11.3.2 Parameters by value and by reference, and variable scope

Name: _____ Class: _____ Date: _____ **Total: 32 marks**

KEY WORDS return value • pass by reference • pass by value • pseudocode • procedure

Objective

Learn how a value gets **into** and **out of** a subroutine: **pass by value** against **pass by reference**, and the **scope** and lifetime of a variable. Sheets 11.3 and 11.3.1 wrote procedures and functions whose parameters only bring values in. This sheet traces what a call changes in the caller, chooses the right mode for each parameter, and explains why a **local variable** loses its value between calls. Sheet 11.3.3 then puts whole programs together from modules.

You must be able to:

- use parameters, understanding that a parameter may be passed by reference or by value
- trace what a subroutine changes in the caller, and what it does not
- choose the right passing mode for each parameter of a procedure
- use the terminology of procedures and functions when explaining your choice
- explain the difference between a local variable and a **global variable**, and write efficient pseudocode that prefers locals and parameters

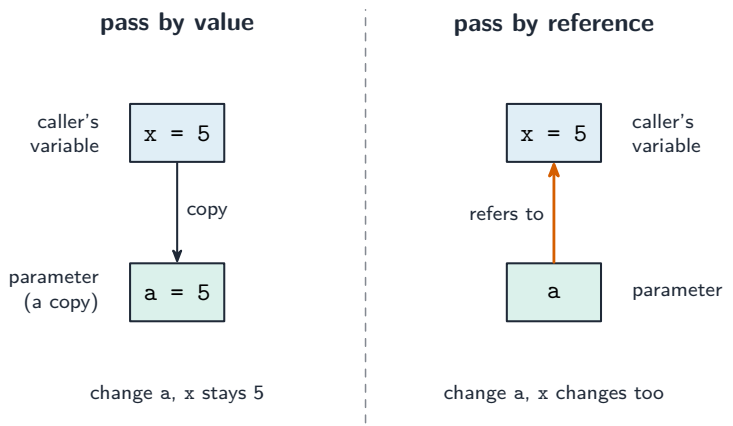
1 Worked examples

Study these first. Each one shows a **method** that a question later on this sheet needs.

By value or by reference?

- A parameter is passed in one of two **modes**, written in the header.
- If no mode is written, `BYVAL` is assumed.

	`BYVAL` (by value)	`BYREF` (by reference)
the parameter is	a **copy** of the variable	the caller's **own variable**
a change inside	is **lost** when the call ends	**stays** in the caller's variable
use it when	the value is only read	the caller must see the change



Tracing a call

```
PROCEDURE Update(BYREF Count : INTEGER, BYVAL Increment : INTEGER)
    Count ← Count + Increment
    Increment  ← 0
ENDPROCEDURE

Total  ← 12
Amount ← 4
CALL Update(Total, Amount)
OUTPUT Total, Amount
```

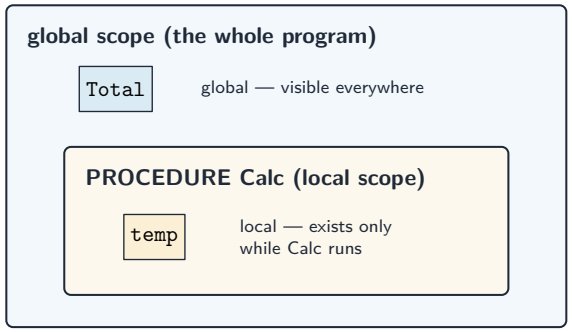
1. `Count` is `BYREF`, so `Count` **is** `Total`: the addition writes `16` into `Total`.
2. `Increment` is `BYVAL`, so it is a copy of `Amount`: `Increment ← 0` changes the copy only, and `Amount` stays `4`.
3. The output is **`16, 4`**. (With `BYVAL Count` the output would be `12, 4`.)

A trace question is marked on the **reason**: write "`Count` is `BYREF`, so it is `Total` itself", not only the number.

Choosing the mode

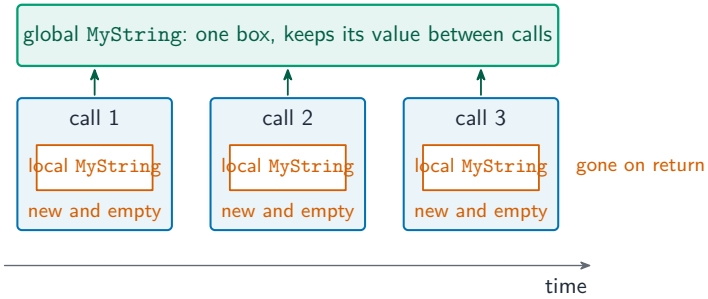
- The subroutine only **reads** the value → `BYVAL` (a copy protects the caller).
- It must **update** the caller's variable → `BYREF`. `Swap` needs **both** parameters `BYREF`.
- It produces **exactly one value** → use a **function** instead.
- The mode is part of the **interface**: changing `BYVAL` to `BYREF` changes the interface, even though the call looks the same.

Scope and lifetime



Scope is where a variable can be used. **Lifetime** is how long it exists.

	global variable	local variable
declared	outside every subroutine	inside one subroutine
scope	anywhere in the program	only inside that subroutine
lifetime	the whole run of the program	from the call to the **return**



A procedure adds 1 to a counter, outputs it, and is called three times. **Global** counter: 1, 2, 3. **Local** counter: 1, 1, 1, because a new, empty variable is made at each call. To keep a value between calls, make it global or pass it BYREF.

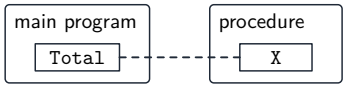
Why local variables are still preferred:

- the same name can be reused in another subroutine
- the value cannot be changed by mistake elsewhere
- the memory is released at the return
- the subroutine is self-contained, so it can be tested alone and reused

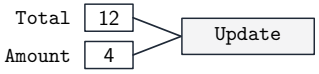
2 Practice

Now use the methods above. The first questions need one idea each.

2.1 State what is meant by passing a parameter **by value**, and what is meant by passing it **by reference**. [2]



2.2 For the Update procedure in the worked example, Total is 12 and Amount is 4 before the call. State the two values output, and give the reason for each. [2]



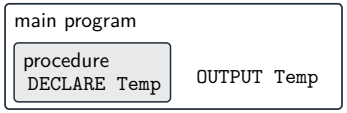
2.3 The header is changed so that **both** parameters are BYVAL. State the two values now output, and explain the change. [2]

2.4 A procedure's header is changed from BYVAL to BYREF for one parameter. State whether the **interface** of the procedure has changed, and state whether the **argument** written at the call site has to change. [2]

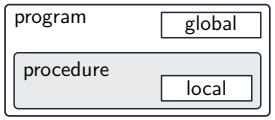
```
PROCEDURE Add(BYVAL N : INTEGER)
PROCEDURE Add(BYREF N : INTEGER)
```

2.5 State what is meant by the **scope** of a variable. [1]

2.6 A programmer declares a variable inside a procedure and then tries to use it in the main program. State what happens, and why. [2]



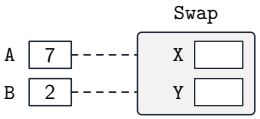
2.7 State **two** advantages of using a local variable rather than a global variable. [2]



3 Exam-style questions

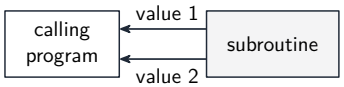
3.1 A procedure `Swap()` is intended to exchange the values of two integer variables. Before the call, `A` is `7` and `B` is `2`.

(a) The header is `PROCEDURE Swap(BYVAL X : INTEGER, BYVAL Y : INTEGER)`. State the values of `A` and `B` after the call, and explain your answer. [2]



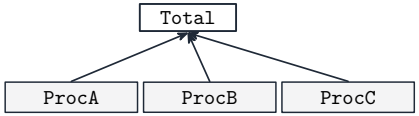
(b) Write a header that makes the procedure work as intended. [2]

3.2 A subroutine must give the calling program **two** values. Describe **two** different ways of doing this, and state one reason to prefer each. [4]



3.3 A program stores a running total in a global variable that three different procedures update.

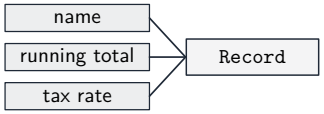
(a) State **two** problems this can cause. [2]



(b) Describe one change that removes those problems. [2]

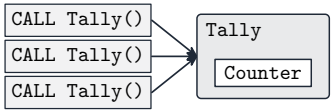
3.4 A procedure `Record()` takes three parameters: a customer's name, which it only reads; a running total, which it must increase; and a tax rate, which it only reads.

State, with a reason, whether each parameter should be passed `BYVAL` or `BYREF`. [4]



3.5 A procedure declares a counter as a local variable, adds 1 to it and outputs it. The procedure is called three times.

State the three values output, explain why, and name **two** ways to make the counter keep its value between calls. [3]



Solutions

Each answer shows the steps, then the **result**.

2.1

- **by value** – the subroutine receives a **copy** of the value, so changes it makes do not affect the caller's variable
- **by reference** – the parameter refers to the caller's **own** variable, so changes it makes do affect the caller

2.2

- `Total` is **16**: `Count` is `BYREF`, so it is `Total`'s own box and the addition is written into it
- `Amount` is **4**: `Increment` is `BYVAL`, so setting it to 0 only empties the copy

2.3

- `Total` is **12** and `Amount` is **4**
- both parameters are now copies, so the addition happens inside the procedure and nothing reaches the caller

2.4

- the **interface has changed**: the passing mode is part of the interface, which is everything a caller must know in order to use the procedure
- the **argument does not change**: it is still written as the variable's name at the call site, exactly as before – what changes is what the procedure may do to it

2.5

- the scope of a variable is the part of the program in which that name can be accessed

2.6

- the variable is **local** to the procedure, so it does not exist outside it
- the main program cannot access the name, so this is an error (and the value is destroyed when the procedure returns)

2.7

- any two: the same identifier may be used in another subroutine without a clash; the value cannot be changed accidentally by another part of the program; the memory is released when the subroutine ends; the subroutine is self-contained, so it can be tested on its own and reused

3.1

(a) `A` is still **7** and `B` is still **2**

- both parameters are copies, so the exchange happens inside the procedure and is lost when it returns

(b) both parameters must be passed by reference `PROCEDURE Swap(BYREF X : INTEGER, BYREF Y : INTEGER)`

3.2

- **two `BYREF` parameters** – the procedure writes into the caller's own variables
- prefer this when neither value is "the answer": both are updates the caller asked for
- **a function returning one value, with the other as a `BYREF` parameter**
- prefer a function when one value is clearly the result, because the return value can be used directly in an expression

3.3

(a) any two: any procedure can change it, so a fault is hard to trace to one place; a change made by one procedure can be overwritten by another; the procedures cannot be tested on their own; the name cannot be reused elsewhere

(b) make the total a **local** variable in the calling program and pass it to each procedure as a `BYREF` parameter

- each procedure is then self-contained and can be tested alone, and only a procedure actually given the total can change it

3.4

- the name: **`BYVAL`** – it is only read, and a copy protects the caller's variable
- the running total: **`BYREF`** – the procedure must increase it and the new value has to reach the caller
- the tax rate: **`BYVAL`**: it is only read, and nothing needs to go back to the caller

3.5

- the outputs are **1, 1, 1**
- a local variable is created when the procedure is called and destroyed when it returns, so each call starts with a new, empty counter and the earlier value is gone
- to keep it: make the counter **global**, or pass it in as a **`BYREF` parameter** so the box belongs to the caller