Exercise walk-through

Constructs ▪ 11.2

eXercise

You must be able to

- write `IF…ELSE`, **nested** `IF`, and `CASE` structures
- write a **count-controlled** (`FOR`), **pre-condition** (`WHILE`) and **post-condition** (`REPEAT`) loop
- **justify** which loop best suits a problem

Method — Selection: IF and CASE

```
IF Age >= 18 THEN
    OUTPUT "Adult"
ELSE
    OUTPUT "Minor"
ENDIF
```

Method — Selection: IF and CASE

For one value against **several** options, a `CASE` is cleaner than deep nested `IF`s:

```
CASE OF Grade
    "A": OUTPUT "Excellent"
    "B": OUTPUT "Good"
    OTHERWISE: OUTPUT "Try again"
ENDCASE
```

Method — The three loops

```
FOR i ← 1 TO 10           // count-controlled - known repeats
    OUTPUT i
NEXT i
```

`WHILE` tests **before** the body (may run **0** times); `REPEAT` tests **after** (runs **at least once**):

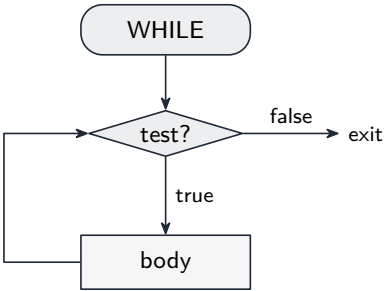
Method — The three loops

```
WHILE Total < 100 DO
    INPUT n
    Total ← Total + n
ENDWHILE
```

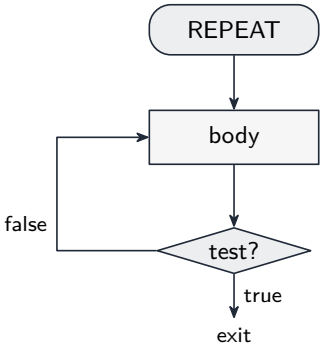
```
REPEAT
    INPUT Password
UNTIL Password = "OPEN"
```

Method — The three loops

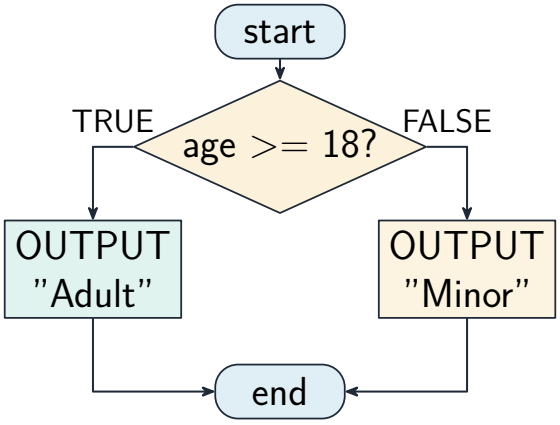
pre-condition (WHILE)



post-condition (REPEAT)



Method — The three loops



Selection (if-else) is the other core construct

Method — Choosing a loop

- repeats **known** in advance → `FOR`
- may need **zero** passes → `WHILE`
- needs **at least one** pass → `REPEAT`

Method — Tracing a loop (trace table)

A **trace table** 追踪表 dry-runs code by hand: one column per variable (plus OUTPUT), one row each time a value changes. Trace this for `N = 3`:

```
Total ← 0
FOR i ← 1 TO N
    Total ← Total + i
NEXT i
OUTPUT Total
```

Method — Tracing a loop (trace table)

i	Total	OUTPUT
	0	
1	1	
2	3	
3	6	6

Practice 2.1 ▪ 2 marks

Write an `IF…ELSE` that outputs `"Pass"` when `Mark >= 50`, otherwise `"Fail"`.

Solution 2.1

Method: Selection: IF and CASE

Worked steps

```
IF Mark >= 50 THEN
    OUTPUT "Pass"
ELSE
    OUTPUT "Fail"
ENDIF
```

Solution 2.1

M1 `IF…ELSE…ENDIF;` **A1** correct condition + outputs.

Practice 2.2 ▪ 3 marks

Rewrite this nested IF as a CASE structure.

```
IF Grade = "A" THEN
    OUTPUT "Excellent"
ELSE
    IF Grade = "B" THEN
        OUTPUT "Good"
    ELSE
        OUTPUT "Try again"
    ENDIF
ENDIF
```

Solution 2.2

Method: Selection: IF and CASE

Worked steps

```
CASE OF Grade
    "A": OUTPUT "Excellent"
    "B": OUTPUT "Good"
    OTHERWISE: OUTPUT "Try again"
ENDCASE
```

Solution 2.2

M1 `CASE OF … ENDCASE`; A1 two value branches; A1 `OTHERWISE`.

Practice 2.3 ▪ 2 marks

State which loop may run **zero** times and which runs **at least once**.

Solution 2.3

Method: The three loops

Worked steps

`WHILE` (pre-condition) may run **zero** times; `REPEAT…UNTIL` (post-condition) runs **at least once** B1 B1.

Practice 2.4 ▪ 2 marks

Write a `FOR` loop that outputs the **even** numbers from 2 to 20.

Solution 2.4

Worked steps

```
FOR i ← 2 TO 20 STEP 2
    OUTPUT i
NEXT i
```

M1 `FOR … STEP 2;` A1 range 2 to 20. *(A step of +2 from 2; outputting `i` only when `i MOD 2 = 0` also scores.)*

Exam-style 3.1 ▪ 4 marks

A menu uses a number `Choice`. Write a `CASE` structure: 1→`"New"`, 2→`"Open"`, 3→`"Save"`, otherwise→`"Invalid"`.

Solution 3.1

Method: Selection: IF and CASE

Worked steps

```
CASE OF Choice
    1: OUTPUT "New"
    2: OUTPUT "Open"
    3: OUTPUT "Save"
    OTHERWISE: OUTPUT "Invalid"
ENDCASE
```

Solution 3.1

M1 `CASE OF … ENDCASE`; A1 three branches; A1 correct outputs; A1 `OTHERWISE`.

Exam-style 3.2 ▪ 3 marks

A program must keep asking for a password until the user types `"OPEN"`. Write the loop using the **most suitable** construct, and **justify** your choice.

Solution 3.2

Method: The three loops

Worked steps

```
REPEAT
    INPUT Password
UNTIL Password = "OPEN"
```

Solution 3.2

A **post-condition (`REPEAT…UNTIL`)** loop, because the password must be entered **at least once** before it can be tested M1 `REPEAT…UNTIL`; A1 correct condition; B1 justification.

Exam-style 3.3 ▪ 3 marks

Write a **pre-condition** loop that reads numbers and adds them to a `Total`, stopping once `Total` reaches **100**.

Solution 3.3

Worked steps

```
Total ← 0
WHILE Total < 100 DO
    INPUT n
    Total ← Total + n
ENDWHILE
```

Solution 3.3

M1 `Total ← 0;` M1 `WHILE Total < 100;` A1 `INPUT` + add inside.

Exam-style 3.4 ▪ 4 marks

Complete the **trace table** for this algorithm when the input X is 5.

```
A ← 1
B ← 1
WHILE A < X DO
    B ← B * 2
    A ← A + 1
ENDWHILE
OUTPUT B
```

Exam-style 3.4 ▪ 4 marks

A	B	OUTPUT
1	1	

Solution 3.4

Method: The three loops

Worked steps

`B` doubles on each pass until `A` reaches 5:

A	B	OUTPUT
1	1	
2	2	
3	4	
4	8	
5	16	16

Solution 3.4

M1 `A` column correct; M1 `B` column doubling; A1 stops when `A = 5`; A1 OUTPUT 16.

Exam-style 3.5 ▪ 6 marks

A program keeps the time of day in three global integer variables `HH`, `MM` and `SS`.
(a) A procedure `Tick()` is called once every second and must advance the time correctly, rolling over at 60 seconds, 60 minutes and 24 hours. **Write** pseudocode for `Tick()`.
(b) **Complete** the trace table by dry running your procedure from the starting time shown, for **three** calls.

call	HH	MM	SS
start	10	59	58
1	_____	_____	_____
2	_____	_____	_____
3	_____	_____	_____

Exam-style 3.5 ▪ 6 marks

(c) **Explain** why the three `IF` statements must be **nested** or tested in the right order, giving what would go wrong otherwise.
(d) The variables are **global**. **Explain** one drawback of that, and **state** how you would restructure `Tick()` to avoid it.
(e) **State** which loop construct you would use to run the clock until the user stops it, and **justify** your choice.

Solution 3.5

Worked steps

(a)

```
PROCEDURE Tick()
    SS ← SS + 1
    IF SS = 60
      THEN
        SS ← 0
        MM ← MM + 1
        IF MM = 60
          THEN
            MM ← 0
            HH ← HH + 1
            IF HH = 24 THEN HH ← 0 ENDIF
        ENDIF
    ENDIF
ENDPROCEDURE
```

Solution 3.5

B1 B1 B1 B1 B1 B1 *increment seconds; reset seconds and carry into minutes; reset minutes and carry into hours; hours wrap at 24; the carries nested, not independent; correct pseudocode syntax*

(b) Call 1: `10 59 59`; call 2: `11 00 00`; call 3: `11 00 01` B1 B1 B1.

(c) The minutes can only roll over **because** the seconds just did, and the hours only because the minutes did B1. If the three were tested independently, `MM` would be checked against 60 before it had been incremented on this tick, so the rollover would be a second late B1; worse, testing hours first would let the clock show `11:59:60` for one tick B1.

(d) Any global variable can be changed by **any** part of the program, so a fault anywhere can corrupt the time and be hard to trace B1 B1. Pass the three values in as parameters and return the updated time — or hold them in one record passed by reference — so only `Tick()` can change them B1.

(e) A **post-condition** loop, `REPEAT … UNTIL Stopped`, because the clock must tick at least once before the user has any chance to stop it B1 B1.