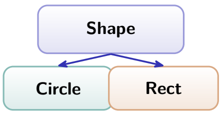
Further Programming

A-Level Computer Science ▪ Topic 20

A-Level Computer Science



What we will cover

1 Paradigms

A paradigm is a style of programming

Four paradigms sit in this syllabus.

low-level 低级
machine code or assembly; registers and addresses; drivers, firmware

imperative 命令式
a sequence of commands that change state – Topics 9 and 11

object-oriented 面向对象
objects bundle data and methods; instances of classes

declarative 声明式
say *what*, not *how* – SQL, Prolog, Haskell

Programming paradigms

Imperative programming 命令式编程 says *how*; **declarative programming** 声明式编程 says *what*, and splits into **functional programming** 函数式编程 and **logic programming** 逻辑编程.

low-level close to the hardware	imperative a sequence of commands
object-oriented objects with data + methods	declarative say what, not how

Low-level and imperative

Low-level programming – **assembly language** 汇编语言 working directly with **memory addresses** 内存地址 – is close to the hardware. Each instruction maps to what the CPU runs. Addressing modes: immediate, direct, indirect, indexed, relative. Fast and compact; architecture-specific and hard to maintain.

Imperative (procedural / structured): assignments, conditionals, loops, function calls. **Variables** 变量 hold state; statements change it. Strong when the algorithm has clear sequential steps. Python, C.

Declarative – what, not how

Two kinds:

- **functional** 函数式 – **pure functions** 纯函数 with no **side effects** 副作用 (Haskell, Lisp)
- **logic** 逻辑 – facts and rules; the engine answers a goal (Prolog)

SQL is the familiar example: `SELECT * FROM Customer WHERE Country = 'UK'` says what you want, not how to walk the records.

> Modern languages mix paradigms. Python supports procedural, OOP and functional. Pick the one that fits the problem.

The paradigms, and what each is for

low-level	maximum control and speed – assembly
imperative	a sequence of statements changing state – most general-purpose languages
object-oriented	objects holding data and behaviour together – large systems, GUIs, simulations and games
declarative	state *what*, not *how* – logic programming
database	data queries – SQL
the point	a paradigm is a style, not a language: many languages support several

> Choose by the problem's shape. A query belongs in SQL, a simulation in objects, a device driver in something low-level – and the mark is for saying why.

2 OOP

The four pillars of object-oriented programming

encapsulation 封装
data is hidden behind methods; a `BankAccount` hides `balance` behind `deposit()`

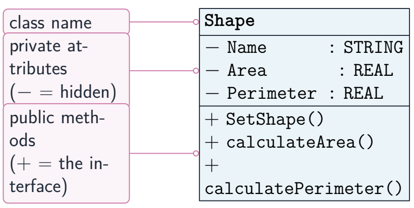
inheritance 继承
a **subclass** 子类 specialises a **superclass** 父类, **overriding** 重写 what it needs. Models is-a.

polymorphism 多态
the same call runs different code: every `Shape` has `Area()`

abstraction 抽象
show a simple interface; hide the implementation

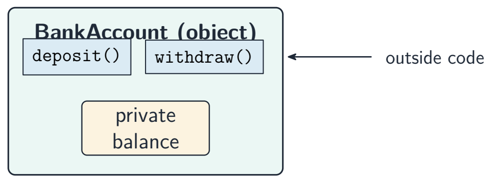
A class diagram

> A class diagram for a Shape: private attributes and public methods



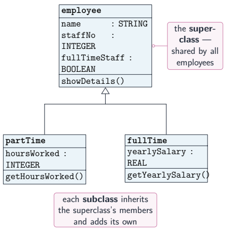
Encapsulation

> Encapsulation: an object's data is private, reached only through its public methods



the data is reached only through the public methods

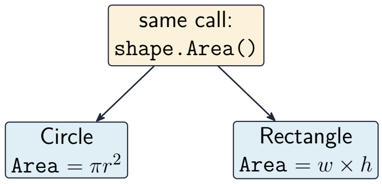
Inheritance: a subclass extends a class



> Inheritance: partTime and fullTime are subclasses of employee

Polymorphism

> Polymorphism: the same method call runs each object's own code



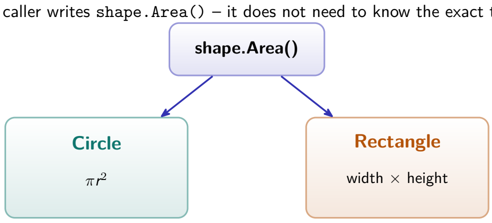
same method name — different code runs

Class vocabulary

class	the template; an object is an **instance** 实例 of it
constructor	a special method that sets up attributes on creation
getters setters	read and write attributes through methods
aggregation	a "has-a"; **containment** 包含 is the stronger form where the part cannot outlive the whole

Polymorphism in one picture

The caller writes `shape.Area()` – it does not need to know the exact type.



3 Files

File operations in pseudocode

Extends Topic 10. Modes: `READ` (existing), `WRITE` (create/overwrite), `APPEND` (add at the end).

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

> Always open, process, then close. Forgetting to close can lose data. `WRITE` when you meant `APPEND` overwrites everything.

You cannot overwrite a line in place

A members file needs one phone number changed. Why not just overwrite that line?

Lines have different lengths. A longer replacement would run into the next record; a shorter one would leave part of the old line behind.

Pattern: open the original for `READ` and a temporary file for `WRITE`; write the new version of the changed line and the original of every other; close both; replace the original with the temp file.

The same shape handles deleting (skip the line) and inserting. Every line gets written – writing only the new record and losing the rest is the classic slip.

4 Exceptions

Handle the errors you cannot prevent

An **exception** 异常 is a runtime error: divide by zero, file not found, index out of range. **Exception handling** 异常处理 lets the program respond instead of crashing.

`TRY` the risky code → `EXCEPT` a named error → `FINALLY` always runs

> A file may be deleted between "exists?" and "open". You cannot prevent that – only handle it. Do not swallow exceptions silently.

Why exception handling exists

the situation	real programs face errors that cannot be prevented up front – files moved, networks down, bad input
without handling	the program crashes, losing unsaved work and telling the user nothing useful
`TRY`	the code that might fail
`EXCEPT`	what to do when it does – retry, use a default, or report clearly
`FINALLY`	cleanup that must run either way – closing a file, releasing a lock
the discipline	catch only what you can actually do something about; let the rest propagate

> Validation prevents the errors you can foresee; exception handling copes with the ones you cannot. Both are needed, and neither replaces the other.

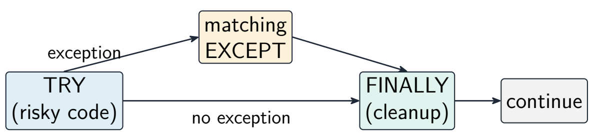
Raising, and where to catch

A subroutine that detects an error can **raise** 抛出 it so the caller handles it: `IF b = 0 THEN`
`RAISE DivideByZero`
`ENDIF` Handle close to the error if the response is simple, or higher up the **call stack** 调用栈 if only the outer code knows what to do.

Common names: `FileNotFound`, `IOError`, `DivisionByZero`, `IndexOutOfRange`, `InvalidArgument`, `NullReference`, `OutOfMemory`.

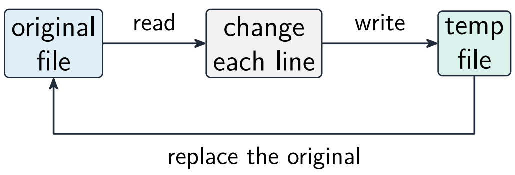
Pattern

> Exception flow: an exception jumps to the matching EXCEPT; FINALLY always runs before the program continues



Updating a file in place

> Updating a file in place: read the original, write changes to a temp file, then replace the original



Pitfalls

Forgetting to close a file (data may be lost); opening for WRITE when you meant APPEND (overwrites everything); reading past `EOF`.

Exam tips

- Distinguish the **paradigms** (procedural, object-oriented, declarative, low-level) and when each suits a problem.
- For OOP, define **class, object, inheritance, encapsulation and polymorphism** with a short example.
- Explain **exception handling** (try/catch) and why it beats letting the program crash.

5 Recap

Topic 20 – key takeaways

1. **Paradigms**
   low-level, imperative, OOP, declarative
2. **OOP**
   encapsulate, inherit, polymorph, abstract
3. **Files**
   open / process / close; update via a temp file
4. **Exceptions**
   TRY / EXCEPT / FINALLY; never swallow

Check yourself

1. Which paradigm says *what* to compute, not *how*?
2. Inheritance models which relationship?
3. Why write an update to a temporary file?
4. Which block always runs, exception or not?

> **Answers** **1.** declarative **2.** is-a **3.** variable-length lines cannot be overwritten in place **4.** FINALLY