A-LEVEL COMPUTER SCIENCE • EXERCISE SHEET

20.2 File Processing and Exception Handling

Name: _____ Class: _____ Date: _____ **Total: 15 marks**

Objective

Build the skills to answer exam questions on **file processing** and **exception handling** 异常处理.

You must be able to:

- write pseudocode for **file** read, write, append and search
- explain what an **exception** is and why **handling** it matters
- write a `TRY ⋯EXCEPT` block and **raise** an exception

1 Worked examples

Study these first. Each one shows the method for a question type used later —follow the steps and you can do the Practice and Exam-style questions yourself.

File processing

`OPENFILE ⋯FOR READ | WRITE | APPEND` (WRITE overwrites; APPEND adds to the end), `READFILE`, `WRITEFILE`, `CLOSEFILE`, `EOF`. Search a file, stopping when found:

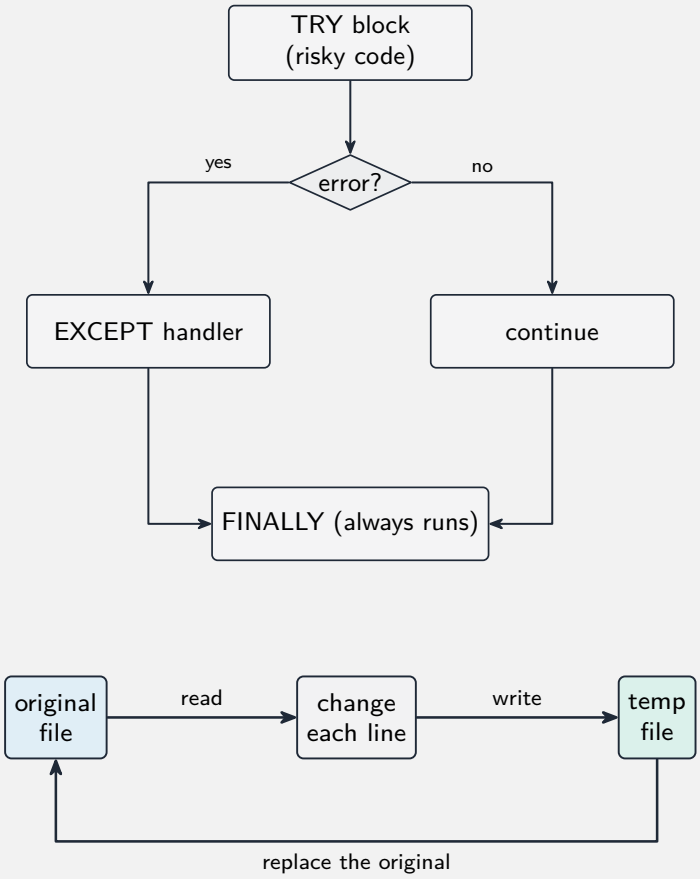
```
found ← FALSE
OPENFILE "people.txt" FOR READ
WHILE NOT EOF("people.txt") AND NOT found DO
    READFILE "people.txt", Line
    IF Line = Target THEN found ← TRUE
ENDWHILE
CLOSEFILE "people.txt"
```

To **edit in place**, copy to a temp file (changing the right lines), then replace the original.

Random (direct-access) files store fixed-size **records** you can jump straight to by position: `OPENFILE ⋯FOR RANDOM`, then `SEEK` to a record number and `GETRECORD` / `PUTRECORD` to read or write that record —without reading the whole file.

Exception handling

An **exception** is an error during execution (file not found, divide by zero). Handling it lets the program respond **gracefully** instead of crashing:



Updating a file record with exception handling

```
TRY
    OPENFILE "data.txt" FOR READ
    READFILE "data.txt", Line
    CLOSEFILE "data.txt"
EXCEPT FileNotFound
    OUTPUT "Sorry, the file does not exist."
ENDTRY
```

A subroutine can **raise** an exception for the caller to handle: `IF b = 0 THEN RAISE DivideByZero`.

2 Practice

Now apply the methods above.

2.1 State what `OPENFILE …FOR APPEND` does that `FOR WRITE` does not. [1]

2.2 State what an **exception** is. [1]

2.3 State the purpose of a `FINALLY` block. [1]

2.4 State **one** pitfall of file processing. [1]

3 Exam-style questions

3.1 Write pseudocode that searches `"people.txt"` for a `Target` name, **stopping** as soon as it is found, and outputs whether it was found. [4]

3.2 Write a `TRY` ⋯ `EXCEPT` block that opens and reads `"data.txt"` and handles the case where the **file does not exist**. [3]

3.3 Explain **why** exception handling is important in real programs. [2]

3.4 Write pseudocode for a function `Divide(a, b)` that **raises** an exception when `b` is

`0`, otherwise returns `a DIV b`. [2]

3.5 A file of fixed-length records is opened `FOR RANDOM`. State what `SEEK` and `PUTRECORD` are each used for. [2]

4 Go further

You are now ready for the real exam questions on this subtopic. Open the **20.2 File Processing and Exception Handling** past-paper sheet in the Library, or try these in **Practice mode**:

- 9618/31 N25 —Q10 (exception handling)
- 9618/33 N25 —Q12 (file processing and exceptions)
- 9618/41 J25 —Q2 (file processing with records)
- 9618/32 N25 —Q10 (causes of exceptions)

Solutions

2.1 `APPEND` **keeps** the existing contents and adds after them; `WRITE` **overwrites** (erases) the file.

2.2 An **error or unexpected condition** that occurs **during execution** (e.g. file not found, divide by zero).

2.3 Code that **always runs** (whether or not an exception happened) —used for **cleanup** such as closing files.

2.4 Any one: forgetting to **close** a file (data lost); opening `FOR WRITE` instead of `APPEND` (overwrites); reading past `EOF`; hard-coded paths.

3.1

```
found ← FALSE
OPENFILE "people.txt" FOR READ
WHILE NOT EOF("people.txt") AND NOT found DO
    READFILE "people.txt", Line
    IF Line = Target THEN found ← TRUE
ENDWHILE
CLOSEFILE "people.txt"
IF found THEN OUTPUT "Found" ELSE OUTPUT "Not found"
```

3.2

```
TRY
    OPENFILE "data.txt" FOR READ
    READFILE "data.txt", Line
    CLOSEFILE "data.txt"
EXCEPT FileNotFound
    OUTPUT "Sorry, the file does not exist."
ENDTRY
```

3.3 Programs face errors that **cannot be prevented up front** (missing files, bad input); handling them lets the program **recover/inform the user** instead of **crashing**, and keeps the normal code clean.

3.4

```
FUNCTION Divide(a : INTEGER, b : INTEGER) RETURNS INTEGER
    IF b = 0 THEN
        RAISE DivideByZero
    ENDIF
    RETURN a DIV b
ENDFUNCTION
```

3.5 `SEEK` moves to a given **record number / position** in the file; `PUTRECORD` **writes a record** at that position.