Further Programming

A-Level Computer Science

Programming paradigms

A **programming paradigm** 编程范式 is a style of programming —a way of structuring programs, with its own ideas and language features. Four **programming paradigms** are in this syllabus.

Four paradigms: low-level, imperative, object-oriented and declarative

Low-level programming

Programming **close to the hardware** in **machine code** 机器码 or **assembly language** 汇编语言, where each instruction maps to what the CPU runs. It gives direct access to **registers** 寄存器 and **memory addresses** 内存地址, using different **addressing modes** 寻址方式 (immediate, direct, indirect, indexed and relative). It is very fast and compact, but architecture-specific, tedious, and hard to maintain. This is **low-level** 低级 programming, used for device drivers, firmware and bootloaders.

Imperative (procedural) programming

In **imperative programming** 命令式编程 the programmer writes a **sequence of commands** that change the program's state —assignments, conditionals, loops, function calls. **Variables** 变量 hold state; statements change it; code is organised into procedures and functions (also called **structured** or **structural programming**). This is the style of Topics 9 and 11 (Python, C). Strong when the algorithm has clear sequential steps.

Object-oriented programming (OOP)

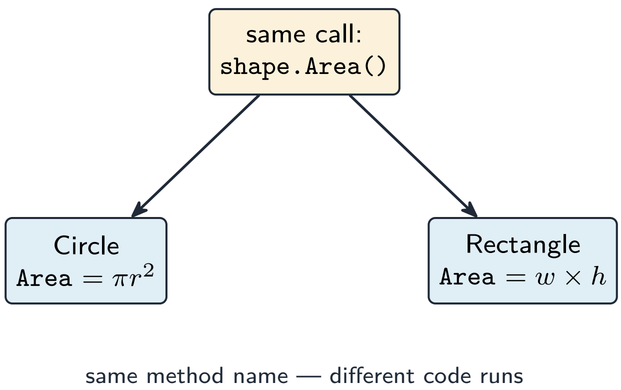
In **object-oriented programming** 面向对象编程 programs are built from **objects** 对象—units combining **data** (**attributes** 属性) and **operations** (**methods** 方法). Objects are **instances** 实例 of **classes** 类. The four pillars:

- **encapsulation** 封装—an object's data is hidden behind its methods; outside code uses the public methods only, not the data directly. This protects the object and lets its internals change without breaking callers. For example, a `BankAccount` hides its `balance`; you change it only through `deposit()` and `withdraw()`, which can enforce a rule like "never go below zero".
- **inheritance** 继承—a **subclass** 子类 specialises a **superclass** 父类, inheriting its attributes and methods and adding or **overriding** 重写 them. Models "is-a" ("a Manager is an Employee").
- **polymorphism** 多态—different objects respond to the **same method call** differently; the caller need not know the exact type. Every `Shape` has `Area()`, and a `Circle` and a `Rectangle` each implement it their own way.
- **abstraction** 抽象—show a simple interface and hide the implementation.

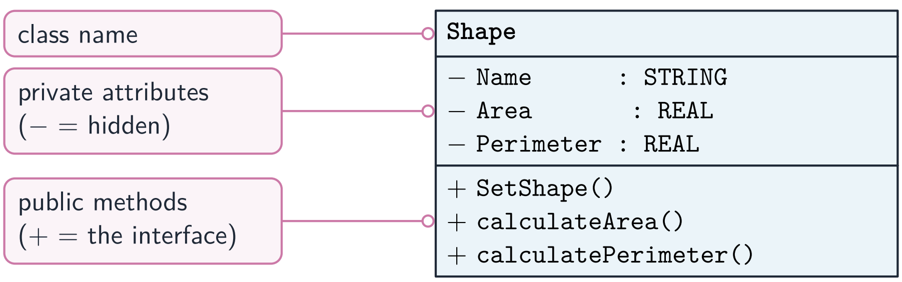
Other terms:

- a **constructor** 构造函数 is a special method run when an object is created, to set up its attributes.
- **getters** and **setters** read and write an object's attributes (its **properties**) through methods.
- **aggregation** 聚合 and **containment** 包含 build an object from other objects (a "has-a" relationship).

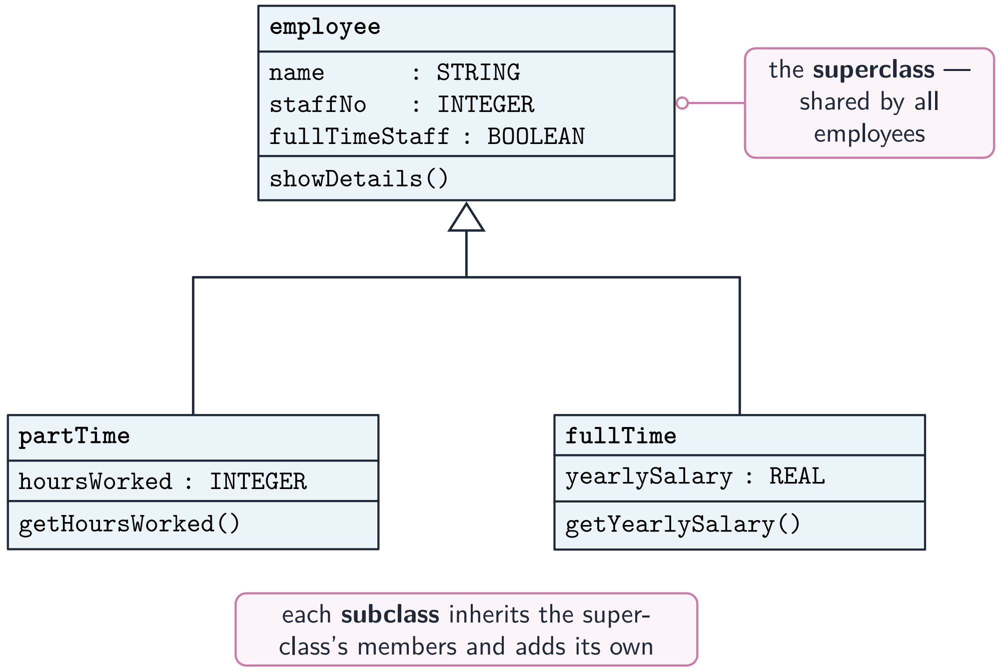
OOP is used for large systems, GUIs, simulations and games.



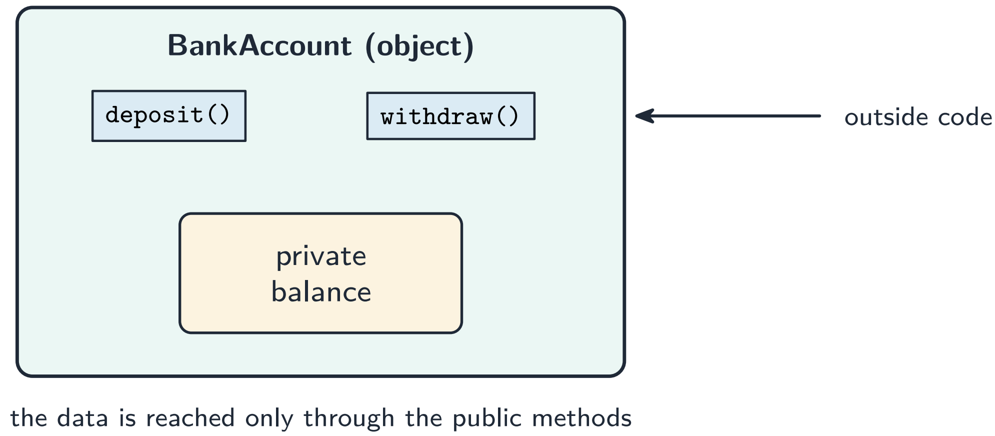
Polymorphism: the same method call runs each object's own code



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



Inheritance: partTime and fullTime are subclasses of employee



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

Declarative programming

In **declarative programming** 声明式编程 you say **what** to compute, not **how** —the runtime works out the steps. Two kinds:

- **functional programming** 函数式编程—built from **pure functions** 纯函数 (no **side effects** 副作用; same input always gives the same output) composed together. Examples: Haskell, Lisp.
- **logic programming** 逻辑编程—state facts and rules; the engine answers a **goal** (query) by inference. Example: Prolog.

A familiar declarative example is **SQL** 结构化查询语言: `SELECT * FROM Customer WHERE Country = 'UK'` says what you want, not how to walk the records.

Comparing paradigms

Paradigm	**Strength**	**Typical languages**
Low-level	maximum control, speed	assembly
Imperative	direct, intuitive	C, Python
Object-oriented	modular, models entities	Java, C#, Python
Functional	clear, no side effects	Haskell, F#
Logic	inference, rules	Prolog
Database	data queries	SQL

Modern languages often **mix paradigms** —Python supports all of procedural, OOP and functional. The right one depends on the problem.

File processing

This extends the **file** 文件 handling from Topic 10, processing **serial**, **sequential** and **random** (direct-access) files. Pseudocode operations: `OPENFILE name FOR READ | WRITE | APPEND` (READ opens an existing file, WRITE creates/overwrites, APPEND adds to the end); `READFILE name, line`; `WRITEFILE name, value`; `CLOSEFILE name`; and `EOF(name)` which is TRUE at the end.

Read a whole file:

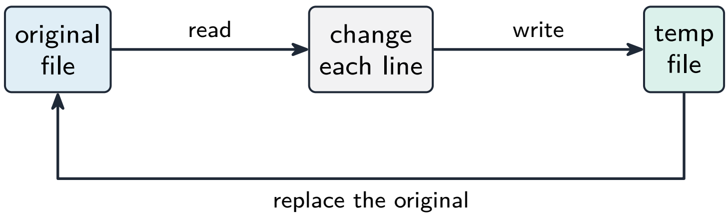
```
OPENFILE "names.txt" FOR READ
WHILE NOT EOF("names.txt") DO
    READFILE "names.txt", thisName
    OUTPUT thisName
ENDWHILE
CLOSEFILE "names.txt"
```

Search a file (stop 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"
```

Updating a file in place

Most languages can't edit a text file in place. Instead: open the original for READ and a temporary file for WRITE; for each line, write the new version if it should change, else the original; close both; then replace the original with the temp file. The same pattern handles deleting lines (skip them) and inserting lines.



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`; hard-coded paths —a path like `/Users/Admin/data.txt` breaks on another machine, so use a relative constant such as `DataFile = "./data/scores.txt"`.

Exception handling

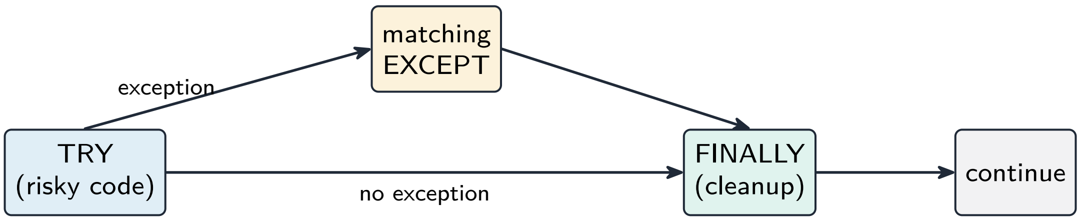
An **exception** 异常 is an error or unexpected condition during execution —divide by zero, file not found, network failure, an **array** 数组 index out of range. **Exception handling** 异常处理 lets a program **detect** it and respond gracefully instead of crashing.

It matters because real programs face errors that **cannot be prevented up front** (files moved, networks down, bad input); without it, every operation needs its own `IF` check; and it **separates** the normal flow from the error handling, so the main path reads cleanly. For example, a file may be deleted by another user between your program checking it exists and actually opening it —you cannot prevent that, only handle the failure when it happens.

Pattern

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

The `TRY` block holds the code that might fail; the first matching `EXCEPT` block runs. Real languages also have a catch-all `EXCEPT` and a `FINALLY` block that runs whether or not an exception happened —useful for cleanup (closing files).



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

Raising an exception

A subroutine that detects an error can **raise 抛出** an exception so the caller handles it:

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

Where to handle exceptions

Handle them **close to the error** if the response is simple (a message, a retry), or **higher up the call stack 调用栈** if only the outer code knows what to do (a top-level GUI loop logs the error and shows a friendly dialog). **Don't swallow exceptions silently** —at least log them, or debugging becomes impossible.

Common exceptions: `FileNotFound`, `IOError`, `DivisionByZero`, `IndexOutOfRange`, `InvalidArgument`, `NullReference`, `OutOfMemory`. Wrapping each failing operation in a `TRY` with the right `EXCEPT` handlers gives a program that **degrades gracefully** instead of crashing.

Worked example. A text file of members needs one member's phone number changed. Why can the program not simply overwrite that line, and what is the pattern? A text file's lines are **different lengths**, and the file has no gaps to absorb a difference: a longer replacement would run into the next record, and a shorter one would leave part of the old line behind. So the pattern is to open the original for **READ** and a **temporary** file for **WRITE**, read every line in turn, writing the **new** version for the line that changes and the **original** line for all the others, close both, then **replace** the original with the temporary file. The same shape handles deleting (skip the line) and inserting (write the extra line). Note that **every** line gets written, not only the changed one - writing just the new record and losing the rest of the file is the classic slip.

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.