

Programming

IGCSE Computer Science 0478 · Topic 8 revision notes

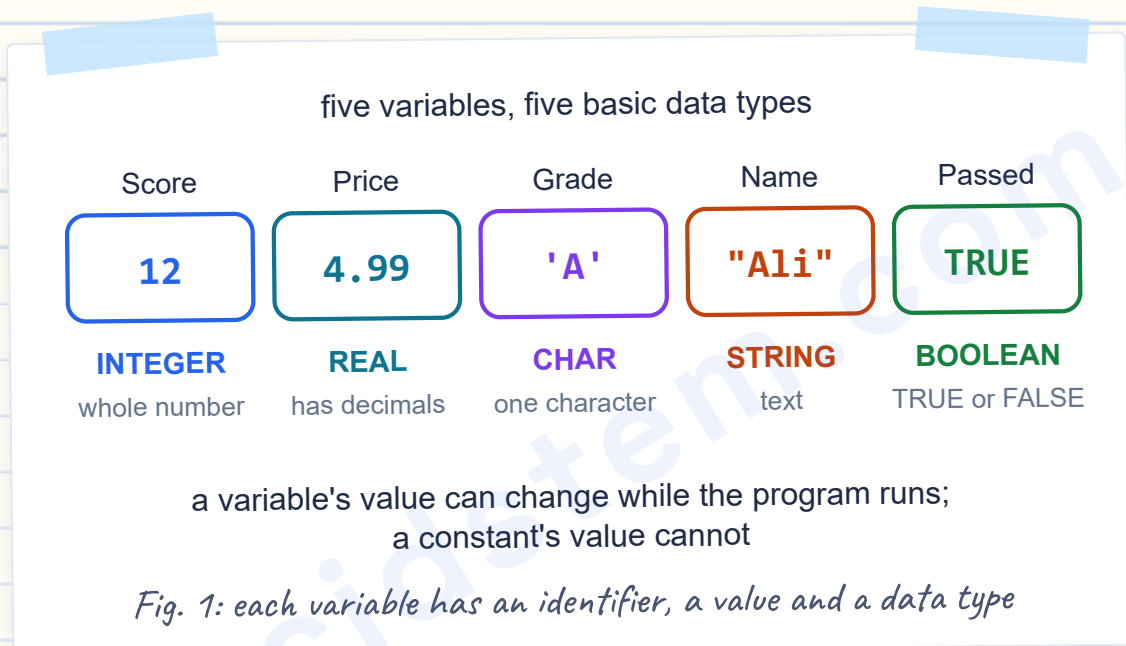
Everything from the 11 lessons of this topic on a few pages: the key idea of each lesson, the diagram to remember, the points to learn and the traps examiners set. Read it, cover it, test yourself.

1. Variables, constants and data types
2. Sequence and selection: IF and CASE
3. Iteration: count-controlled, pre-condition and post-condition loops
4. String handling
5. Arithmetic, relational and logical operators
6. Procedures, functions, parameters and scope
7. Library routines: MOD, DIV, ROUND and RANDOM
8. Writing maintainable programs
9. One-dimensional arrays
10. Two-dimensional arrays and nested loops
11. Reading from and writing to files

1 Variables, constants and data types

Syllabus 8.1

A **variable** is a named memory location that stores a value which **can change** while the program runs. A **constant** is a named value that **cannot change** while the program runs. In pseudocode you declare them with `DECLARE Score : INTEGER` and `CONSTANT VatRate ← 0.2`, and assign with the arrow `←`. The five basic **data types** are **INTEGER**, **REAL**, **CHAR**, **STRING** and **BOOLEAN**. Statements run one after another in **sequence**, so their order matters.



Worked example: Choosing data types

A club stores, for each member: their surname, their age in years, their height in metres, the first letter of their team name, whether they have paid, and their phone number. Choose a data type for each.

1. Surname is text: **STRING**.
2. Age in whole years: **INTEGER**.
3. Height such as 1.62 needs a fractional part: **REAL**.
4. One letter: **CHAR**. Paid or not: **BOOLEAN**.
5. A phone number such as 01632 960123 starts with 0 and is never used in arithmetic: **STRING**.

Answer: **STRING, INTEGER, REAL, CHAR, BOOLEAN, STRING**

Name, type, value.

- ✓ **Variable:** named memory location; its value can change while the program runs.
- ✓ **Constant:** named value that cannot change while the program runs; declared with `CONSTANT` and a literal.
- ✓ **Data types:** `INTEGER`, `REAL`, `CHAR`, `STRING`, `BOOLEAN`. Phone numbers are `STRING`.
- ✓ **Sequence:** work through statements in order, updating each variable as you go.

Watch out!

"A constant never changes" is not enough on its own. Say that its value cannot be changed while the program is running, and give a benefit: it only has to be updated in one place.

A phone number is a `STRING`, not an `INTEGER`: storing it as a number would lose the leading zero, and nobody adds or multiplies phone numbers.

`OUTPUT "Total"` displays the word `Total`; `OUTPUT Total` displays the value stored in the variable `Total`. Check your quotation marks.

2 Sequence and selection: IF and CASE

Syllabus 8.1

Sequence means statements run in order, one after another. **Selection** means the program chooses which statements to run by testing a **condition** that is either TRUE or FALSE. An **IF** statement runs the THEN branch when the condition is TRUE and the optional ELSE branch when it is FALSE. A **nested IF** is an IF inside another IF. A **CASE** statement compares one variable with several single values and runs only the first match; **OTHERWISE** catches every other value.

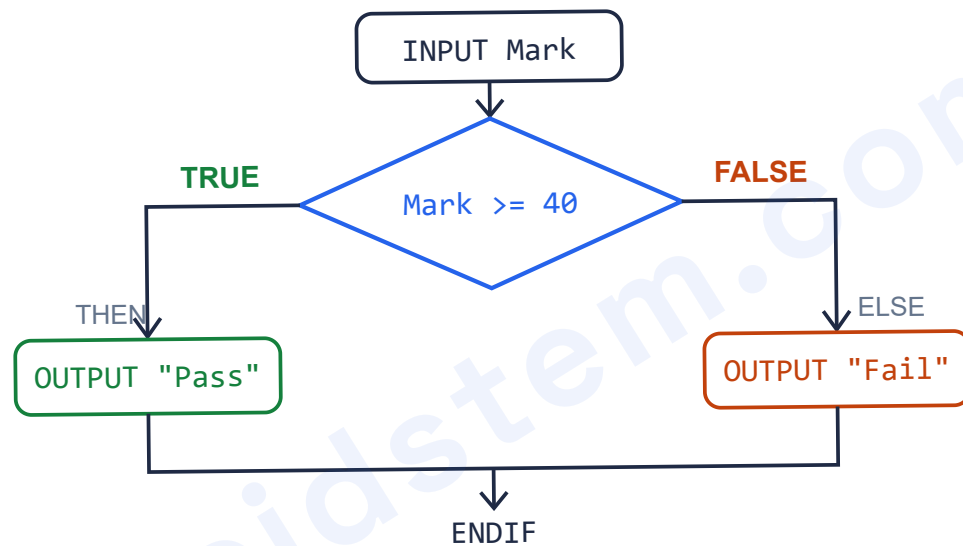


Fig. 1: an IF statement follows one of two paths, then both paths meet again at ENDIF

Worked example: Dry run

What does the program above output when Mark is 70, and when Mark is 39?

1. Mark = 70: is 70 >= 70? TRUE, so output Distinction. The ELSE branch (and the inner IF) is skipped.
2. Mark = 39: is 39 >= 70? FALSE, so go to ELSE.
3. Inner IF: is 39 >= 40? FALSE, so the inner ELSE runs.

Answer: 70 gives Distinction; 39 gives Fail

Condition, then branch.

- ✓ **IF:** IF condition, THEN, optional ELSE, ENDIF.
- ✓ **Boundaries:** decide whether the boundary value passes: > or >=.
- ✓ **Nested IF:** test the highest band first; close every IF with its own ENDIF.
- ✓ **CASE:** one variable, single values, first match only, OTHERWISE last.

Watch out!

Check the boundary. With `IF Age > 18` an 18-year-old would be told they are too young. Use `>=` when the boundary value itself should pass.

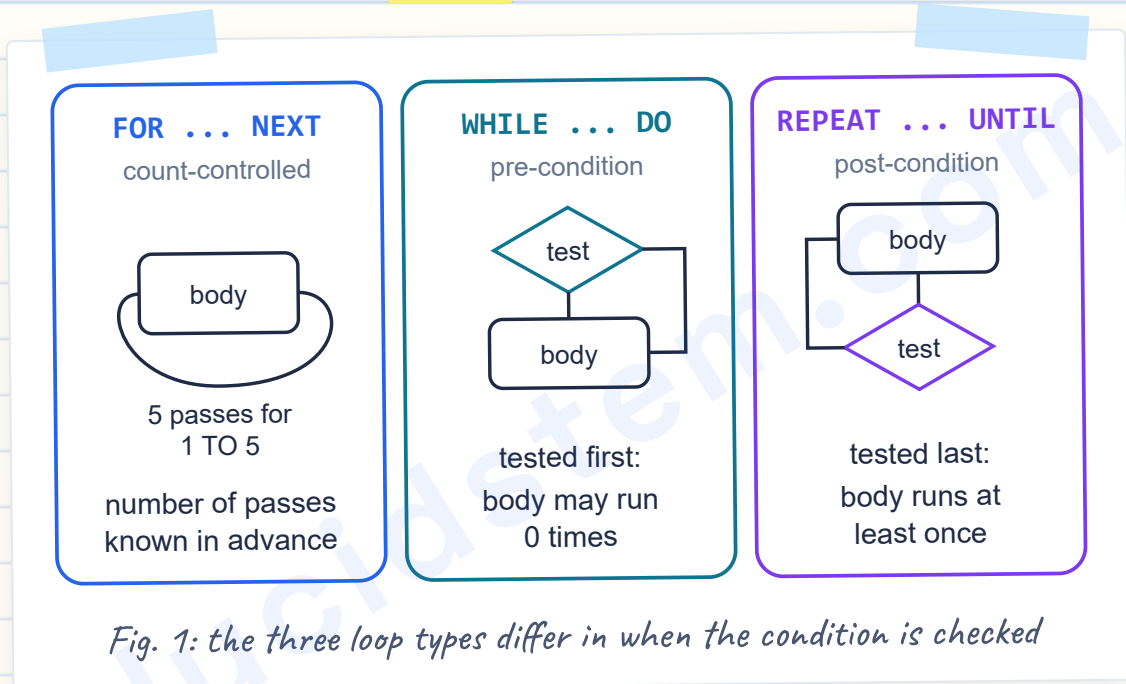
Test the highest band first. If the program tested `Mark ≥ 40` first, a mark of 85 would output Pass and never reach Distinction.

OTHERWISE must be the last case, and it runs only if none of the cases above it match. Use an IF, not CASE, when a branch depends on a range such as `Mark >= 40`.

3 Iteration: count-controlled, pre-condition and post-condition loops

Syllabus 8.1

Iteration means repeating a group of statements. A **count-controlled** loop (FOR ... NEXT) runs a set number of times. A **pre-condition** loop (WHILE ... DO ... ENDWHILE) tests its condition **before** each pass, so it may run zero times. A **post-condition** loop (REPEAT ... UNTIL) tests its condition **after** each pass, so it always runs at least once. **Totalling** ($Total \leftarrow Total + Number$) and **counting** ($Count \leftarrow Count + 1$) are done inside loops.



Worked example: Counting passes

How many times does the body of `FOR X ← 3 TO 12 STEP 4` run, and with which values of X?

1. X starts at 3: $3 \leq 12$, run the body.
2. $X \leftarrow 7$: run. $X \leftarrow 11$: run.
3. $X \leftarrow 15$ goes past 12, so the loop ends.

Answer: 3 times, with $X = 3, 7$ and 11

Know when it stops.

- ✓ **FOR:** known number of passes; the end value is included.
- ✓ **WHILE:** tested first, may run 0 times, continues while TRUE.
- ✓ **REPEAT:** tested last, runs at least once, stops when the UNTIL condition is TRUE.
- ✓ **Totals and counts:** set to 0 before the loop, update inside it, output after it.

Watch out!

A FOR loop in pseudocode includes the end value: `FOR Count ← 1 TO 5` runs 5 times. Python's `range(1, 5)` would stop at 4, so do not mix the two up.

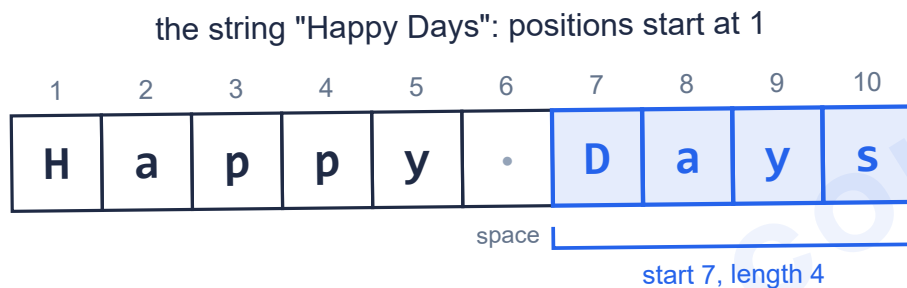
The REPEAT condition says when to stop; the WHILE condition says when to keep going. Turning one into the other means reversing the condition.

Put `Total ← 0` before the loop. If it is inside the loop, the total is reset on every pass and only the last mark is kept.

4 String handling

Syllabus 8.1

A **string** is a sequence of characters. **LENGTH(s)** returns the number of characters, spaces included. **SUBSTRING(s, start, length)** returns *length* characters starting at position *start*. **UCASE** and **LCASE** return the string in upper or lower case. In 0478 pseudocode the first character is generally at **position 1**; many programming languages, including Python, start at 0.



`SUBSTRING("Happy Days", 7, 4) = "Days"`

`LENGTH("Happy Days") = 10`

Fig. 1: **SUBSTRING** takes a start position and a number of characters

Worked example: Counting positions

What is output by the pseudocode above?

1. Number the characters: C1 O2 M3 P4 U5 T6 E7 R8.
2. `SUBSTRING(Word, 4, 3)` starts at position 4 and takes 3 characters: P, U, T.
3. `LENGTH(Word)` is 8, so `SUBSTRING(Word, 8, 1)` is the last character.

Answer: PUT, then R

Number the characters.

- ✓ **Write positions** under each character, starting at 1. Spaces count.
- ✓ **SUBSTRING(s, start, length)**: the third value is how many characters, not where to stop.
- ✓ **Case**: convert with UCASE or LCASE before comparing.
- ✓ **Every character**: FOR Index $\leftarrow$ 1 TO LENGTH(s) with SUBSTRING(s, Index, 1).

Watch out!

Spaces and punctuation are characters too. LENGTH("Hi there!") is 9, not 7.

The third value is a *length*, not an end position. Characters 3 to 6 are SUBSTRING(Word, 3, 4), not SUBSTRING(Word, 3, 6).

Without LCASE the capital E at the start of Eleven would not equal 'e', and the program would output 2.

5 Arithmetic, relational and logical operators

Syllabus 8.1

Arithmetic operators: $+$ $-$ $*$ $/$ $^$ (raised to the power of), plus MOD and DIV. **Relational** operators: $=$ $<$ $<=$ $>$ $>=$ and $<>$ (not equal to); they compare two values and always give a **BOOLEAN** result. **Logical** operators: **AND** (TRUE only if both sides are TRUE), **OR** (TRUE if either side is TRUE) and **NOT** (reverses TRUE and FALSE). Powers are worked out first, then $*$ and $/$, then $+$ and $-$; brackets make the order explicit.

Arithmetic

calculate

$+$ $-$ $*$ $/$
 $^$ (power)
MOD (remainder)
DIV (whole quotient)

Relational

compare

$=$ equal to
 $<$ less than
 $<=$ less or equal
 $>$ greater than
 $>=$ greater or equal
 $<>$ not equal to

Logical

combine conditions

AND both
OR either
NOT reverses

relational and logical operators always give a BOOLEAN: TRUE or FALSE

Fig. 1: the three groups of operators in 0478 pseudocode

Worked example: Area of a circle

Evaluate $\text{Area} \leftarrow 3.142 * 5 ^ 2$.

1. The power is worked out first: $5 ^ 2 = 25$.
2. Then the multiplication: $3.142 * 25$.

Answer: 78.55

Brackets first.

- ✓ **Order:** brackets, powers (^), then * and /, then + and -.
- ✓ **/ versus DIV:** $7 / 2 = 3.5$; $DIV(7, 2) = 3$; $MOD(7, 2) = 1$.
- ✓ **Relational:** = < <= > >= <>, always TRUE or FALSE.
- ✓ **Logical:** AND needs both, OR needs either, NOT reverses; each side a full comparison.

Watch out!

/ gives the full answer, including any fractional part: $7 / 2$ is 3.5. Use DIV only when you want the whole-number quotient, 3.

In a condition, = compares; it does not store anything. Assignment always uses the arrow ←. (Python uses == to compare and != for not equal.)

Each side of AND or OR must be a complete comparison. `Age >`

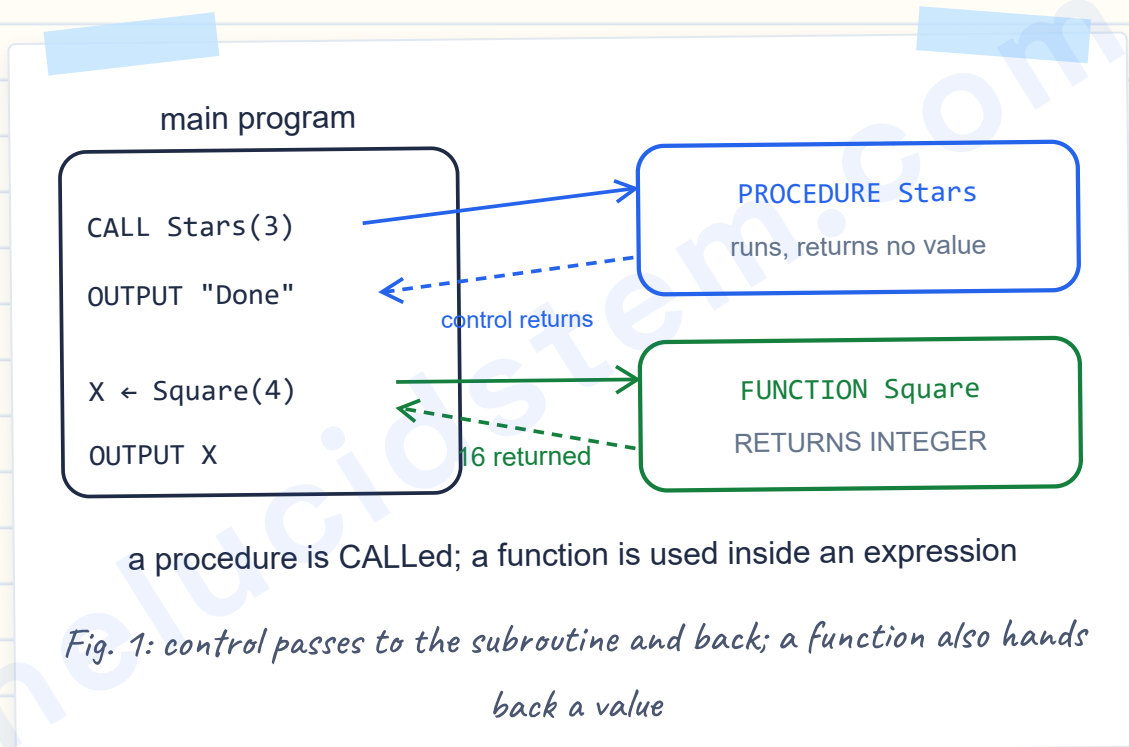
`11 AND < 18` is wrong; write

`Age > 11 AND Age < 18`.

6 Procedures, functions, parameters and scope

Syllabus 8.1

A **procedure** is a named block of code that performs a task; it is run with **CALL** as a statement on its own and does not return a value. A **function** is a named block of code that **returns a single value**, so it is used inside an expression; its header states the data type it RETURNS. **Parameters** are the variables in the header that receive the values passed in when it is called (up to three in the exam). A **local** variable is declared inside a subroutine and can only be used there; a **global** variable is declared in the main program and can be used anywhere.



Worked example: Two calls

Using the procedure above, the main program is `CALL Stars(2)` then `CALL Stars(1)`. How many stars are output?

1. The first call sets Number to 2: the loop outputs 2 stars.
2. Control returns, then the second call sets Number to 1: 1 star.

Answer: 3 stars

Procedure or function?

- ✓ **Procedure:** PROCEDURE ...
ENDPROCEDURE, run with CALL,
returns nothing.
- ✓ **Function:** FUNCTION ... RETURNS type
... RETURN value ... ENDFUNCTION,
used in an expression.
- ✓ **Parameters:** name and data type in
the header; values passed in the call, in
the same order.
- ✓ **Scope:** local inside a subroutine only;
global anywhere in the program.

Watch out!

A procedure with no parameters is still called with CALL, e.g. `CALL DrawLine`. Do not write a procedure call inside an expression: it has no value to give back.

Never use CALL with a function. Its value must be used: assigned, output or part of a calculation, e.g.

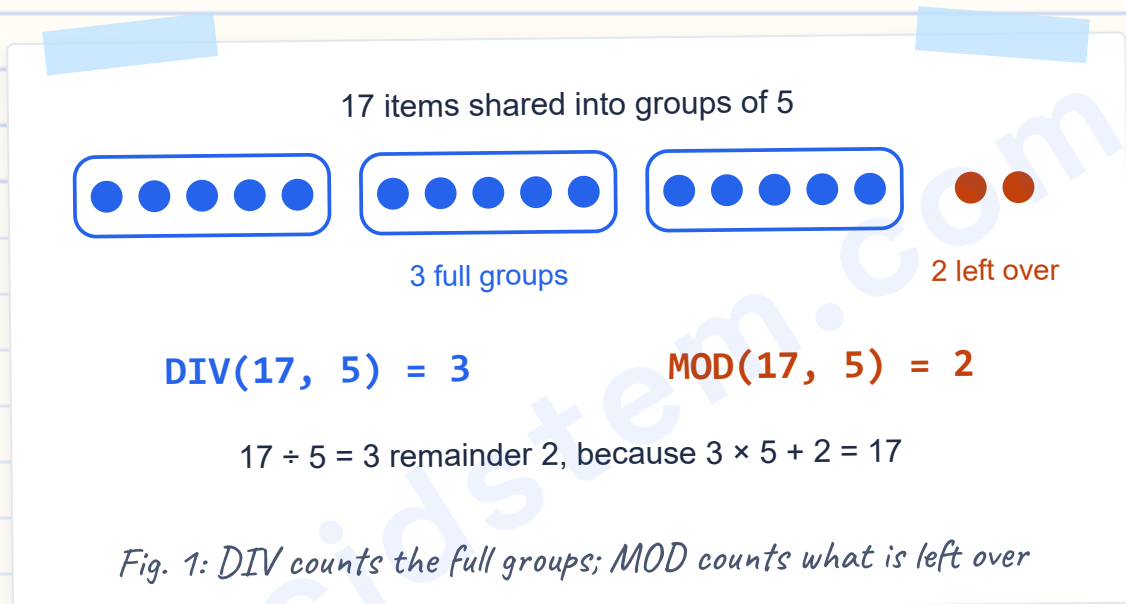
`Total ← SumSquare(2, 3)`.

The lab program outputs Local Total 6 and then Global Total 10: assigning to a local variable with the same name never touches the global one.

7 Library routines: MOD, DIV, ROUND and RANDOM

Syllabus 8.1

A **library routine** is a pre-written, pre-tested subroutine that comes with a programming language. **DIV(a, b)** returns the whole-number quotient of a divided by b , with the fractional part discarded. **MOD(a, b)** returns the **remainder**. **ROUND(x, places)** returns x rounded to that number of decimal places. **RANDOM()** returns a random number between 0 and 1 inclusive.



Worked example: Splitting a number

Evaluate $\text{DIV}(472, 100)$, $\text{MOD}(472, 10)$ and $\text{MOD}(\text{DIV}(472, 10), 10)$.

1. $472 \div 100 = 4$ remainder 72, so DIV gives 4.
2. $472 \div 10 = 47$ remainder 2, so MOD gives 2.
3. $\text{DIV}(472, 10) = 47$, then $\text{MOD}(47, 10) = 7$.

Answer: 4, 2 and 7: the hundreds, units and tens digits

Divide, then split.

- ✓ **DIV(a, b):** whole number of times b goes into a .
- ✓ **MOD(a, b):** the remainder left over.
Check: $\text{DIV} \times b + \text{MOD} = a$.
- ✓ **ROUND(x, n):** look at the next decimal place; 5 or more rounds up.
- ✓ **RANDOM():** 0 to 1; multiply, ROUND to 0 places, then add to set the range.

Watch out!

MOD gives the remainder, not the decimal part: MOD(7, 2) is 1, not 0.5. DIV(7, 2) is 3, not 3.5.

ROUND returns a new value; it does not change the variable unless you assign the result back, e.g. `Price ← ROUND(Price, 2)`.

`ROUND(RANDOM() * 6, 0)` gives 0 to 6, which is seven possible values, not a dice roll. Check the smallest and largest values of your expression.

8 Writing maintainable programs

Syllabus 8.1

A **maintainable** program is easy for another programmer to understand, correct and update. Use **meaningful identifiers** for variables, constants, arrays, procedures and functions, so each name says what it holds or does. Add **comments** (starting with `//` in pseudocode) to explain the purpose of sections of code; the computer ignores them. Use **constants** for fixed values and **procedures and functions** to split the program into named, reusable parts.

hard to maintain

```
X ← 0
FOR I ← 1 TO 30
  INPUT Y
  X ← X + Y
NEXT I
OUTPUT X / 30
```

maintainable

```
// average mark for one class
CONSTANT ClassSize ← 30
TotalMark ← 0
FOR Student ← 1 TO ClassSize
  INPUT Mark
  TotalMark ← TotalMark + Mark
NEXT Student
OUTPUT TotalMark / ClassSize
```

same algorithm: meaningful names, a comment and a constant

Fig. 1: both programs do the same job, but only one explains itself

Worked example: Useful or not?

Which comment is more useful above the line `Count ← Count + 1` inside a validation loop: (A) `// add 1 to Count`, or (B) `// count the number of failed attempts`?

1. Comment A only repeats what the code already says.
2. Comment B explains the purpose: what the count is for.

Answer: B: comments should explain why, not repeat the code

Name, explain, reuse.

- ✓ **Identifiers:** meaningful names for variables, constants, arrays, procedures and functions.
- ✓ **Comments:** // in pseudocode; explain purpose, not every line.
- ✓ **Constants:** one place to change a fixed value.
- ✓ **Subroutines:** write once, call many times, test separately.

Watch out!

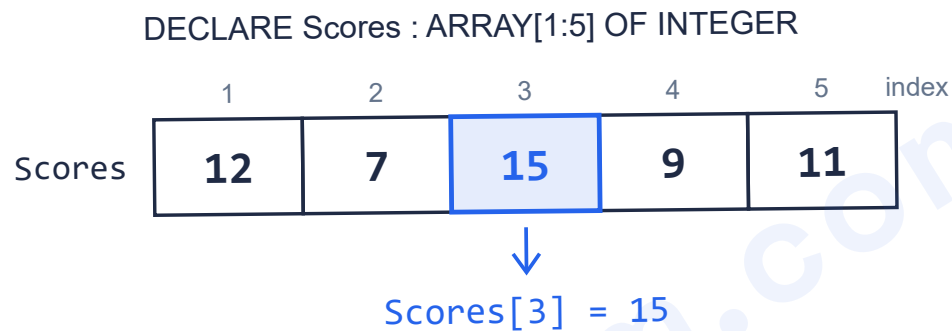
"Use good variable names" is too vague for a mark. Say *meaningful identifiers* and explain the benefit: another programmer can see what each variable stores, so the code is easier to understand and update.

Comments are ignored by the translator, so they do not make the program run faster or slower, and they do not fix errors. They help people.

9 One-dimensional arrays

Syllabus 8.2

An **array** is a data structure that holds a **fixed number** of elements of the **same data type** under one identifier. Each element is reached by its **index**, written in square brackets: `Scores[3]`. Declare it with its lower and upper bounds: `DECLARE Scores : ARRAY[1:5] OF INTEGER`. The first index can be 0 or 1 (exam pseudocode generally uses 1). Because the index can be a **variable**, a FOR loop can visit every element in turn.



one identifier, five elements of the same data type, each reached by its index

Fig. 1: a one-dimensional array of five integers

Worked example: Finding the largest

Scores holds 12, 7, 15, 9, 11. Trace this code and give the output.

`Highest ← Scores[1]`, then `FOR Index ← 2 TO 5`: `IF Scores[Index] > Highest THEN Highest ← Scores[Index]`; after the loop, `OUTPUT Highest`.

1. Highest starts as `Scores[1] = 12`.
2. Index 2: `7 > 12?` No. Index 3: `15 > 12?` Yes, `Highest ← 15`.
3. Index 4: `9 > 15?` No. Index 5: `11 > 15?` No.

Answer: 15

Identifier, index, element.

- ✓ **Declare:** DECLARE Name : ARRAY[1:n] OF type.
- ✓ **Access:** Name[Index], where Index can be a variable or expression.
- ✓ **Loop:** FOR Index ← 1 TO n to input, total or search every element.
- ✓ **Max and min:** start with the first element, then compare the rest.

Watch out!

Using an index outside the bounds, such as Scores[6] or Scores[0] for ARRAY[1:5], is an error: that element does not exist.

Start Highest at the first element, not at 0: if every value were negative, a starting value of 0 would be wrongly output as the largest.

thelucidstem.com

10 Two-dimensional arrays and nested loops

Syllabus 8.2

A **two-dimensional (2D) array** is a table of elements of the same data type, with **two indexes**: one for the row and one for the column. Declare it with two sets of bounds: `DECLARE Sales : ARRAY[1:3, 1:4] OF INTEGER` (3 rows, 4 columns, 12 elements). Access an element with `Sales[Row, Column]`. To process every element, use **nested loops**: the inner loop runs completely for every pass of the outer loop.

Sales : ARRAY[1:3, 1:4] OF INTEGER

	column (week)			
	1	2	3	4
row 1	5	8	2	6
row 2	3	9	4	7
row 3	6	1	5	8

`Sales[2, 3] = 4`

first index = row, second index = column

Fig. 1: a 2D array of sales for 3 shops (rows) over 4 weeks (columns)

Worked example: Row totals

Using the Sales table, what does the program above output?

1. Row 1: RowTotal resets to 0, then $5 + 8 + 2 + 6 = 21$.
2. Row 2: reset, then $3 + 9 + 4 + 7 = 23$.
3. Row 3: reset, then $6 + 1 + 5 + 8 = 20$.
4. The inner statement runs $3 \times 4 = 12$ times in all.

Answer: Shop 1 total: 21, Shop 2 total: 23, Shop 3 total: 20

Row first, then column.

- ✓ **Declare:** `ARRAY[1:rows, 1:columns]` OF type.
- ✓ **Access:** `Name[Row, Column]`; elements = rows × columns.
- ✓ **Nested loops:** the inner loop completes for each pass of the outer loop.
- ✓ **Totals:** reset a row total inside the outer loop, before the inner loop.

Watch out!

Do not swap the indexes. `Sales[3, 2]` is row 3, column 2 (the value 1), but `Sales[2, 3]` is row 2, column 3 (the value 4).

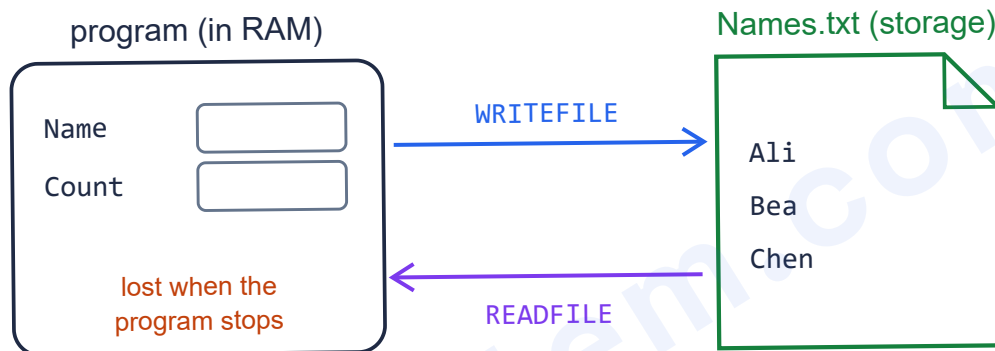
`RowTotal ← 0` must be inside the outer loop but before the inner loop. Outside both loops, each row's total would include the rows before it.

thelucidstem.com

11 Reading from and writing to files

Syllabus 8.3

Variables are held in main memory and are **lost when the program ends**. Storing data in a **file** keeps it permanently in secondary storage, so it can be **used again** later, by this or another program. In pseudocode: `OPENFILE "Names.txt" FOR READ` or `FOR WRITE`, then **READFILE** to read the next line into a variable, **WRITEFILE** to write a value, and **CLOSEFILE** when finished. Opening a file `FOR WRITE` creates a new file: **any existing data in it is lost**.



data in a file is kept after the program ends

Fig. 1: **WRITEFILE** copies data from a variable into the file; **READFILE** copies it back

Worked example: Running it twice

The program above is run with the inputs Ali, Bea and Chen. It is then run again with Dev, Eve and Fay. What does Names.txt contain at the end?

1. First run: the file holds Ali, Bea, Chen, one per line.
2. Second run: `OPENFILE ... FOR WRITE` creates a new file, so the old names are lost.
3. The three new names are written.

Answer: Dev, Eve and Fay only

Open, use, close.

- ✓ **Purpose:** data in a file is kept after the program ends and can be used again.
- ✓ **Open:** `OPENFILE "name" FOR READ` or `FOR WRITE`; one mode at a time.
- ✓ **Use:** `READFILE "name"`, Variable reads the next line; `WRITEFILE "name"`, Value writes one.
- ✓ **Close:** `CLOSEFILE "name"`. `WRITE` mode replaces any existing contents.

Watch out!

"Files save data" earns little. Say that data in variables is lost when the program ends, but data in a file is stored permanently so it can be accessed again later.

Opening an existing file in `WRITE` mode wipes it. Only one mode is used at a time, so to read and then rewrite a file, close it and open it again in the other mode.

Always `CLOSEFILE` when you have finished with a file. Closing it makes sure the data is saved and lets the file be opened again, for example in the other mode.