## Solutions

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

**2.1**

- any one: data is **kept** after the program ends; it stores large amounts permanently; it lets programs share data / restart from a saved state

**2.2**

- when the **end of the file** has been reached — there is nothing left to read

**2.3**

- open for read, then close

```
OPENFILE "scores.txt" FOR READ
CLOSEFILE "scores.txt"
```

**2.4**

- any one: so **buffered writes are saved**; so the file is not left **locked** for other programs

**3.1**

- open, loop until EOF, read and output, then close

```
OPENFILE "names.txt" FOR READ
WHILE NOT EOF("names.txt") DO
    READFILE "names.txt", Line
    OUTPUT Line
ENDWHILE
CLOSEFILE "names.txt"
```

**3.2**

- open for write, loop 1 to 50, then close

```
OPENFILE "nums.txt" FOR WRITE
FOR i ← 1 TO 50
    WRITEFILE "nums.txt", i
NEXT i
CLOSEFILE "nums.txt"
```

**3.3**

- start `Count` at 0, then read every record

```
Count ← 0
OPENFILE "members.txt" FOR READ
WHILE NOT EOF("members.txt") DO
    READFILE "members.txt", Name
    Count ← Count + 1
ENDWHILE
CLOSEFILE "members.txt"
OUTPUT Count
```

- increment inside the loop; close and output

**3.4**

(a) the three items are of different lengths, so without a marker the program cannot tell where one field ends and the next begins

- the separator must be a character that can **never appear inside the data itself** — a surname could contain a space or a hyphen, so `|` is safer than either

(b)

```
PROCEDURE CountClass(Wanted : STRING)
    DECLARE Line, ThisClass : STRING
    DECLARE Count : INTEGER
    Count ← 0
    OPENFILE "members.txt" FOR READ
    WHILE NOT EOF("members.txt")
        READFILE "members.txt", Line
        ThisClass ← the text after the second "|" in Line
        IF ThisClass = Wanted THEN
            Count ← Count + 1
        ENDIF
    ENDWHILE
    CLOSEFILE "members.txt"
    OUTPUT Count
ENDPROCEDURE
```

- header with parameter; counter initialised; `OPENFILE … FOR READ`; `WHILE NOT EOF` with `READFILE`; the field extracted and compared; `CLOSEFILE` and output

(c) encrypted data is arbitrary bytes, so it may **contain the separator character itself**, which would split one field into two and corrupt every field after it on that line

- encode the encrypted values in a form that cannot contain `|` (hex or Base64), or use fixed-length fields instead of a separator

(d) in a **serial** file the records are in the order they were added, so a search must read from the start until it finds the member: on average half the file

- a **sequential** file is ordered by membership number, so the search can stop as soon as it passes the wanted key, and the file can be searched far faster

- the cost is that inserting a new member into a sequential file means rewriting the file to keep the order, whereas a serial file just appends