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**: `Step` 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 scope of a variable is the part of the program in which that name can be accessed

2.5

- 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

2.6

- 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**, or returning a record/array holding both
- 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 array of prices: **`BYVAL`** – it is only read (passing it `BYREF` would avoid copying it, but nothing needs to go back)

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