Iteration

Ref As Pseudocode

FOR loops and STEP

Iteration means repeating a group of statements. A **loop** is the structure that does the repeating. Pseudocode has three loops, and each one fits a different job.

The count-controlled loop

Use `FOR` when you know **how many times** to repeat before the loop starts. It is called a **count-controlled loop** .

```
FOR <identifier> ← <value1> TO <value2>
   <statement(s)>
NEXT <identifier>
```

```
DECLARE Counter : INTEGER
FOR Counter ← 1 TO 5
   OUTPUT Counter, " x 3 = ", Counter * 3
NEXT Counter
```

- The loop counter (`Counter` here) must be an `INTEGER`.
- It takes **every** value from `value1` to `value2`, **including both ends**. So `1 TO 5` runs five times.
- If `value1 = value2`, the body runs once. If `value1 > value2`, it does not run at all.
- Writing the counter's name after `NEXT` is good practice, especially with one loop inside another.
- Do not change the counter yourself inside the loop. The `FOR` loop does that.

STEP

Add `STEP` to count in jumps other than 1. The step can be **negative** to count down.

```
FOR <identifier> ← <value1> TO <value2> STEP <increment>
```

```
DECLARE i : INTEGER
FOR i ← 0 TO 20 STEP 5
   OUTPUT i
NEXT i
FOR i ← 3 TO 1 STEP -1
   OUTPUT "Countdown ", i
NEXT i
```

The loop stops as soon as the next value would go **past** `value2`. So `FOR i ← 1 TO 10 STEP 4` gives 1, 5 and 9 only.

Loops inside loops

A loop can sit inside another loop. The inner loop runs **completely** for each single pass of the outer loop.

```
DECLARE Row : INTEGER
DECLARE Column : INTEGER
DECLARE Line : STRING
FOR Row ← 1 TO 3
   Line ← ""
   FOR Column ← 1 TO 4
      Line ← Line & NUM_TO_STR(Row * Column) & " "
   NEXT Column
   OUTPUT Line
NEXT Row
```

(`NUM_TO_STR` turns a number into a string so that `&` can join it; see topic 7.) The outer loop runs 3 times and the inner loop runs 4 times each time: 12 passes in total.

Common mistakes

- `FOR Counter = 1 TO 5`. The `FOR` line uses the arrow `←`.
- Counting down with no `STEP -1`. `FOR i ← 10 TO 1` does not run at all.
- Ending the inner and outer loops in the wrong order. The loop that opened **last** must close **first**.

WHILE, the pre-condition loop

Use `WHILE` when you do **not** know how many times to repeat, and the loop may need to run **zero** times.

```
WHILE <condition>
   <statement(s)>
ENDWHILE
```

```
DECLARE Number : INTEGER
Number ← 40
WHILE Number > 9
   Number ← Number - 9
ENDWHILE
OUTPUT Number
```

- The condition is tested **before** each pass. That is why it is called a **pre-condition loop** .

- The loop repeats **while** the condition is `TRUE`, and stops when it becomes `FALSE`.
- If the condition is `FALSE` the first time, the body **never runs**.

■ Something inside must change

The body must change something that the condition tests. If it does not, the condition stays `TRUE` for ever and you get an **infinite loop** .

```
DECLARE Balance : REAL
DECLARE Years : INTEGER
Balance ← 1000.0
Years ← 0
WHILE Balance < 2000.0
   Balance ← Balance * 1.1
   Years ← Years + 1
ENDWHILE
OUTPUT "Doubled after ", Years, " years"
```

Here `Balance` grows on every pass, so the loop must end.

Common mistakes

- Forgetting `ENDWHILE`.
- Not setting the variable before the loop. The condition is tested first, so the variable needs a value already.
- An infinite loop: nothing in the body changes the condition.

REPEAT, the post-condition loop

Use `REPEAT` when the body must run **at least once**, for example to ask the user for input and then check it.

```
REPEAT
   <statement(s)>
UNTIL <condition>
```

```
DECLARE Mark : INTEGER
REPEAT
   OUTPUT "Enter a mark from 0 to 100"
   INPUT Mark
UNTIL Mark >= 0 AND Mark <= 100
OUTPUT "Mark saved: ", Mark
```

- The condition is tested **after** each pass: a **post-condition loop** .
- The loop repeats **until** the condition is `TRUE`. When it becomes `TRUE`, the loop stops.
- There is no `ENDREPEAT`: the `UNTIL` line closes the loop.

Notice that the two conditions are opposites: `WHILE` keeps going while its condition is `TRUE`; `REPEAT` stops when its condition is `TRUE`.

```
DECLARE Total : INTEGER
DECLARE Count : INTEGER
Total ← 0
Count ← 0
REPEAT
   Count ← Count + 1
   Total ← Total + Count
UNTIL Total > 20
OUTPUT "Added 1 to ", Count, " to pass 20: total ", Total
```

Common mistakes

- Writing the condition the wrong way round. "Keep asking while the age is wrong" becomes `UNTIL` the age is **right**.
- Using `REPEAT` when the body might need to run zero times. Then use `WHILE`.

Choosing the right loop

Loop	Use it when	Tested	Runs at least
`FOR`	you know the number of passes before you start	(counts)	0 times
`WHILE`	you do not know, and it may need **zero** passes	**before** each pass	0 times
`REPEAT`	you do not know, and it needs **at least one** pass	**after** each pass	1 time

Typical exam situations:

- Go through every element of an array: `FOR`.
- Read a file until it ends: `WHILE NOT EOF(...)`, because the file may be empty.
- Ask for input until it is valid: `REPEAT ... UNTIL`.
- Search an array and stop early once you find the item: `WHILE` (or `REPEAT`) with a flag.

■ Practice

1. Output the even numbers from 2 to 20 with one `FOR` loop.

2. Starting at 1, keep doubling a number. Output how many doublings it takes to go above 1000. Use a `WHILE` loop.

3. Rewrite this `WHILE` loop as a `REPEAT` loop that gives the same output:

```
Count ← 1
WHILE Count <= 3
   OUTPUT Count
   Count ← Count + 1
ENDWHILE
```

<details> <summary>Model answer</summary>

```
DECLARE Number : INTEGER
FOR Number ← 2 TO 20 STEP 2
   OUTPUT Number
NEXT Number
```

```
DECLARE Value : INTEGER
DECLARE Doublings : INTEGER
Value ← 1
Doublings ← 0
WHILE Value <= 1000
   Value ← Value * 2
   Doublings ← Doublings + 1
ENDWHILE
OUTPUT Doublings
```

It prints `10`, because 2 to the power 10 is 1024.

```
DECLARE Count : INTEGER
Count ← 1
REPEAT
   OUTPUT Count
   Count ← Count + 1
UNTIL Count > 3
```

The `WHILE` condition `Count <= 3` turns into its opposite, `Count > 3`. The two loops only give the same output because the `WHILE` body runs at least once here.

</details>