Week 55

A-Level Computer Science

Homework bundle for this week

本周作业合集

exercises.lol

Contents 目录

20. Further Programming

Starter Quiz · 课前小测

A-Level Computer Science Name: _____ Score: _____

1. Imperative (procedural) programming is based on:

   ☐ a sequence of commands that change the program's state

   ☐ stating facts and rules for an engine to infer

   ☐ objects that combine data and methods

   ☐ pure functions with no side effects

2. An object combines:

   ☐ attributes (data) and methods (operations)

   ☐ only data

   ☐ only functions

   ☐ two separate programs

3. What does opening a file FOR WRITE do to existing contents?

   ☐ it overwrites them (the file is created/emptied)

   ☐ it adds to the end

   ☐ it leaves them and reads only

   ☐ it encrypts them

4. An exception is:

   ☐ an error or unexpected condition that occurs while a program runs

   ☐ a syntax mistake caught at compile time

   ☐ a comment in the code

   ☐ a type of variable

5. Declarative programming means you specify:

   ☐ what to compute, leaving the runtime to work out how

   ☐ every step the machine must take

   ☐ the exact registers to use

   ☐ the order of machine instructions

6. Declarative paradigms (functional, logic, SQL) state WHAT to compute and let the runtime decide how, whereas imperative code spells out every step.

   ☐ True ☐ False

7. Inheritance models an ”is-a” relationship (a Manager is an Employee), so a subclass inherits and extends its superclass.

   ☐ True ☐ False

8. Forgetting to close a file can lose buffered writes and lock other programs out — so you should always close a file when finished.

   ☐ True ☐ False

9. Silently ”swallowing” an exception (catching it but doing nothing) hides real errors and makes debugging hard — you should handle it or at least log it.

   ☐ True ☐ False

10. A constructor is a special method that:

    ☐ runs when an object is created, to set up its attributes

    ☐ deletes an object

    ☐ copies a file

    ☐ runs every loop

A-Level Computer Science

1. **a sequence of commands that change the program's state**

   Imperative code gives step-by-step commands (assignments, loops, calls) that change state.

2. **attributes (data) and methods (operations)**

   An object bundles its data (attributes) with the operations (methods) that act on it.

3. **it overwrites them (the file is created/emptied)**

   WRITE creates or overwrites the file. To keep existing contents and add more, use APPEND.

4. **an error or unexpected condition that occurs while a program runs**

   Exceptions are run-time problems (divide by zero, file not found) that handling lets you respond to gracefully.

5. **what to compute, leaving the runtime to work out how**

   Declarative code (functional, logic, SQL) states the goal; the runtime decides the steps.

6. **True**

   A SQL query says which rows you want, not how to scan the tables — the opposite of step-by-step imperative code.

7. **True**

   Inheritance is "is-a" specialisation; "has-a" relationships are modelled by composition instead.

8. **True**

   Closing flushes buffered data to disk and releases the lock — a FINALLY block is a good place to guarantee it.

9. **True**

   An empty handler hides the very problems you need to find; always respond or record the error.

10. **runs when an object is created, to set up its attributes**

    The constructor initialises a new object's attributes at creation time.

A-LEVEL COMPUTER SCIENCE • EXERCISE SHEET

20.1 Programming Paradigms

Name: _____ Class: _____ Date: _____ **Total: 18 marks**

KEY WORDS attributes 属性 • methods 方法 • objects 对象 • inheritance 继承 • polymorphism 多态

Objective

Build the skills to answer exam questions on **programming paradigms** 编程范式 — low-level, imperative, object-oriented and declarative.

You must be able to:

- explain what a **paradigm** is and the four kinds
- describe the **OOP** features (encapsulation, inheritance, polymorphism, abstraction)
- give examples of **declarative** (functional, logic, SQL) programming

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.

■ The four paradigms

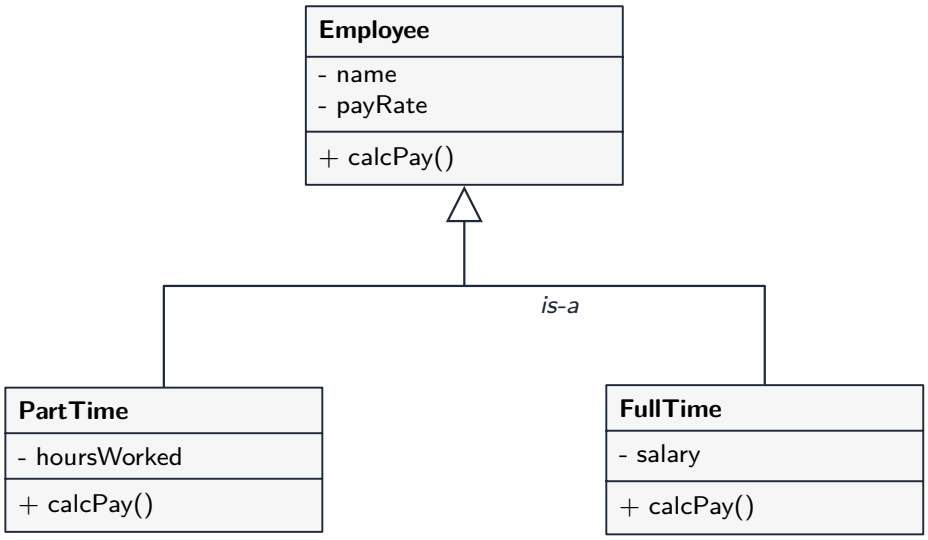
A **paradigm** is a style of structuring programs.

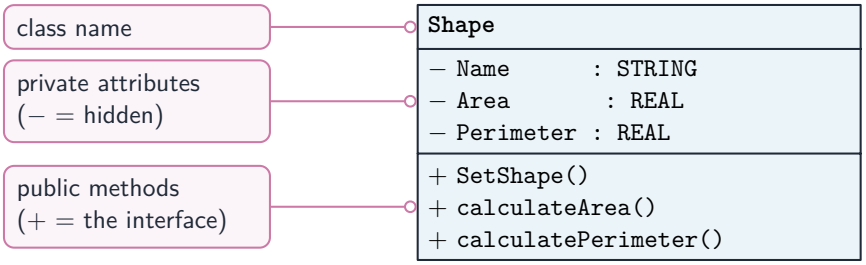
- **Low-level** — machine/assembly code, close to the hardware (fast, but architecture-specific).
- **Imperative (procedural)** — a **sequence of commands** that change state (C, Python).
- **Object-oriented (OOP)** — programs built from **objects** (data + methods).
- **Declarative** — say **what**, not **how** (SQL, Prolog, Haskell).

■ OOP features

An **object** is an **instance** of a **class** (data = **attributes**, operations = **methods**). The pillars:

- **encapsulation** — data is **hidden** behind methods (outside code uses public methods only).
- **inheritance** — a **subclass** specialises a **superclass** (an "is-a" relationship):





A class diagram models an object-oriented design

- **polymorphism** — the **same method call** behaves differently per object type (each `Shape` has its own `Area()`).
- **abstraction** — show a simple interface, hide the implementation.

Writing a class (pseudocode)

A class declares **private** attributes and **public** methods, including a **constructor** (`NEW`) that sets up a new object; a subclass uses **INHERITS**:

```
CLASS Employee
    PRIVATE Name : STRING
    PRIVATE PayRate : REAL

    PUBLIC PROCEDURE NEW(GivenName : STRING, GivenRate : REAL)
        Name ← GivenName
        PayRate ← GivenRate
    ENDPROCEDURE

    PUBLIC FUNCTION GetName() RETURNS STRING
        RETURN Name      // a "get" method reads a private attribute
    ENDFUNCTION
ENDCLASS

CLASS Manager INHERITS Employee
    PRIVATE Bonus : REAL
ENDCLASS
```

Declarative

Functional uses **pure functions** (no side effects); **logic** states facts/rules (Prolog); **SQL** queries data — `SELECT * FROM Customer WHERE Country = 'UK'` says **what**, not **how**.

2 Practice

Now apply the methods above.

2.1 State what a **programming paradigm** is. [1]

2.2 Name the **four** features (pillars) of **OOP**. [2]

2.3 State what **encapsulation** means. [1]

2.4 Give **one** example of a **declarative** language. [1]

3 Exam-style questions

3.1 Describe what is meant by the OOP feature **inheritance**, with an **example**. [3]

3.2 Identify the OOP feature that **restricts external access to an object's data**, and state **one** benefit of it. [2]

3.3 Explain what is meant by **polymorphism**. [2]

3.4 State **one** characteristic of **low-level** programming and **one** of **declarative** programming. [2]

3.5 A class `Book` has a private attribute `Title` of type `STRING`. Write pseudocode for a **constructor** that sets `Title`, and a **get method** `GetTitle()` that returns it. [4]

Solutions

2.1 A **style/approach to structuring programs**, with its own concepts and language features.

2.2 Encapsulation, inheritance, polymorphism, abstraction.

2.3 An object's **data is hidden** and accessed **only through its methods** (not directly).

2.4 Any one: **SQL**, **Prolog**, **Haskell**, Lisp, F#.

3.1 A **subclass** inherits the **attributes and methods** of a **superclass** and can add or **override** its own; example: a `Manager` inherits from `Employee` ("is-a").

3.2 Encapsulation; benefit: it **protects the data** from invalid changes / lets the internals change without breaking other code.

3.3 Different object types respond to the **same method call** in their **own way**, so the caller need not know the exact type (e.g. `Circle` and `Rectangle` each implement `Area()`).

3.4 Low-level: any one — close to hardware, direct register/memory access, fast/compact, architecture-specific. Declarative: any one — you state **what** not **how**, no explicit step-by-step control flow.

3.5

```
PUBLIC PROCEDURE NEW(GivenTitle : STRING)
    Title ← GivenTitle
ENDPROCEDURE

PUBLIC FUNCTION GetTitle() RETURNS STRING
    RETURN Title
ENDFUNCTION
```

5 Complete the table by filling in the missing object-oriented programming (OOP) terms and descriptions.

OOP term	Description
.....	A method that accesses the value of a property.
.....	A method that changes the value of a property.
Object	....
Method	....

[4]

3 A program stores data about animals using Object-Oriented Programming (OOP).

The class `Animal` stores the data about animals.

Animal	
`Name : String`	stores the name of the animal
`Sound : String`	stores the sound the animal makes
`Size : Integer`	stores the size of the animal as an integer between 1 (smallest) and 10 (largest)
`Intelligence : Integer`	stores the intelligence of the animal as an integer between 1 (least) and 10 (most)
`Constructor()`	initialises all attributes to its parameter values
`Description()`	returns a string message that contains the data from the attributes

(a) (i) Write program code to declare the class `Animal` and its constructor.

Do **not** declare the other methods.

Use your programming language appropriate constructor.

If you are writing in Python, include attribute declarations using comments.

Save your program as **Question3_J25**.

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

[4]

(ii) The method `Description()` creates and returns a string of the animal's data in the format:

```
"The animal's name is " <Name> ", it makes a " <Sound> ", its size
is " <Size> " and its intelligence level is " <Intelligence>
```

For example:

```
The animal's name is Teddy, it makes a Bark, its size is 4 and its
intelligence level is 6
```

Write program code for `Description()`

Save your program.

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

[3]

(b) The class `Parrot` inherits from the class `Animal`

`Parrot` inherits the attributes from `Animal` and overrides the `Description()` method. The additional attributes and methods in the class `Parrot` are:

Parrot	
`WingSpan : Integer`	the width of the parrot's wings to the nearest cm
`NumberWords : Integer`	the number of words the parrot can speak
`Constructor()`	calls the parent constructor using its parameter values; initialises `WingSpan` and `NumberWords` to its parameter values
`ChangeNumberWords()`	takes an integer parameter and adds this to the number currently in `NumberWords`

(i) Write program code to declare the class `Parrot`, its constructor and the method `ChangeNumberWords()`

Use your programming language appropriate constructor.

If you are writing in Python, include attribute declarations using comments.

Save your program.

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

[4]

(ii) The method `Description()` in the class `Parrot` creates and returns a string of the animal's data in the format:

```
"The animal's name is " <Name> ", it makes a " <Sound> ", its size
is " <Size> " and its intelligence level is " <Intelligence> ". It
has a wingspan of " <WingSpan> "cm and can say " <NumberWords> "
words."
```

For example:

```
The animal's name is Chewie, it makes a Squawk, its size is 1 and
its intelligence level is 10. It has a wingspan of 30cm and can
say 29 words.
```

Write program code for `Description()`

Save your program.

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

[2]

(c) The class `Wolf` inherits from the class `Animal`

`Wolf` inherits the attributes from `Animal` and overrides the `Description()` method. The additional attributes and methods in the class `Wolf` are:

Wolf	
`TerritorySize : Integer`	stores the size of the area where the wolf lives, to the nearest square mile
`Constructor()`	calls the parent constructor using its parameter values; initialises `TerritorySize` to its parameter value
`SetTerritorySize()`	takes an integer parameter and adds this to the number currently in `TerritorySize`

(i) Write program code to declare the class `Wolf`, its constructor and the method `SetTerritorySize()`

Use your programming language appropriate constructor.

If you are writing in Python, include attribute declarations using comments.

Save your program.

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

[4]

(ii) The method `Description()` in the class `Wolf` creates and returns a string of the animal's data in the format:

```
"The animal's name is " <Name> ", it makes a " <Sound> ", its size
is " <Size> " and its intelligence level is " <Intelligence> ".
Its territory is " <TerritorySize> " square miles."
```

For example:

```
The animal's name is Nighteyes, it makes a Howl, its size is 8 and
its intelligence level is 7. Its territory is 100 square miles.
```

Write program code for `Description()`

Save your program.

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

[2]

(d) (i) The main program declares instances of the classes for **three** animals:

- A parrot with the name 'Chewie'; it makes a 'Squawk' sound. Its size is 1, intelligence is 10, wingspan is 30 cm and it can say 29 words.
- A wolf with the name 'Nighteyes'; it makes a 'Howl' sound. Its size is 8, intelligence is 7 and its territory is 100 square miles.
- An animal that is a horse with the name 'Copper'; it makes a 'Neigh' sound. Its size is 10 and its intelligence is 6.

Write program code for the main program.

Save your program.

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

[2]

(ii) The main program also needs to:

- decrease the territory for the wolf Nighteyes by 20 square miles
- increase the number of words the parrot Chewie can say by 2 words
- output the description for all **three** animals.

Write program code to extend the main program.

Save your program.

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

[3]

(iii) Test your program.

Take a screenshot of the output(s).

Save your program.

Copy and paste the screenshot into part **3(d)(iii)** in the evidence document.

[2]

3 A program stores data in a linked list that is designed using Object-Oriented Programming (OOP).

The class `Node` stores data about the nodes.

Node	
`TheData : Integer`	stores the integer data
`NextNode : Node`	stores the next node in the linked list
`Constructor()`	initialises `TheData` to its parameter value, initialises `NextNode` to a null value
`GetData()`	returns `TheData`
`GetNextNode()`	returns `NextNode`
`SetNextNode()`	takes an object of type `Node` as a parameter and stores it in `NextNode`

(a) (i) Write program code to declare the class `Node` and its constructor.

Do **not** declare the other methods.

Use your programming language appropriate constructor.

If you are writing in Python, include attribute declarations using comments.

Save your program as **Question3_J25**.

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

[4]

(ii) Write program code for the **two** get methods.

Save your program.

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

[3]

(iii) The method `SetNextNode()` takes an object of type `Node` as a parameter. The method stores the parameter in the attribute `NextNode`

Write program code for `SetNextNode()`

Save your program.

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

[2]

(b) The class `LinkedList` stores the linked list.

LinkedList	
`HeadNode : Node`	stores the first node in the linked list
`Constructor()`	initialises `HeadNode` to a null value
`InsertNode()`	creates a new node using its integer parameter. Sets this node as the new head node and updates `NextNode`
`RemoveNode()`	finds the first node that contains its integer parameter and removes this node from the linked list
`Traverse()`	concatenates and returns the integer data in each node in the linked list

(i) Write program code to declare the class `LinkedList` and its constructor.

Do **not** declare the other methods.

Use your programming language appropriate constructor.

If you are writing in Python, include attribute declarations using comments.

Save your program.

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

[2]

(ii) The method `InsertNode()`:

- takes an integer as a parameter
- creates a new node with the parameter as the integer data
- uses the new node's method `SetNextNode()` to store the current `HeadNode` as the next node
- replaces `HeadNode` with the current node.

Write program code for `InsertNode()`

Save your program.

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

[4]

(iii) The method `Traverse()` concatenates the integer data from the nodes in the linked list, starting with the node stored in `HeadNode`. The method returns the final string with each integer data separated with a space.

Write program code for `Traverse()`

Save your program.

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

[3]

(iv) The method `RemoveNode()` takes an integer parameter to search for and remove from the linked list.

The method first checks if the linked list is empty. If the linked list is empty the method returns `FALSE`

If the linked list is **not** empty, the integer data in `HeadNode` is compared to the parameter. If it matches the parameter, `HeadNode` is changed to store the next node.

If the parameter does **not** match, the nodes are followed until either:

- the node with matching integer data is found. This node is removed, the appropriate nodes updated and `TRUE` returned
  or
- none of the nodes contain matching integer data to the parameter. No nodes are removed and `FALSE` is returned.

Write program code for `RemoveNode()`

Save your program.

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

[6]

(c) (i) The main program creates a new `LinkedList` object and uses the appropriate method to insert **five** nodes with the following integer data values in the order given:

```
10      20      30      40      50
```

The main program then:

- calls `Traverse()`
- removes the node containing the integer data `30` using `RemoveNode()`
- calls `Traverse()`

Write program code for the main program.

Save your program.

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

[3]

(ii) Test your program.

Take a screenshot of the output(s).

Save your program.

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

[2]

10 Objects and classes form the basic structure of Object-Oriented Programming (OOP).

(a) Outline the structure of a class.

.....

.....

.....

.....

.....

..... [3]

(b) Give **three** differences between an object and a class.

1 .....

.....

.....

2 .....

.....

.....

3 .....

.....

.....

[3]

9 A veterinary surgery wants to create a class for individual pets.
Some of the attributes required in the class are listed in the table.

Attribute	Data type	Description
`PetID`	`STRING`	unique ID assigned at registration
`PetType`	`STRING`	type of pet assigned at registration
`OwnerTelephone`	`STRING`	telephone number of owner assigned at registration
`DateRegistered`	`DATE`	date of registration

(a) State **one** reason why the attributes would be declared as `PRIVATE`.

.....

..... [1]

(b) Complete the class diagram for `Pet`, to include:

- an attribute and data type for the name of the pet
- an attribute and data type for the name of the owner
- a method to create a `Pet` object and set attributes at the time of registration
- a method to assign a pet ID
- a method to assign the date of registration
- a method to return the pet name
- a method to return the owner's telephone number.

Pet	
`PetID`	: `STRING`
`PetType`	: `STRING`
`OwnerTelephone`	: `STRING`
`DateRegistered`	: `DATE`
.....	: .....
.....	: .....
.....	
.....	
.....	
.....	
.....	.....

[5]

A-LEVEL COMPUTER SCIENCE • EXERCISE SHEET

20.2 File Processing and Exception Handling

Name: _____ Class: _____ Date: _____ **Total: 17 marks**

KEY WORDS exception 异常 • exception handling 异常处理 • file 文件 • raise 抛出 • methods 方法

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:

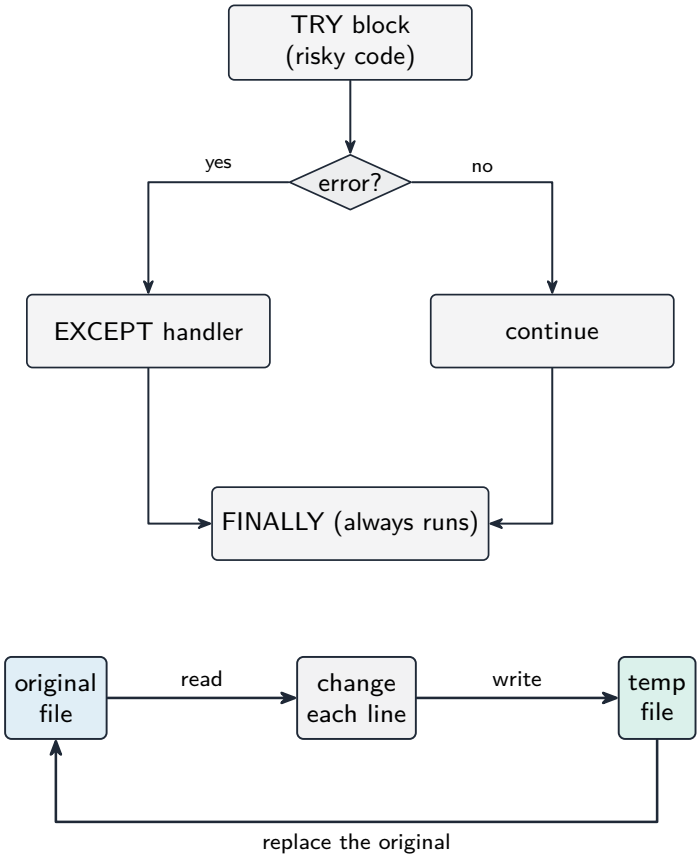
```
found ← FALSE
OPENFILE "people.txt" FOR READ
WHILE NOT EOF("people.txt") AND NOT found DO
    READFILE "people.txt", Line
    IF Line = Target THEN found ← TRUE
ENDWHILE
CLOSEFILE "people.txt"
```

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]

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.

2 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()`:

- 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.

[7]

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

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.

[6]

(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.

Write program code for `StoreData()`

Save your program.

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

[5]

(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`
- `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]

(e) The main program calls `ReadData()` and then `SplitData()`

(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]

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

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]

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

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`.

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()`.

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()`.

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:

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.

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.

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]