Past-paper walk-through

File Processing and Exception Handling ▪ 20.2

eXercise

Questions in this set

- 9618/31 N25 Q10 ▪ 3 marks
- 9618/32 N25 Q10 ▪ 4 marks
- 9618/33 N25 Q12 ▪ 5 marks
- 9618/41 J25 Q2 ▪ 25 marks
- 9618/42 N24 Q3 ▪ 17 marks

Reading the mark scheme

- **B1** a mark that stands on its own – the point is either made or not.
- **M1** a *method* mark: the right approach, even if the number is wrong.
- **A1** an *answer* mark, and it usually needs its M mark first.
- **C1** a working mark on the way to the answer.
- // separates answers the examiner accepts equally.
- **ecf** = error carried forward: a later part is marked on your own earlier answer.
- **owtte** = "or words to that effect" – the wording need not match.

Question 10

9618/31 N25 ▪ Q10 ▪ 3 marks

Explain what is meant by **exception handling**. Include an example of a possible cause of an exception in your answer. **[3]**

Solution 10

- One mark per mark point (Max 2)
- MP1
- Exception handling is a process that responds to unwanted / unexpected events when a program runs
- MP2
- ... to prevent the program / computer from stopping unexpectedly
- One mark for example (Max 1)
- MP3

Solution 10

- Programming errors
- MP4
- User errors
- MP5
- Hardware failure // losing connection to a device such as a printer

Question 10

9618/32 N25 ▪ Q10 ▪ 4 marks

An exception is an error that may cause a program to halt unexpectedly.

(a) Describe how program termination due to an exception can be avoided. **[2]**

(b) Identify **two** possible causes of exceptions.

1. 2. **[2]**

Solution 10

(a)

Mark scheme

- One mark per mark point (Max 2)
- MP1
- Use an exception handling routine (to respond to unwanted / unexpected events when the program is running)
- MP2
- Use of try ... except / catch // Generate some form of error message

Solution 10

(b) Mark scheme

- One mark per mark point (Max 2)
- MP1
- Coding errors
- MP2
- User errors
- MP3
- Hardware failure // Losing connection to a device e.g. a printer
- 2

Solution 10

- October/November
- 2025

Question 12

9618/33 N25 ▪ Q12 ▪ 5 marks

An exception can occur when running a program.

(a) Explain what is meant by an exception. **[3]**

(b) Identify **one** example of an exception and give **one** reason why the exception may cause a problem. **[2]**

Solution 12

(a) Mark scheme

- One mark per mark point (Max 3)
- MP1
- An exception is an unplanned / unexpected event that occurs when a program is running
- MP2
- ... due to an error in programming/logic that wasn't detected during program construction/compilation
- MP3
- The effect of an exception can be that the program halts unexpectedly.

Solution 12

(b) Mark scheme

- One mark per mark point (Max 2)
- MP1
- One mark for an example of an exception (see list in guidance)
- MP2
- One mark for the reason for the given exception
- Example answer
- Division by zero

Solution 12

- The processor will not be able to evaluate this answer because a number divided by zero is infinity, so the program will crash.

Question 2

9618/41 J25 ▪ Q2 ▪ 25 marks

A program reads data from a text file, splits the data depending on its content and stores the separated data into different files.

The text file `TheData.txt` contains 72 lines of data. Each line of data has an integer number and a string colour that are separated by a comma. For example, the first line in the file is:

```
10,red
```

The integer is `10` and the string is `"red"`

The file contains six different colours: red, green, blue, orange, yellow, pink.

(a) The function `ReadData()`:

Question 2

- prompts the user to enter a filename and reads this filename from the user
- opens the file and reads each line of data into a 1D array
- returns the populated 1D array.

The function needs to work for a file that contains an unknown number of lines.
Write program code for `ReadData()`

Save your program as **Question2_J25**.

Copy and paste the program code into part **2(a)** in the evidence document.

Question 2

[7]

Question 2

9618/41 J25 ▪ Q2 ▪ 25 marks

A program reads data from a text file, splits the data depending on its content and stores the separated data into different files.

The text file `TheData.txt` contains 72 lines of data. Each line of data has an integer number and a string colour that are separated by a comma. For example, the first line in the file is:

`10,red`

The integer is 10 and the string is `"red"`

The file contains six different colours: red, green, blue, orange, yellow, pink.

(b) The procedure `SplitData()` takes a 1D string array as a parameter with the identifier `DataArray`

Question 2

The procedure declares six 1D arrays: one array for each colour that appears in the file (red, green, blue, orange, yellow, pink).
The procedure accesses each string in `DataArray`. The data in each string is split into the integer and the colour. The integer is stored in the array that matches the colour.
For example, the first string in `DataArray` has the integer `10` and the colour `red`, so the integer `10` is stored in the array for the colour red.
Write program code for `SplitData()`

Save your program.

Copy and paste the program code into part **2(b)** in the evidence document.

Question 2

[6]

Question 2

9618/41 J25 ▪ Q2 ▪ 25 marks

A program reads data from a text file, splits the data depending on its content and stores the separated data into different files.

The text file `TheData.txt` contains 72 lines of data. Each line of data has an integer number and a string colour that are separated by a comma. For example, the first line in the file is:

`10,red`

The integer is 10 and the string is `"red"`

The file contains six different colours: red, green, blue, orange, yellow, pink.

(c) The procedure `StoreData()`:

- takes **two** parameters: a 1D array `DataToStore` and a filename
- opens the text file with the filename that is passed as a parameter
- appends each item of data from `DataToStore` to a new line in the text file
- uses exception handling when opening and writing data to the text file.

Question 2

Write program code for `StoreData()`
Save your program. Copy and paste the program code into part **2(c)** in the evidence document. **[5]**

Question 2

9618/41 J25 ▪ Q2 ▪ 25 marks

A program reads data from a text file, splits the data depending on its content and stores the separated data into different files.

The text file `TheData.txt` contains 72 lines of data. Each line of data has an integer number and a string colour that are separated by a comma. For example, the first line in the file is:

`10,red`

The integer is 10 and the string is `"red"`

The file contains six different colours: red, green, blue, orange, yellow, pink.

(d) Each of the six colours has a blank text file where the numbers will be stored. The names of these six text files are:

- Blue.txt
- Green.txt
- Orange.txt

Question 2

- Pink.txt
- Red.txt
- Yellow.txt

The procedure `SplitData()` needs amending to call `StoreData()` six times, with each of the six colour arrays and the name of the text file that corresponds to that colour. For example, `StoreData()` will be called with the red array and the file name "Red.txt"

Write program code to amend `SplitData()`

Save your program. Copy and paste the program code into part **2(d)** in the evidence document. **[2]**

Question 2

9618/41 J25 ▪ Q2 ▪ 25 marks

A program reads data from a text file, splits the data depending on its content and stores the separated data into different files.

The text file `TheData.txt` contains 72 lines of data. Each line of data has an integer number and a string colour that are separated by a comma. For example, the first line in the file is:

`10,red`

The integer is 10 and the string is `"red"`

The file contains six different colours: red, green, blue, orange, yellow, pink.

(e)(i) Write program code for the main program.

Save your program. Copy and paste the program code into part **2(e)(i)** in the evidence document. **[3]**

(e)(ii) Test your program. Input the text "TheData.txt" when prompted.

Question 2

Take a screenshot of the output(s) and a screenshot showing the content of the file that stores the red numbers.
Save your program. Copy and paste the screenshot(s) into part **2(e)(ii)** in the evidence document. **[2]**

Solution 2

(a) Worked steps

ReadData() asks for a filename, reads that file into an array, and returns it.

```
FUNCTION ReadData() RETURNS ARRAY
    DECLARE DataArray : ARRAY[0:199] OF STRING
    DECLARE FileName, Line : STRING
    DECLARE Count : INTEGER

    OUTPUT "Enter the filename"
    INPUT FileName
    Count <- 0
    TRY
        OPENFILE FileName FOR READ
        WHILE NOT EOF(FileName)
            READFILE FileName, Line
            DataArray[Count] <- Line
            Count <- Count + 1
        ENDWHILE
        CLOSEFILE FileName
    EXCEPT
        OUTPUT "The file could not be opened"
```

Solution 2

Seven marks: function header and end; **prompt and input** for the filename; the file opened for read and **closed** in a sensible place; **looping until EOF**; reading each line; removing the line break and **inserting it in the array**; **returning** the populated array; and **exception handling** with a `TRY`, a handler and an output.

Solution 2

Two details. The prompt is a separate mark from the input, so print one. And the filename is a **variable** — `OPENFILE FileName`, not a literal — which is the whole point of asking the user for it. **Mark scheme**

- 1 mark each to max 7
- ▪ Function header (and end)
- ▪ Prompt to enter filename and reading input
- ▪ Opening the file (to read) and closing the file in an appropriate place
- ▪ Looping until EOF ...
- ▪ ... reading each line in the file ...
- ▪ ... (removing line break and) inserting in array
- ▪ Returning populated array
- ▪ Exception handling try catch with appropriate output
- 7
- Example program code:
- Python def ReadData():
- DataList = []

Solution 2

(b) Worked steps

SplitData() sorts each line into one of six arrays by the colour it names.

```
PROCEDURE SplitData(DataArray : ARRAY)
    DECLARE Red, Green, Blue, Orange, Yellow, Pink : ARRAY[0:199] OF STRING
    DECLARE Index, Comma : INTEGER
    DECLARE Number, Colour : STRING

    FOR Index <- 0 TO 199
        Comma  <- <position of "," in DataArray[Index]>
        Number <- LEFT(DataArray[Index], Comma - 1)
        Colour <- MID(DataArray[Index], Comma + 1,
                      LENGTH(DataArray[Index]) - Comma)
        IF Colour = "Red" THEN
            <append Number to Red>
        ELSE IF Colour = "Green" THEN
            <append Number to Green>
        ...
        ENDIF
    NEXT Index
ENDPROCEDURE
```

Solution 2

Six marks: the procedure header taking the array as a **parameter**; **six arrays declared**, one per colour; a loop through each line; **splitting each line at the comma**; **comparing the colour** to select the right array; and **storing the number** in it.
The line's format is `number,colour`, so the split gives two parts and only the **first** is stored. Storing the whole line puts the colour into a file named after that colour — which is the mistake the "1st value/integer in correct array" mark point exists to catch.

Solution 2

Six comparisons is the straightforward way; a `CASE OF Colour` is neater and scores the same. **Mark scheme**

- 1 mark each
- ▪ Procedure header (and end) taking (1D array) DataArray (of strings) as a parameter
- ▪ Declaration/use of 6 1D arrays (equivalent), one for each colour
- ▪ Looping through each line in parameter DataArray ...
- ▪ ... splitting by comma
- ▪ Comparing 2nd value/colour to each colour to select array ...
- ▪ ... storing 1st value/integer in correct array
- 6
- Example program code:
- Python def SplitData(DataArray):
- Red = []
- Green = []
- Blue = []
- Orange = []

Solution 2

(c) Worked steps

`StoreData()` appends one array to one file.

```
PROCEDURE StoreData(DataToStore : ARRAY, FileName : STRING)
    DECLARE Index : INTEGER
    TRY
        OPENFILE FileName FOR APPEND
        FOR Index <- 0 TO <last used index>
            WRITEFILE FileName, DataToStore[Index]
        NEXT Index
        CLOSEFILE FileName
    EXCEPT
        OUTPUT "The file could not be written"
    ENDTRY
ENDPROCEDURE
```

Solution 2

Five marks: the header taking **an array and a filename** as parameters, with the file opened to **append** and closed in a suitable place; a loop through the array; **writing each item**; **a line break between items**; and **exception handling** with a suitable output.

`FOR APPEND`, not `FOR WRITE`. Opening for write **empties the file first**, so the six calls in part (d) would each leave only their own data behind — and the first five colours would vanish without any error being reported.

Solution 2

`WRITEFILE` adds the line break in pseudocode; in a real language check whether the write function does, and add `\n` if not — that is a mark of its own. **Mark scheme**

- 1 mark each
- ▪ Procedure header taking (1D) array and filename as parameters, opening file to append and closing file (in appropriate place)
- ▪ Looping through each item in array parameter ...
- ▪ ... writing to the file
- ▪ ... with new line break between each line
- ▪ Using exception handling try and catch with suitable output
- 5
- Example program code:
- Python def StoreData(DataToStore, FileName):
- try:
- File = open(FileName,"a+") for Item in DataToStore:
- File.write(Item)

Solution 2

(d) Worked steps

```
CALL StoreData(Red,    "Red.txt")
CALL StoreData(Green,  "Green.txt")
CALL StoreData(Blue,   "Blue.txt")
CALL StoreData(Orange, "Orange.txt")
CALL StoreData(Yellow, "Yellow.txt")
CALL StoreData(Pink,   "Pink.txt")
```

Solution 2

Two marks: one for a correct call with an array and its filename, one for the remaining five.

Solution 2

Match each array to the **file named after the same colour**. Both arguments come from part (b) and part (c) respectively, and the whole design exists so that one procedure can serve all six. **Mark scheme**

- 1 mark each
- ▪ Calling StoreData with one array and filename
- ▪ Calling StoreData with remaining 5 arrays and filename
- 2
- Example program code:
- Python
- StoreData(Red, "Red.txt")
- StoreData(Green, "Green.txt")
- StoreData(Blue, "Blue.txt")
- StoreData(Orange, "Orange.txt")
- StoreData(Yellow, "Yellow.txt")
- StoreData(Pink, "Pink.txt")
- VB.NET

Solution 2

(e)(i) Worked steps

```
DataFromFile <- ReadData()
CALL SplitData(DataFromFile)
```

Three marks: calling `ReadData()`; **storing or using** its return value; and calling `SplitData()` with that array as a parameter.

Solution 2

`ReadData()` is a **function** and `SplitData()` is a **procedure**, so the first result must be captured and the second must not be assigned to anything. Getting that the wrong way round is what the two separate marks for "calling" and "storing the return value" are checking. **Mark scheme**

- 1 mark each
- ▪ Calling ReadData() ...
- ▪ ... and storing/using return value
- ▪ Calling SplitData() with returned array as a parameter
- 3
- Example program code:
- Python
- DataFromFile = ReadData()
- SplitData(DataFromFile)
- VB.NET
- Sub Main(args As String())
- Dim DataFromFile(,) As String = ReadData()

Solution 2

(e)(ii)

Mark scheme

- 1 mark screenshots showing
- ▪ Prompt and input of filename TheData.txt
- ▪ Screenshot of data in red file. Filename must be shown in same screenshot as data

Question 3

9618/42 N24 ▪ Q3 ▪ 17 marks

3 The text file `HighScoreTable.txt` stores the top seven player scores for a game.

The data in the file is stored in the order:

Player ID

Game level

Score

Each item of data is stored on a new line. For example, the first set of data in the file is:

Player ID: GHEH

Game level: 3

Question 3

Score: 10

(a) The data about the players and their scores is stored in a 2D array of strings with the identifier `HighScores`.

The first dimension of the array has seven elements: one for each player. The second dimension of the array has three elements: one for the player ID, one for the game level and one for the score. All data is stored in the array as strings.

`HighScores` is declared local to the main program, and all elements are initialised to an empty string, for example `""`.

Write program code to declare and initialise `HighScores`.

Question 3

Save your program as **Question3_N24**.

Copy and paste the program code into part **3(a)** in the evidence document.

[2]

(b) The function `ReadData()` reads the data from `HighScoreTable.txt` and stores the data in a 2D array. The function returns the 2D array.

The function uses exception handling when opening and reading data from the file.

Write program code for `ReadData()`.

Question 3

Save your program.

Copy and paste the program code into part **3(b)** in the evidence document.

[5]

(c) The procedure `OutputHighScores()` takes a 2D array as a parameter. It outputs each player's ID, level and score in the order they are in the array. The outputs are in the format:

GHEH reached level 3 with a score of 10

Write program code for `OutputHighScores()`.

Question 3

Save your program.

Copy and paste the program code into part **3(c)** in the evidence document.

[2]

(d) The scores in `HighScoreTable.txt` are **not** in order.

The player scores are first sorted by the game level they reached. For example, all players that reached level 5 are higher in the high score table than players that reached level 4.

The players that reached the same level are then sorted in descending order of score.

An example sorted high score table is:

Question 3

Position	Player ID	Game level	Score
1	GFED	5	25
2	HJKM	4	21
3	RERR	4	19
4	TTYU	4	15
5	WSVG	3	20
6	PPTR	3	15
7	SNQT	2	10

The function `SortScores()` uses a bubble sort to sort the data as described and returns the sorted array.

Question 3

Write program code for `SortScores()`.

Save your program.

Copy and paste the program code into part **3(d)** in the evidence document.

[4]

(e) (i) Amend the main program to implement these steps in the order given:

- call `ReadData()` and store the returned array in `HighScores`
- output `"Before"`
- call `OutputHighScores()` with `HighScores` as a parameter
- call `SortScores()` and store the returned array in `HighScores`
- output `"After"`
- call `OutputHighScores()` with `HighScores` as a parameter.

Question 3

Save your program.

Copy and paste the program code into part **3(e)(i)** in the evidence document.

[2]

(ii) Test your program

Take a screenshot of the output.

Save your program.

Copy and paste the screenshot into part **3(e)(ii)** in the evidence document.

[2]

Solution 3

(a) Worked steps

Seven players, three data items each, so the array is **two-dimensional**: seven rows of three.

```
DECLARE HighScores : ARRAY[0:6, 0:2] OF STRING
FOR Row <- 0 TO 6
    FOR Col <- 0 TO 2
        HighScores[Row, Col] <- ""
    NEXT Col
NEXT Row
```

Solution 3

Two marks: the array declared with **at least 7 × 3 elements of type STRING**, and **every element initialised to the empty string**.
Everything is STRING because the file is text and the values are read as text. Declaring the score column as INTEGER means converting on the way in, which the mark scheme does not ask for and which makes the sort in part (d) harder rather than easier.

Solution 3

The initialisation needs a nested loop — one loop sets a row, not a cell. **Mark scheme**

- 1 mark each
- ▪
- HighScores created as 2D array, (of strings) with (min) 7 × 3 elements (local to main) ...
- ▪
- ... all elements initialised to empty string ("")
- 2
- e.g.
- Python
- HighScores = [] #String, 7 x 3
- HighScores = [[" for x in range(3)] for y in range(7)]
- VB.NET
- Dim HighScores(7, 3) As String
- For(X = 0 to 7)
- For(Y = 0 to 3)

Solution 3

(b) Worked steps

`ReadData()` is a **function**: it builds the array and returns it.

```
FUNCTION ReadData() RETURNS ARRAY
    DECLARE HighScores : ARRAY[0:6, 0:2] OF STRING
    DECLARE Row : INTEGER
    TRY
        OPENFILE "HighScoreTable.txt" FOR READ
        FOR Row <- 0 TO 6
            READFILE "HighScoreTable.txt", HighScores[Row, 0]
            READFILE "HighScoreTable.txt", HighScores[Row, 1]
            READFILE "HighScoreTable.txt", HighScores[Row, 2]
        NEXT Row
        CLOSEFILE "HighScoreTable.txt"
    EXCEPT
        OUTPUT "The high score file could not be read"
    ENDTRY
    RETURN HighScores
ENDFUNCTION
```

Solution 3

Five marks:

1. Function header and end, **returning the populated array**.
2. The file opened for reading and **closed** in the right place.
3. A loop over the seven players — or to end of file, or 21 times.
4. Each **group of three** lines read and stored in the three columns of one player's row.
5. Exception handling with **all** the file access inside the `TRY`, a matching handler, and an output.

Solution 3

The structure of the file is the whole design: the values are stored one per line in the order ID, level, score, so three reads fill one row. A single loop of 21 reads works too, but then the row and column have to be computed from the counter — `Row = Index DIV 3` and `Col = Index MOD 3` — and that is easier to get wrong. **Mark scheme**

- 1 mark each
- ▪
- Function header (and end where appropriate) that returns populated array
- ▪
- Opening text file to read and closing the file in an appropriate place
- ▪
- Looping through 7 players/to EOF/21 times ...
- ▪
- ... reading in each group of 3 data items and storing each in separate element in 2D array for each player
- ▪
- Exception handling with all file handling within the try, appropriate catch and an output
- 5

Solution 3

(c) Worked steps

`OutputHighScores` is a **procedure**: it prints and returns nothing.

```
PROCEDURE OutputHighScores(HighScores : ARRAY)
    DECLARE Row : INTEGER
    FOR Row <- 0 TO 6
        OUTPUT HighScores[Row, 0], " reached level ", HighScores[Row, 1],
               " with a score of ", HighScores[Row, 2]
    NEXT Row
ENDPROCEDURE
```

Solution 3

Two marks: the procedure taking the array as a **parameter** and looping over its first dimension, and outputting all the data in the format the question specifies.
Pass the array in rather than reading a global. The question calls this procedure twice in part (e) — once before sorting and once after — and a version that reads a global cannot show two different states.

Solution 3

Loop over the **first** dimension only. The three columns of a row belong on one output line, so they are named individually inside the loop, not iterated over. **Mark scheme**

- 1 mark each
- ▪
- Procedure (header and end) taking (min) 1 parameter (2D array), looping through each of the first dimension in array
- ...
- ▪
- ... outputting all data in correct format
- 2
- e.g.
- Python
- def OutputHighScores(HighScores):
- for x in range(0, len(HighScores)):
- print(HighScores[x][0], "reached level", HighScores[x][1], "with a score of",

Solution 3

(d) Worked steps

A two-key sort: **level first, then score** when the levels are equal.

```
FUNCTION SortScores(HighScores : ARRAY) RETURNS ARRAY
    DECLARE Outer, Inner, Col : INTEGER
    DECLARE Temp : STRING
    FOR Outer <- 0 TO 5
        FOR Inner <- 0 TO 5 - Outer
            IF (HighScores[Inner, 1] < HighScores[Inner + 1, 1])
               OR (HighScores[Inner, 1] = HighScores[Inner + 1, 1]
                   AND HighScores[Inner, 2] < HighScores[Inner + 1, 2]) THEN
                FOR Col <- 0 TO 2
                    Temp <- HighScores[Inner, Col]
                    HighScores[Inner, Col] <- HighScores[Inner + 1, Col]
                    HighScores[Inner + 1, Col] <- Temp
                NEXT Col
            ENDIF
        NEXT Inner
    NEXT Outer
    RETURN HighScores
ENDFUNCTION
```

Solution 3

Four marks: the header and end returning the sorted array; comparing **levels** and swapping; comparing **scores when the levels are equal** and swapping; and loops that put the whole table in order.

The one thing that must not be got wrong: **swap the entire row, all three columns**. Swapping only the sort key detaches every player from their own data, and the result is a table of plausible numbers that is completely wrong — a failure no compiler reports.

Solution 3

The nested-loop bounds are the standard bubble sort: the outer loop runs one fewer than the number of rows, and the inner one shrinks by the outer's count because the tail is already in place. **Mark scheme**

- 1 mark each
- ▪
- Function header (and end taking array as parameter) returning sorted array.
- ▪
- Comparing the levels and swapping all dimensions when in incorrect order
- ▪
- Comparing scores when levels are the same and swapping all dimensions when in incorrect order
- ▪
- Correct loops and comparisons to put data in correct order
- 4
- e.g.
- Python
- def SortScores(HighScores):

Solution 3

(e)(i) Worked steps

The main program calls the three modules in order, showing the table before and after.

```
HighScores <- ReadData()
OUTPUT "Before"
CALL OutputHighScores(HighScores)
HighScores <- SortScores(HighScores)
OUTPUT "After"
CALL OutputHighScores(HighScores)
```

Solution 3

Two marks: the statements calling `ReadData()` then `SortScores()` **in that order and storing what each returns**, and the two labelled outputs with `OutputHighScores` called on the unsorted and the sorted array.

Solution 3

The assignment is what makes this work. `SortScores(HighScores)` on its own throws the sorted array away in a language that passes arrays by value, and the "After" listing then prints the same order as "Before" — which is exactly the output the last mark is checking. **Mark scheme**

- 1 mark each
- ▪
- Code statements in order:
- HighScores = ReadData()
- HighScores = SortScores(HighScores) // HighScores = SortScores()
- ▪
- Code statements in order:
- OUTPUT "Before"
- OutputHighScores(HighScores) unsorted
- OUTPUT "After"
- OutputHighScores(HighScores) sorted
- 2

Solution 3

(e)(ii)

Mark scheme

- Output showing 'Before' and players and scores in correct format before sorting
- Output showing 'After' players and scores in correct order and format after sorting
- 2
- e.g.