUT "Y is bigger or equal"
ENDIF
```

- THEN and ELSE are indented **two** spaces from their IF.
- The statements under them are indented **four**.
- ENDIF lines up with its own IF.

Common mistakes

- Putting THEN on the IF line. That is the A-Level 9618 layout — see topic 11.
- No indentation at all. A marker has to see the structure to award the structure marks.

2 Variables, constants and types

2.1 The five data types

0478 pseudocode has exactly **five** data types . Learn the list — naming a type that is not on it loses the mark.

Type	Holds	Example
INTEGER	a whole number	42, -7
REAL	a number with a decimal part	3.14, -0.5
CHAR	a single character	'A'
STRING	a sequence of characters	"Hello"
BOOLEAN	TRUE or FALSE	TRUE

```
DECLARE Age : INTEGER
DECLARE Price : REAL
DECLARE Grade : CHAR
DECLARE Name : STRING
DECLARE Passed : BOOLEAN
Age ← 17
Price ← 2.50
Grade ← 'A'
Name ← "Mei"
Passed ← TRUE
OUTPUT Name, " is ", Age, " and passed: ", Passed
```

- A STRING is written in **double** quotes, a CHAR in **single** quotes.
- TRUE and FALSE are keywords, so they are upper case and carry no quotes.

Common mistakes

- Writing FLOAT, DOUBLE or BOOL. Those are real languages, not this syllabus.
- 'Hello' for a string — single quotes hold one character only.

2.2 DECLARE

DECLARE names a variable and fixes its type before you use it.

```
DECLARE <identifier> : <data type>
```

```
DECLARE Counter : INTEGER
DECLARE TotalToPay : REAL
Counter ← 1
TotalToPay ← 19.99
OUTPUT "Item ", Counter, " costs ", TotalToPay
```

Several names of one type go on a single line, separated by commas:

```
DECLARE Length, Width, Area : INTEGER
Length ← 6
Width ← 4
Area ← Length * Width
OUTPUT "Area = ", Area
```

- Declare a variable **before** its first use, near the top of the algorithm.
- A variable keeps its type for the whole algorithm — an INTEGER never holds "Mei".

Common mistakes

- Leaving out the colon: DECLARE Counter INTEGER.
- Using a variable that was never declared. The exam expects the declaration.

2.3 CONSTANT

A **constant** is a value that is fixed for the whole algorithm — a tax rate, π , the size of a class. Naming it once means a change is made in one place.

```
CONSTANT <identifier> ← <value>
```

```
CONSTANT TaxRate ← 0.20
DECLARE Price, Tax : REAL
Price ← 50.00
Tax ← Price * TaxRate
OUTPUT "Tax to pay: ", Tax
```

- In **0478 the arrow is used**, exactly as for an assignment.
- The value must be a **literal** — a number, a string, TRUE — never a calculation and never another variable.
- A constant can never be given a new value later. That is the whole point of it.

Common mistakes

- CONSTANT TaxRate = 0.20. The = form is the A-Level 9618 spelling — see topic 11.
- Assigning to a constant further down the algorithm.

3 Assignment, input and output

3.1 The assignment arrow

The **assignment** operator is $\leftarrow$, a left-pointing arrow. It means *take the value on the right and put it in the box on the left*.

```
<identifier>  $\leftarrow$  <value>
```

```
DECLARE Counter : INTEGER  
Counter  $\leftarrow$  0  
Counter  $\leftarrow$  Counter + 1  
Counter  $\leftarrow$  Counter + 1  
OUTPUT Counter
```

`Counter  $\leftarrow$  Counter + 1` looks impossible as mathematics and is ordinary as an instruction: work out the right-hand side with the value the box holds now, then store the result back.

- If you cannot type $\leftarrow$, write `<-`. Both are accepted, here and in the exam.
- The left side must be a **single identifier** or one array element. Never a calculation.

Common mistakes

- `Counter = 0`. In pseudocode `=` is a **comparison**, not an assignment — and confusing the two is the single commonest syntax error in this paper.
- `Counter + 1  $\leftarrow$  Counter`. The arrow points at the thing being changed.

3.2 INPUT and OUTPUT

INPUT reads a value from the user into a variable. OUTPUT displays one or more values.

```
DECLARE Name : STRING  
DECLARE Age : INTEGER  
INPUT Name  
INPUT Age  
OUTPUT "Hello ", Name  
OUTPUT "Next year you will be ", Age + 1
```

- INPUT takes exactly **one** variable, and that variable must already be declared.
- OUTPUT takes any number of items separated by **commas**; they are printed one after another with nothing between them.
- A literal string in an OUTPUT goes in double quotes; a variable does not.

3.2.1 Prompting the user

An INPUT on its own shows nothing on screen, so a well-written algorithm prints a prompt first:

```
DECLARE Radius : REAL  
OUTPUT "Enter the radius: "  
INPUT Radius  
OUTPUT "The diameter is ", Radius * 2
```

Common mistakes

- `INPUT "Enter a number", X` — the prompt is a separate OUTPUT statement.
- `OUTPUT "Total: " + Total`. 0478 joins output items with a comma; `+` is arithmetic and `&` belongs to A-Level 9618 (topic 11).
- Forgetting the quotes: `OUTPUT Hello` looks for a *variable* called Hello.

4 Arithmetic and logic

4.1 Arithmetic operators

Five arithmetic operators, in the order a calculator uses them.

Operator	Meaning	A is 7, B is 2
+	add	A + B is 9
-	subtract	A - B is 5
*	multiply	A * B is 14
/	divide	A / B is 3.5
^	raise to a power	A ^ B is 49

```
DECLARE A, B : INTEGER
A ← 7
B ← 2
OUTPUT "Sum: ", A + B
OUTPUT "Product: ", A * B
OUTPUT "Quotient: ", A / B
OUTPUT "Power: ", A ^ B
```

Brackets come first, then ^, then * and /, then + and - — the usual precedence . Use brackets whenever the reader would have to stop and think.

```
DECLARE Total : REAL
Total ← (12 + 18) / 2
OUTPUT "Mean: ", Total
```

Common mistakes

- Expecting / to give a whole number. 7 / 2 is 3.5, not 3; for a whole number use DIV.
- Writing A x B or A ÷ B. Use * and /.

4.2 DIV and MOD

Integer division answers two different questions, and 0478 writes each of them as a **function** , with the two values in brackets.

- DIV(a, b) — how many whole times b goes into a, the **quotient** .
- MOD(a, b) — what is left over, the **remainder** .

```
OUTPUT DIV(10, 3)
OUTPUT MOD(10, 3)
OUTPUT DIV(17, 5)
OUTPUT MOD(17, 5)
```

Both take integers, and both return an integer.

4.2.1 What they are for

MOD is the standard test for *divides exactly*, which makes it the way to ask whether a number is even :

```
DECLARE Number : INTEGER
Number ← 12
IF MOD(Number, 2) = 0
    THEN
        OUTPUT Number, " is even"
    ELSE
        OUTPUT Number, " is odd"
ENDIF
```

DIV converts a total into whole units — seconds into minutes, pence into pounds:

```
DECLARE Seconds : INTEGER
Seconds ← 200
OUTPUT DIV(Seconds, 60), " minutes and ", MOD(Seconds, 60), " seconds"
```

Common mistakes

- 10 MOD 3. That is the A-Level 9618 form; **0478 uses the function form** MOD(10, 3).
- Mixing the two up. DIV gives the whole part; MOD gives what does not fit.

4.3 Relational and logic operators

A **relational operator** compares two values and produces TRUE or FALSE.

Operator	Means
=	is equal to
<>	is not equal to
>	is greater than
<	is less than
>=	is greater than or equal to
<=	is less than or equal to

```

DECLARE Mark : INTEGER
Mark ← 55
OUTPUT "Pass: ", Mark >= 50
OUTPUT "Perfect: ", Mark = 100
OUTPUT "Not zero: ", Mark <> 0

```

The three **logic operators** join conditions:

- AND — both sides must be TRUE.
- OR — at least one side must be TRUE.
- NOT — turns TRUE into FALSE and back.

```

DECLARE Age : INTEGER
DECLARE HasTicket : BOOLEAN
Age ← 16
HasTicket ← TRUE
IF Age >= 15 AND HasTicket = TRUE
    THEN
        OUTPUT "Admitted"
ENDIF
IF NOT (Age > 18)
    THEN
        OUTPUT "Still a student rate"
ENDIF

```

Common mistakes

- <> is the *not equal* sign here — not !=, which belongs to other languages.
- IF Age >= 15 AND <= 18. Each side of AND needs its own complete comparison: Age >= 15 AND Age <= 18.
- Calling these *comparison operators* in an answer. The syllabus term is **relational**.

5 String operations

5.1 LENGTH, UCASE and LCASE

0478 publishes four string routines and no others. Three of them are here.

LENGTH(<identifier>) returns how many characters a string holds — spaces included.

```

OUTPUT LENGTH("Happy Days")
OUTPUT LENGTH("")

```

UCASE(<identifier>) returns the string in upper case , LCASE(<identifier>) in lower case . Neither changes the original — each hands back a new value.

```

DECLARE Name : STRING
Name ← "Happy"
OUTPUT UCASE(Name)
OUTPUT LCASE(Name)
OUTPUT Name

```

That last line still prints Happy: to keep the change you must assign it back.

```

DECLARE Name : STRING
Name ← "Happy"
Name ← UCASE(Name)
OUTPUT Name

```

5.1.1 Comparing without worrying about case

UCASE on both sides is the standard way to accept an answer however it was typed:

```

DECLARE Answer : STRING
Answer ← "yes"
IF UCASE(Answer) = "YES"
    THEN
        OUTPUT "Accepted"
ENDIF

```

Common mistakes

- Expecting UCASE(Name) to change Name. Assign the result back.
- LENGTH counts characters, not words. LENGTH("Happy Days") is 10.

5.2 SUBSTRING

SUBSTRING(<identifier>, <start>, <length>) pulls a piece out of a string — starting at position **start**, taking **length** characters.

```
OUTPUT SUBSTRING("Happy Days", 1, 5)
OUTPUT SUBSTRING("Happy Days", 7, 4)
```

△ **Counting starts at 1**, not 0. The first character of "Happy Days" is at position 1.

5.2.1 Taking one character, and the rest

```
DECLARE Word : STRING
Word ← "Computer"
OUTPUT "First letter: ", SUBSTRING(Word, 1, 1)
OUTPUT "Last letter: ", SUBSTRING(Word, LENGTH(Word), 1)
OUTPUT "Without the first: ", SUBSTRING(Word, 2, LENGTH(Word) - 1)
```

LENGTH inside SUBSTRING is what makes the last two lines work for a word of any size — hard-coding 8 would break the moment the word changed.

5.2.2 Reading a string one character at a time

```
DECLARE Word : STRING
DECLARE Index : INTEGER
Word ← "Cat"
FOR Index ← 1 TO LENGTH(Word)
    OUTPUT SUBSTRING(Word, Index, 1)
NEXT Index
```

Common mistakes

- Starting at 0. Position 1 is the first character.
- Reading the third argument as an *end position*. It is a **count** — SUBSTRING(W, 2, 3) takes three characters beginning at the second.
- Asking for more characters than the string has.

6 Selection

6.1 IF, THEN, ELSE, ENDIF

Selection runs one group of statements or another, depending on a condition .

```
IF <condition>
    THEN
        <statements>
ENDIF
```

```
DECLARE Mark : INTEGER
Mark ← 72
IF Mark >= 50
    THEN
        OUTPUT "Pass"
ENDIF
```

Add ELSE for the other case. Exactly one of the two branches runs — never both, never neither.

```
DECLARE Mark : INTEGER
Mark ← 41
IF Mark >= 50
    THEN
        OUTPUT "Pass"
    ELSE
        OUTPUT "Fail"
ENDIF
```

The layout is the one part students lose marks on, so learn it as a shape:

- IF and its condition on one line, **nothing after the condition**.
- THEN and ELSE on their own lines, indented **two** spaces.
- The statements indented **four**.
- ENDIF back at the column of its IF.

6.1.1 Nesting

An IF inside an IF handles more than two outcomes. Each one needs its own ENDIF.

```
DECLARE Mark : INTEGER
Mark ← 85
IF Mark >= 80
  THEN
    OUTPUT "Distinction"
  ELSE
    IF Mark >= 50
      THEN
        OUTPUT "Pass"
      ELSE
        OUTPUT "Fail"
      ENDIF
    ENDIF
  ENDIF
```

Common mistakes

- Forgetting ENDIF. Every IF closes, and a missing one is a syntax error the marker will see at a glance.
- IF Mark >= 50 THEN OUTPUT "Pass" on one line. **No such form exists** in either syllabus; write the three lines.
- IF Mark = 50 THEN written as IF Mark ← 50. The arrow assigns; = compares.

6.2 CASE OF

When one variable is tested against several **single values**, CASE OF says it far more clearly than a stack of nested IFs.

```
CASE OF <identifier>
  <value 1> : <statement>
  <value 2> : <statement>
  OTHERWISE <statement>
ENDCASE
```

```
DECLARE Choice : INTEGER
Choice ← 2
CASE OF Choice
  1 : OUTPUT "You chose north"
  2 : OUTPUT "You chose south"
  3 : OUTPUT "You chose east"
  4 : OUTPUT "You chose west"
  OTHERWISE OUTPUT "That is not a direction"
ENDCASE
```

- Each branch is a **value**, a colon, then what to do.
- OTHERWISE catches everything the listed values missed. It is optional, and writing one is good practice — it is what happens when the user types something unexpected.
- The first matching branch runs, and the rest are skipped.

6.2.1 A range, and a character

```
DECLARE Grade : CHAR
Grade ← 'B'
CASE OF Grade
  'A' : OUTPUT "Excellent"
  'B' : OUTPUT "Good"
  'C' : OUTPUT "Satisfactory"
  OTHERWISE OUTPUT "Unclassified"
ENDCASE
```

Common mistakes

- Using CASE OF for a **condition**. It tests one variable against values; Mark >= 50 needs an IF.
- Forgetting ENDCASE.
- Leaving out OTHERWISE and then wondering why an unexpected value does nothing at all.

7 Iteration

7.1 FOR, the counted loop

Iteration repeats statements. Use a FOR loop when you know **before you start** how many times — a count-controlled loop.

```
FOR <identifier> ← <value1> TO <value2>
    <statements>
NEXT <identifier>
```

```
DECLARE Index : INTEGER
FOR Index ← 1 TO 5
    OUTPUT "Line ", Index
NEXT Index
```

The counter starts at the first value, and the loop runs once for every value up to and **including** the second. 1 TO 5 runs five times.

7.1.1 STEP

STEP changes the size of each jump. A negative step counts down.

```
DECLARE Index : INTEGER
FOR Index ← 2 TO 10 STEP 2
    OUTPUT Index
NEXT Index
FOR Index ← 3 TO 1 STEP -1
    OUTPUT "Countdown ", Index
NEXT Index
```

7.1.2 Running totals

The commonest use: a variable outside the loop that the loop adds to.

```
DECLARE Index, Total : INTEGER
Total ← 0
FOR Index ← 1 TO 10
    Total ← Total + Index
NEXT Index
OUTPUT "Sum of 1 to 10 is ", Total
```

Total ← 0 must sit **before** the loop. Inside, it would reset on every pass.

Common mistakes

- NEXT naming the wrong counter. Nested loops must close in the reverse order they opened: the inner NEXT first.
- Assuming 1 TO 5 runs four times. Both ends are included.
- Changing the counter inside the loop. Let FOR own it.

7.2 WHILE, the pre-condition loop

A WHILE loop tests **before** each pass, so it can run **zero** times. Use it when the number of repeats depends on something that happens while the loop runs.

```
WHILE <condition> DO
    <statements>
ENDWHILE
```

```
DECLARE Total : INTEGER
Total ← 1
WHILE Total < 100 DO
    Total ← Total * 2
ENDWHILE
OUTPUT "First power of two past 100: ", Total
```

Something inside the loop must eventually make the condition false, or the loop never ends — an **infinite loop** .

```
DECLARE Count : INTEGER
Count ← 5
WHILE Count > 0 DO
    OUTPUT Count
    Count ← Count - 1
ENDWHILE
OUTPUT "Lift off"
```

Common mistakes

- Forgetting to change the variable the condition tests, giving an infinite loop.
- Leaving out DO or ENDWHILE.
- Using WHILE where the count is known. A FOR says what you mean in one line.

7.3 REPEAT, the post-condition loop

A REPEAT loop tests **after** each pass, so it always runs **at least once**.

```
REPEAT
    <statements>
UNTIL <condition>
```

That “at least once” is the reason to choose it: validating input, where you must ask before you can judge the answer.

```
DECLARE Number : INTEGER
Number ← 0
REPEAT
    Number ← Number + 25
    OUTPUT "Trying ", Number
UNTIL Number >= 100
OUTPUT "Reached ", Number
```

△ UNTIL states the condition that **stops** the loop; WHILE states the one that **keeps it going**. They are opposites, and swapping them is the classic error.

	Tests	Runs at least once?	Condition means
FOR	a counter	no (an empty range runs zero times)	—
WHILE	before	no	keep going while this is true
REPEAT	after	yes	stop when this becomes true

Common mistakes

- Writing UNTIL Number < 100 when you mean UNTIL Number >= 100.
- Using REPEAT where the body must be able to run zero times.
- Forgetting that UNTIL ends the structure — there is no ENDREPEAT.

8 Arrays

8.1 One-dimensional arrays

An **array** holds many values of one type under one name. Each value sits at an **index** .

```
DECLARE <identifier> : ARRAY[<lower>:<upper>] OF <data type>
```

```
DECLARE Names : ARRAY[1:3] OF STRING
Names[1] ← "Mei"
Names[2] ← "Sam"
Names[3] ← "Ana"
OUTPUT Names[2]
```

- The bounds are written lower:upper, and **both are included** — [1:3] has three elements .
- Cambridge arrays normally start at 1, not 0.
- Every element has the same type, fixed by OF.

8.1.1 Filling and reading with a FOR loop

An array and a counted loop belong together: the counter is the index.

```
DECLARE Scores : ARRAY[1:5] OF INTEGER
DECLARE Index, Total : INTEGER
FOR Index ← 1 TO 5
    Scores[Index] ← Index * 10
NEXT Index
Total ← 0
FOR Index ← 1 TO 5
    Total ← Total + Scores[Index]
NEXT Index
OUTPUT "Total: ", Total
OUTPUT "Mean: ", Total / 5
```

8.1.2 Finding the largest

```
DECLARE Scores : ARRAY[1:5] OF INTEGER
DECLARE Index, Largest : INTEGER
Scores[1] ← 42
Scores[2] ← 17
Scores[3] ← 93
Scores[4] ← 8
Scores[5] ← 55
Largest ← Scores[1]
FOR Index ← 2 TO 5
    IF Scores[Index] > Largest
    THEN
        Largest ← Scores[Index]
    ENDIF
NEXT Index
OUTPUT "Largest: ", Largest
```

Start **Largest** at the **first element**, never at 0 — with negative data, 0 would win and the answer would be wrong.

Common mistakes

- Reading `Scores[6]` from an array declared `[1:5]`. That is out of bounds .
- Writing `Scores(3)`. Arrays use square brackets; round brackets call a function.
- Declaring `[1:5]` and then looping `0 TO 4`.

8.2 Two-dimensional arrays

A **2-D array** is a table: rows and columns, two indexes.

```
DECLARE <identifier> : ARRAY[<lower1>:<upper1>, <lower2>:<upper2>] OF
↪ <data type>
```

```
DECLARE Grid : ARRAY[1:2, 1:3] OF INTEGER
Grid[1,1] ← 1
Grid[1,2] ← 2
Grid[1,3] ← 3
Grid[2,1] ← 4
Grid[2,2] ← 5
Grid[2,3] ← 6
OUTPUT Grid[2,3]
```

The **first** index is the row , the **second** the column .

8.2.1 Nested loops walk a table

One loop per dimension . The inner loop finishes a whole row before the outer moves on.

```
DECLARE Grid : ARRAY[1:2, 1:3] OF INTEGER
DECLARE Row, Col : INTEGER
FOR Row ← 1 TO 2
    FOR Col ← 1 TO 3
        Grid[Row, Col] ← Row * Col
    NEXT Col
NEXT Row
FOR Row ← 1 TO 2
    FOR Col ← 1 TO 3
        OUTPUT "Grid[" , Row, ", ", Col, "]" = " , Grid[Row, Col]
    NEXT Col
NEXT Row
```

△ The inner NEXT closes the **inner** counter. NEXT Row before NEXT Col crosses the loops over and is nonsense — a marker spots it instantly.

Common mistakes

- Swapping row and column. `Grid[2,3]` is row 2, column 3.
- `Grid[Row] [Col]`. Cambridge writes **one** pair of brackets with a comma inside.
- Crossing the NEXT lines over, as above.

9 Procedures and functions

9.1 PROCEDURE and CALL

A **procedure** is a named block of statements. Writing one once and calling it from several places is **decomposition** — the idea the whole of paper 2 rests on.

```
PROCEDURE <identifier>
    <statements>
ENDPROCEDURE
```

A procedure is **called** by name, with the keyword **CALL**:

```
PROCEDURE DefaultLine
    OUTPUT "-----"
ENDPROCEDURE

CALL DefaultLine
OUTPUT "Report"
CALL DefaultLine
```

- The definition does nothing on its own. Only **CALL** runs it.
- Control returns to the line after the **CALL** when the procedure ends.
- A procedure **does not give a value back**. If you need one, you want a function.

Common mistakes

- Writing **DefaultLine** on its own to run it. 0478 requires **CALL**.
- Leaving out **ENDPROCEDURE**.
- Expecting the procedure's own variables to exist outside it.

9.2 FUNCTION and RETURN

A **function** is a procedure that hands a value back. It is written where that value is needed — inside an **OUTPUT**, on the right of an assignment, inside a condition.

```
FUNCTION <identifier>(<parameters>) RETURNS <data type>
    <statements>
    RETURN <value>
ENDFUNCTION
```

```
FUNCTION SumSquare(Number1 : INTEGER, Number2 : INTEGER) RETURNS INTEGER
    RETURN Number1 * Number1 + Number2 * Number2
ENDFUNCTION
```

```
OUTPUT "Sum of squares = ", SumSquare(10, 20)
```

- **RETURNS** <data type> on the header says what type comes back. It is not optional.
- **RETURN** ends the function **immediately** — nothing after it runs.
- A function is **never** called with **CALL**; it is used wherever its value belongs.

9.2.1 A function inside a condition

```
FUNCTION IsEven(Number : INTEGER) RETURNS BOOLEAN
    RETURN MOD(Number, 2) = 0
ENDFUNCTION

DECLARE Index : INTEGER
FOR Index ← 1 TO 6
    IF IsEven(Index)
        THEN
            OUTPUT Index, " is even"
        ENDIF
    NEXT Index
```

Common mistakes

- **CALL** **SumSquare**(10, 20) — that throws the answer away.
- A function with no **RETURN** on some path. Every route out must return a value.
- Leaving **RETURNS INTEGER** off the header.

9.3 Parameters

A **parameter** is a value a procedure or function is given when it is called. Each one carries its own type.

```
PROCEDURE Line(Size : INTEGER)
    DECLARE Index : INTEGER
    FOR Index ← 1 TO Size
        OUTPUT "-"
    NEXT Index
ENDPROCEDURE
```

```
CALL Line(5)
CALL Line(20)
```

Several parameters are separated by commas, and the order at the call must match the order in the definition:

```
PROCEDURE Greet(Name : STRING, Times : INTEGER)
  DECLARE Index : INTEGER
  FOR Index ← 1 TO Times
    OUTPUT "Hello ", Name
  NEXT Index
ENDPROCEDURE

CALL Greet("Mei", 2)
CALL Greet("Sam", 1)
```

△ In **0478** every parameter is passed by value : the procedure works on a copy, so changing it inside changes nothing outside. To get a value back, use a **FUNCTION** and **RETURN** it. (A-Level 9618 adds **BYREF** for the other behaviour — topic 11.)

```
FUNCTION Doubled(Number : INTEGER) RETURNS INTEGER
  RETURN Number * 2
ENDFUNCTION

DECLARE Value : INTEGER
Value ← 7
Value ← Doubled(Value)
OUTPUT Value
```

Common mistakes

- **PROCEDURE** `Line(Size)` with no type.
- Calling with the arguments in the wrong order — `Greet(2, "Mei")` is a type error.
- Expecting a procedure to change the variable you passed in. In 0478 it cannot.

10 File handling

10.1 Writing to a file

A file keeps data after the program ends. 0478 uses four commands, and every one names the file it works on.

```
OPENFILE <file identifier> FOR WRITE
WRITEFILE <file identifier>, <data>
CLOSEFILE <file identifier>
```

```
OPENFILE "names.txt" FOR WRITE
WRITEFILE "names.txt", "Mei"
WRITEFILE "names.txt", "Sam"
CLOSEFILE "names.txt"
OUTPUT "Saved"
```

Three **file modes** :

Mode	What it does
WRITE	start a new file — anything already there is lost
APPEND	add to the end of what is there
READ	read from the start

```
OPENFILE "log.txt" FOR WRITE
WRITEFILE "log.txt", "first"
CLOSEFILE "log.txt"
OPENFILE "log.txt" FOR APPEND
WRITEFILE "log.txt", "second"
CLOSEFILE "log.txt"
OUTPUT "Both lines saved"
```

- One **WRITEFILE** writes **one line**.
- **CLOSEFILE** is not optional. An unclosed file can lose the data still waiting to be written, and that is a stock exam answer.

Common mistakes

- **FOR WRITE** when you meant **FOR APPEND**, which silently destroys the old contents.
- Forgetting **CLOSEFILE**.
- Opening a file that is already open.

10.2 Reading a file to the end

READFILE takes one line into a variable. The variable must be declared first.

```
OPENFILE <file identifier> FOR READ
READFILE <file identifier>, <variable>
CLOSEFILE <file identifier>
```

Normally you do not know how many lines there are, so you read **until the end of the file** — which is what the EOF function reports.

```
OPENFILE "names.txt" FOR WRITE
WRITEFILE "names.txt", "Mei"
WRITEFILE "names.txt", "Sam"
WRITEFILE "names.txt", "Ana"
CLOSEFILE "names.txt"
```

```
DECLARE Line : STRING
OPENFILE "names.txt" FOR READ
WHILE NOT EOF("names.txt") DO
    READFILE "names.txt", Line
    OUTPUT Line
ENDWHILE
CLOSEFILE "names.txt"
```

EOF("names.txt") is TRUE once every line has been read, so WHILE NOT EOF(...) means *while there is still something to read*.

10.2.1 Counting while reading

```
OPENFILE "marks.txt" FOR WRITE
WRITEFILE "marks.txt", 40
WRITEFILE "marks.txt", 65
WRITEFILE "marks.txt", 88
CLOSEFILE "marks.txt"
```

```
DECLARE Mark, Count, Total : INTEGER
Count ← 0
Total ← 0
OPENFILE "marks.txt" FOR READ
WHILE NOT EOF("marks.txt") DO
    READFILE "marks.txt", Mark
    Count ← Count + 1
    Total ← Total + Mark
ENDWHILE
CLOSEFILE "marks.txt"
OUTPUT Count, " marks, mean ", Total / Count
```

Common mistakes

- A WHILE EOF(...) loop — that runs only when there is nothing left. You want NOT.
- Reading past the end because the loop has no EOF guard.
- Opening FOR READ a file that was never written.

11 A-Level 9618: what changes

11.1 Layout and operators

Cambridge publishes **two** pseudocode specifications, and they disagree. Everything in topics 1-10 is **IGCSE 0478**. A-Level **9618** changes the details below — and a student who studies both loses marks by writing one in the other’s paper.

	IGCSE 0478	A-Level 9618
THEN	on its own line , indented 2	on the IF line
constants	CONSTANT Pi ← 3.142	CONSTANT Pi = 3.142
DIV / MOD	functions: DIV(10, 3)	operators: 10 DIV 3
joining strings	commas in OUTPUT	the & operator
identifiers	must start with a capital	mixed case allowed
NEXT	must name its counter	good practice
ROUND, RANDOM()	both published	neither exists

This is the 0478 form, and it runs here:

```
DECLARE Total : REAL
CONSTANT Rate ← 0.5
Total ← 10
IF MOD(7, 2) = 1
    THEN
        OUTPUT "Odd, half rate ", Total * Rate
    ENDIF
```

The 9618 form of the same algorithm is written like this — note THEN on the IF line, = in the CONSTANT, and MOD between its two values:

```
DECLARE Total : REAL
CONSTANT Rate = 0.5
Total ← 10
IF 7 MOD 2 = 1 THEN
    OUTPUT "Odd, half rate " & STR(Total * Rate)
ENDIF
```

△ Which one is right depends only on which paper you are sitting. Neither is a mistake; each is a mistake in the other’s exam. The playground on this page runs the **0478** rules, so a 9618 construct is refused with a message naming the syllabus it belongs to — which is a quick way to check your own habits.

Common mistakes

- Writing THEN on the IF line in an 0478 answer, because that is what a textbook or

- a website showed. It is the single commonest cross-syllabus slip.
- 10 MOD 3 in 0478. There it is MOD(10, 3).
- ROUND(3.14159, 2) in a 9618 answer. 9618 has no ROUND at all.

11.2 Parameters: BYVAL and BYREF

0478 passes **every** parameter by value (topic 9). 9618 lets you choose, and says so in the header.

- BYVAL — the procedure gets a **copy**. Changing it changes nothing outside. This is the default when neither word is written.
- BYREF — the procedure works on the caller’s **own variable** , so a change is visible after the call.

This is 9618 only. It does not run here, and it is not accepted in an 0478 answer:

```
PROCEDURE Increase(BYREF Value : INTEGER)
    Value ← Value + 1
ENDPROCEDURE

DECLARE Count : INTEGER
Count ← 5
CALL Increase(Count)
OUTPUT Count          // prints 6
```

11.2.1 The 0478 way to get a value back

Use a FUNCTION and assign what it returns. This is the whole of the 0478 answer, and it runs:

```
FUNCTION Increased(Value : INTEGER) RETURNS INTEGER
    RETURN Value + 1
ENDFUNCTION

DECLARE Count : INTEGER
Count ← 5
Count ← Increased(Count)
OUTPUT Count
```

Common mistakes

- Writing BYREF in an 0478 answer. The keyword is not in that syllabus.
- Expecting a plain 9618 parameter to behave like BYREF. Without the word it is BYVAL.

11.3 Records and other types

9618 has data structures that 0478 does not. You will meet them only at A-Level, and none of them runs under the 0478 rules of this page.

A **record** groups fields of different types under one name (9618 section 4):

```
TYPE Student
  DECLARE Name : STRING
  DECLARE Mark : INTEGER
ENDTYPE

DECLARE Candidate : Student
Candidate.Name ← "Mei"
Candidate.Mark ← 91
OUTPUT Candidate.Name, " scored ", Candidate.Mark
```

An **enumerated type** lists the only values a variable may take:

```
TYPE Vehicle = (Car, Bus, Taxi)
DECLARE MyRide : Vehicle
MyRide ← Taxi
```

9618 also adds a DATE data type, random-access files (OPENFILE ... FOR RANDOM), pointers, and classes with CLASS ... ENDCLASS for its object-oriented paper.

△ 0478 has **five** data types and arrays, and nothing else (topic 2). A 0478 answer that declares a record is answering a different syllabus.

Common mistakes

- Using a record in an 0478 answer. Use parallel arrays instead — one array per field, sharing an index.
- Thinking TYPE means the same as DECLARE. TYPE defines a **new type**; DECLARE creates a **variable** of an existing one.