

Further Programming

A-Level Computer Science 9618 · Topic 20 revision notes

Everything from the 4 lessons of this topic on a few pages: the key idea of each lesson, the diagram to remember, the points to learn and the traps examiners set. Read it, cover it, test yourself.

1. Low-level, imperative and declarative paradigms

2. Object-oriented programming

3. Serial, sequential and random file processing

4. Exception handling

thelucidstem.com

1 Low-level, imperative and declarative paradigms

Syllabus 20.1

A **programming paradigm** is a style or approach to programming, with its own concepts and way of structuring a solution. **Low-level** programs use the processor's instruction set directly, working with registers and memory addresses through **addressing modes**.

Imperative (procedural) programs are sequences of statements that change the program's state (its variables), organised with sequence, selection, iteration and **procedures and functions**. **Object-oriented** programs are built from **classes** and **objects**. **Declarative** programs state **facts** and **rules**; the programmer gives a **goal** (query) and the language's inference engine searches the facts and rules to satisfy it. Imperative code says *how*; declarative code says *what*.

Four programming paradigms: four ways of telling a computer what to do

Low-level

instructions for one processor:
registers, addresses, addressing modes

```
LDX 300 ADD TOTAL
```

Imperative (procedural)

a sequence of statements that change variables, grouped into procedures

```
Total ← Total + N
```

Object-oriented

classes bundle data with methods;
objects are instances of classes

```
S ← NEW Circle("C1", 2)
```

Declarative

facts and rules describe WHAT is true;
the system finds HOW to meet a goal

```
parent(X, Y) IF ...
```

Imperative and low-level code say HOW, step by step; declarative code says WHAT

Fig. 1: the four paradigms in the 9618 syllabus, each with a typical line of code

Worked example: Trace the low-level loop

Addresses 300 to 303 hold 5, 9, 2 and 7. Trace the low-level program above and state the final contents of address 310.

1. LDR #0 sets IX to 0; LDM #0 then STO 310 and STO 311 set the total and the counter to 0.
2. Pass 1: LDX 300 loads the contents of 300 + 0 = 5. ADD 310 gives 5, stored in 310. IX becomes 1. The counter becomes 1; CMP #4 is false, so JPN jumps back to LOOP.

3. Pass 2: LDX 300 loads the contents of 301 = 9; total 14; IX 2; counter 2; jump back.
4. Pass 3: loads 2 from 302; total 16; IX 3; counter 3; jump back.
5. Pass 4: loads 7 from 303; total 23; IX 4; counter 4. CMP #4 is now true, so JPN does not jump and END returns control to the OS.

Answer: Address 310 holds **23**. The loop ran four times; the low-level version needed a counter in memory because CMP can only compare the accumulator.

How, how, objects, what.

- ✓ **Low-level:** instruction set, registers, addresses; LDM immediate, LDD direct, LDI indirect, LDX indexed, relative = offset from current instruction.
- ✓ **Imperative:** sequence of statements changing variables, structured with procedures and functions.
- ✓ **Declarative:** facts and rules; a goal is satisfied by matching and backtracking. Constants lower case, variables upper case.
- ✓ **Trace goals in clause order** and list every answer that backtracking finds.

Watch out!

Do not describe a paradigm by naming a language. "Prolog" earns nothing; "the program is a set of facts and rules, and a goal is satisfied by searching them" earns the mark.

Variables must start with a capital letter and constants with a lower-case letter. Writing

```
studies(Ana, physics)
```

makes Ana a variable, so the clause would be true for anyone.

2 Object-oriented programming

Syllabus 20.1

A **class** is a blueprint that defines the **attributes** (properties: the data) and **methods** (procedures and functions) of a type of object. An **object** is an **instance** of a class, created by calling its **constructor** (`NEW`). **Encapsulation** keeps attributes **PRIVATE** so they can only be read or changed through **PUBLIC** **getters** and **setters**. **Inheritance** lets a subclass take on the attributes and methods of its superclass and add its own; **polymorphism** lets a subclass **override** an inherited method so the same method call behaves differently for different classes. **Containment (aggregation)** is when an object of one class holds objects of another class as attributes.

A class diagram: inheritance (hollow arrow) and containment (diamond)

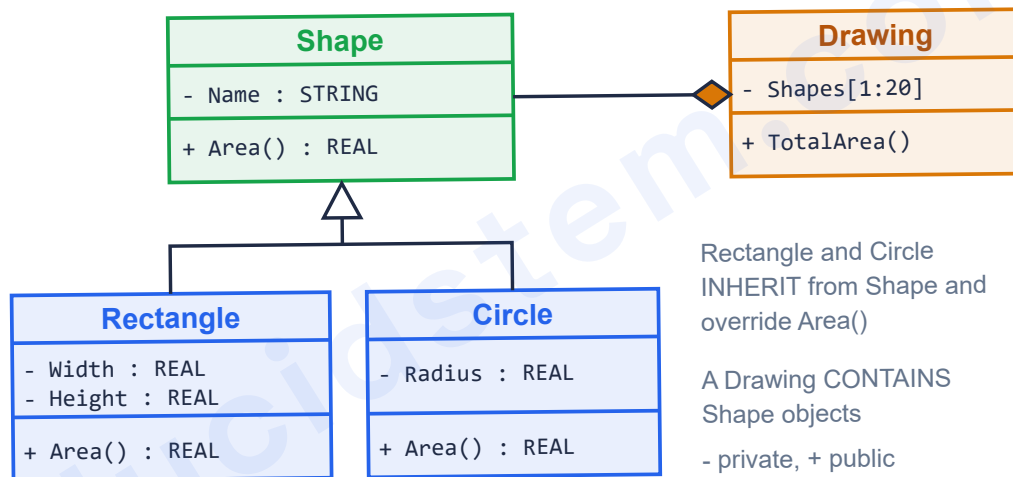


Fig. 1: Rectangle and Circle inherit from Shape (is-a); a Drawing contains Shape objects (has-a)

Worked example: Trace method calls

After the code above, these statements run: `OUTPUT MyAccount.Withdraw(80)`, `OUTPUT MyAccount.Withdraw(20)`, `MyAccount.Deposit(-10)`, `OUTPUT MyAccount.GetBalance()`. What is output?

1. After the constructor and `Deposit(50)`, `Balance = 50`.
2. `Withdraw(80)`: `80 <= 50` is false, so the balance is unchanged and `FALSE` is output.
3. `Withdraw(20)`: `20 > 0` and `20 <= 50`, so `Balance` becomes 30 and `TRUE` is output.
4. `Deposit(-10)`: the amount is not greater than 0, so nothing changes.
5. `GetBalance()` returns 30.

Answer: `FALSE`, `TRUE`, 30. Because `Balance` is **PRIVATE**, the only way to change it is through methods that validate the change: that is the point of encapsulation.

Blueprint, instance, is-a, has-a.

- ✓ **CLASS Name ... ENDClass**; attributes PRIVATE with a data type; methods PUBLIC unless told otherwise.
- ✓ **Constructor** = PUBLIC PROCEDURE NEW(...) that sets every attribute; create objects with $Obj \leftarrow NEW\ Name(...)$.
- ✓ **Subclass**: CLASS Sub INHERITS Super, and call SUPER.NEW(...) first in its constructor.
- ✓ **Is-a** means inheritance; **has-a** means containment; same method name, different behaviour means polymorphism.

Watch out!

A class and an object are not the same thing. The **class** is the definition (written once); an **object** is an instance created from it with NEW, and there can be many objects of one class, each with its own attribute values.

Inheritance and polymorphism are different marks. **Inheritance** is the subclass receiving the superclass's attributes and methods; **polymorphism** is a method with the same name behaving differently in different classes (by overriding).

Do not use inheritance for a has-a relationship. A Drawing is not a kind of Shape, so `CLASS Drawing INHERITS Shape` is wrong; a Drawing **contains** Shapes.

3 Serial, sequential and random file processing

Syllabus 20.2

A file must be opened in a **mode** before use and closed afterwards. **READ** reads lines from the start; **WRITE** creates a new file (any existing data is lost); **APPEND** adds to the end; **RANDOM** opens a file of fixed-length records for direct access. In a **serial** file records are stored in the order they were added; in a **sequential** file they are stored in order of a **key field**, so inserting a record means copying to a new file. In a **random** file each record is at an address, often calculated from its key by a **hashing** function, and **SEEK** moves the file pointer straight there so **GETRECORD** and **PUTRECORD** can read or replace that record.

Serial, sequential and random (direct access) files

Serial

M031	M007	M052	M012	M040
------	------	------	------	------

records in the order they were added; new records go at the end

Sequential

M007	M012	M031	M040	M052
------	------	------	------	------

records stored in order of a key field; inserting means rewriting

Random

0	1	2	3	4
---	---	---	---	---

fixed-length records at numbered slots

↑
SEEK file, 3

Fig. 1: the same five member records stored serially, sequentially by ID, and at numbered addresses in a random file

Worked example: Trace the insertion

Members.txt holds M007Lee, M012Shah, M031Novak, M040Ito and M052Grant, in that order. Trace the code above and give the contents of NewMembers.txt.

1. M007Lee: is "M007" > "M020"? No. Write M007Lee.
2. M012Shah: "M012" > "M020"? No. Write M012Shah.
3. M031Novak: "M031" > "M020"? Yes, and Inserted is FALSE, so write M020Patel, set Inserted to TRUE, then write M031Novak.
4. M040Ito and M052Grant: Inserted is TRUE, so each is simply copied.
5. End of file: Inserted is TRUE, so the final IF does nothing.

Answer: M007Lee, M012Shah, M020Patel, M031Novak, M040Ito, M052Grant. The final IF is needed for a new key larger than every existing key, which must be written at the

end.

Open, loop, close.

- ✓ **Open** in the right mode: `READ`, `WRITE` (new file, old data lost), `APPEND` (add at the end), `RANDOM` (fixed-length records).
- ✓ **Loop** with `WHILE NOT EOF(...)` and one `READFILE` per pass; always `CLOSEFILE` every file you opened.
- ✓ **Sequential insert:** copy to a new file, writing the new record before the first larger key, and handle the end of the file.
- ✓ **Random:** calculate the address, `SEEK`, then `GETRECORD` or `PUTRECORD`; `SEEK` again before writing back.

Watch out!

Use `WHILE NOT EOF(...)`

before each `READFILE`, not a `FOR` loop with a guessed number of lines. `EOF` is also `TRUE` straight away for an empty file, so the loop correctly runs zero times.

You cannot insert a line into the middle of a text file in place. For a sequential file the answer must copy the records to a **new file** (and then, usually, the new file replaces the old one).

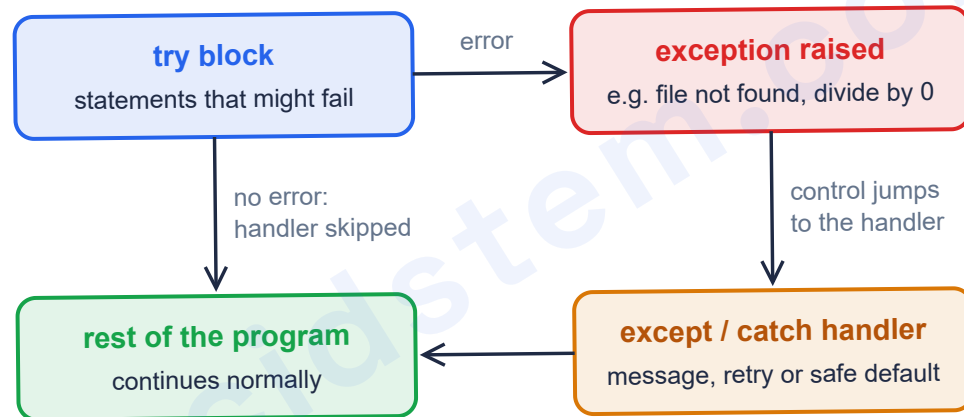
Always `SEEK` before `PUTRECORD` when updating a record you have just read. Re-seeking the same address makes certain the changed record replaces the original rather than being written somewhere else.

4 Exception handling

Syllabus 20.2

An **exception** is an unexpected event or error that happens while a program is **running** (a run-time error) and disrupts the normal flow of instructions, for example a **file not found**, **division by zero**, a **failed type conversion** or an **array index out of range**. If it is not handled, the program **crashes**. **Exception handling** places code that might fail in a **try** block; if an exception is **raised (thrown)**, control jumps to a matching **except / catch** block, which can report the problem, retry or use a safe default, so the program can **continue** or end **gracefully**. An optional **finally** block runs either way, for tidying up such as closing files.

What happens when an exception is raised inside a try block



Without a handler the exception is unhandled and the program crashes with a run-time error

Fig. 1: if nothing goes wrong the handler is skipped; if an exception is raised, control jumps to the handler

Worked example: Trace the handler

Totals.txt holds four lines: 12, 7, abc and 5. What does `read_total("Totals.txt")` output and return? What happens if the file does not exist?

1. The file opens, so the first except block is skipped and total starts at 0.
2. Line 12: int succeeds, total = 12. Line 7: total = 19.
3. Line abc: `int("abc")` raises `ValueError`. The rest of the try block (the addition) is skipped and the handler prints `Skipped a line that is not a whole number: abc`. The loop then carries on.
4. Line 5: total = 24. The file is closed and 24 is returned.
5. If the file does not exist, `open` raises `FileNotFoundError`, the handler prints `Cannot find Totals.txt` and the function returns -1 instead of crashing.

Answer: It outputs one message about abc and returns 24. Without the inner handler, the bad line would have crashed the whole program and the valid numbers would have been lost.

Try, catch, recover.

- ✓ **Define:** an exception is an unexpected event / error that occurs at run time and disrupts normal execution.
- ✓ **Importance:** no crash; meaningful message; files closed and data saved; the program can recover or retry.
- ✓ **Code:** risky statements in try; a handler for each expected exception; finally for tidying up.
- ✓ **Appropriate** when the cause is outside the program (files, devices, networks, conversions); use IF validation for predictable rules.

Watch out!

An exception is a **run-time** event. Exception handling cannot catch a syntax error (the program would not run at all) and does not fix a logic error that simply produces the wrong answer.

Put only the statements that might fail, and the ones that depend on them, inside the try block. Code after the failing statement in the try block is **skipped**, so a total updated after the risky conversion is safe, but one updated before it is not rolled back.

"Exception handling stops the program crashing" is only part of the answer. To earn the marks, say what the handler **does**: outputs a meaningful message, retries, uses a default value, saves data or closes files.