

Further programming

A2

Read online at pastopic.com/cambridge/computer-science-9618/notes/further-programming

Last checked 30 September 2026

Read first: [Processor fundamentals](#)

© 2026 Pastopic. Free for personal study. Please share the link, not the file.

What you need to know

This is an A Level topic (Paper 3, with the practical side in Paper 4). By the end of it you should be able to:

1. **Programming paradigms:** say what a paradigm is and describe the four in the syllabus: low-level, imperative (procedural), object-oriented and declarative. Recognise each from a short piece of code.
2. **Low-level programming:** read and write assembly language that uses the five addressing modes: immediate, direct, indirect, indexed and relative.
3. **Imperative (procedural) programming:** the AS work on variables, sequence, selection, iteration, procedures and functions, which this topic assumes you already know.
4. **Object-oriented programming (OOP):** know the terms object, property (attribute), method, class, inheritance, polymorphism, containment (aggregation), encapsulation, getter, setter and instance; design the classes for a problem (a class diagram); and write OOP code.
5. **Declarative programming:** turn given information into facts and rules, and write goals that are satisfied from those facts and rules.
6. **File processing:** open a file for read, write or append, and close it; read and write a record; process serial, sequential and random files.
7. **Exceptions:** say what an exception is, why handling exceptions matters and when to use it, and write code that handles one.

Understand it

Programming paradigms

A **programming paradigm** is a style of programming: a way of thinking about what a program is. Each paradigm has its own features and suits different problems.

Paradigm	What a program is	Tiny example
Low-level	Instructions from the processor's own instruction set (assembly language or machine code), working on registers and memory addresses	LDD 120 then ADD #1

Paradigm	What a program is	Tiny example
Imperative (procedural)	A sequence of statements that say step by step how to reach the result, changing the values of variables (the program's state) as they run, often grouped into procedures and functions	FOR i ← 1 TO 5 ... Total ← Total + i
Object-oriented	Objects made from classes , each object holding its own data and the methods that work on it; uses inheritance, encapsulation and polymorphism	CLASS Lamp ... ENDCLASS
Declarative	Facts and rules ; you state what result you want (a goal) and the language works out how to find it	planet(mars) .

The two most often contrasted are imperative ("do this, then this": the order of the steps matters and the programmer controls it) and declarative ("here is what is true; find answers to this query": the order of the searching is up to the language).

Low-level programming and addressing modes

In assembly language each instruction is an **opcode** (what to do, such as LDD) and usually an **operand**. The **addressing mode** says how the operand is used to find the data. The exam gives a table of the instructions for a processor with one register, the **accumulator (ACC)**, and sometimes an **index register (IX)**.

Suppose memory holds these values and IX holds 3:

Address	60	61	62	63	64
Contents	64	7	11	29	5

Mode	Instruction	What the operand means	ACC afterwards
Immediate	LDM #61	the operand is the value	61
Direct	LDD 61	the operand is the address of the value	7
Indirect	LDI 60	the operand is the address of a location that holds the address of the value: 60 holds 64, and 64 holds 5	5
Indexed	LDX 60	the address is the operand plus the contents of IX : $60 + 3 = 63$	29
Relative	(offset 22 in an instruction at address 40)	the address is an offset from a base address , usually the current instruction's address: $40 + 22 = 62$	11

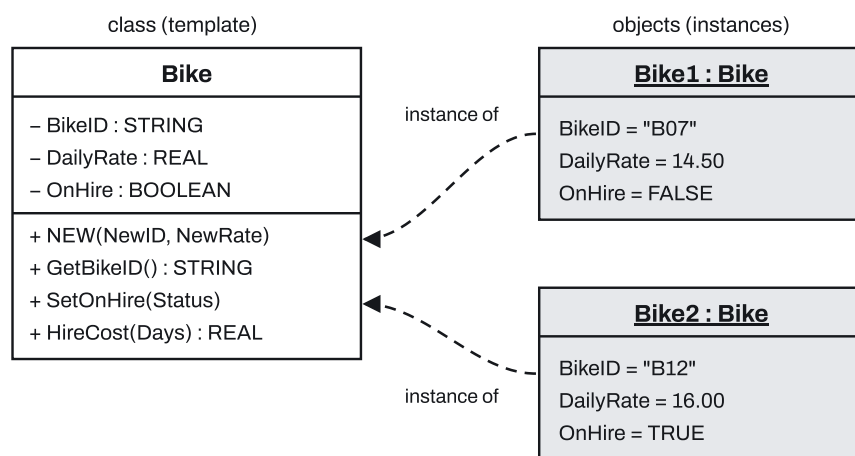
in front of a number marks an immediate (denary) value. Without it, a number is an address. Indexed addressing lets one instruction walk through an array by changing IX; relative addressing lets code run wherever it is loaded in memory.

Labels. Instead of a number, an address can be given a name. The line `Total : 0` puts the value 0 in a memory location and calls that location `Total`; after that, `STO Total` and `ADD Total` use it like a variable. A **named constant** is a labelled location whose value is set once and never changed.

Object-oriented programming

A **class** is a **template** (a blueprint) that defines the **attributes** (also called **properties**: the data, each with a data type) and the **methods** (procedures and functions that act on that data) that its objects will have. An **object** is one **instance** of a class: it is created (**instantiated**) from the class and holds its own values for the attributes. Creating it runs the **constructor**, the method (called `NEW` in pseudocode) that sets up the attributes.

Class	Object
The description: which attributes and methods exist	One actual thing built from that description
Lists the attributes, but stores no values in them	Stores its own value for every attribute (<code>Bike1</code> and <code>Bike2</code> have different IDs)
Written in the code, one <code>CLASS ...</code> <code>ENDCLASS</code> block	Built by the constructor (<code>NEW</code>) while the program runs; there can be any number at once



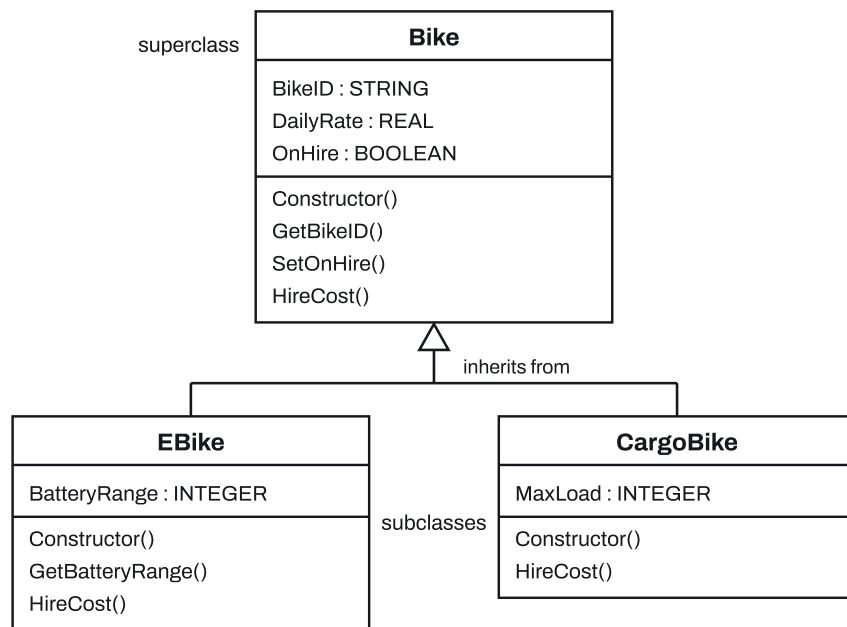
– means private: only the class's own methods can reach it. + means public.

Encapsulation means putting the data and the methods that use it together in one unit, the class, and **hiding** the data: attributes are declared `PRIVATE`, so code outside the class can reach them only through the class's own `PUBLIC` methods. That protects the values from being changed by mistake, and lets the class check every change.

- A **getter** is a method that **returns** the value of an attribute, such as `GetBikeID()`.
- A **setter** is a method that **changes** the value of an attribute, usually from a parameter, such as `SetOnHire(Status : BOOLEAN)`. A setter can validate the new value first.

Inheritance lets a new class (the **subclass**, child or derived class) take all the attributes and methods of an existing class (the **superclass**, parent or base class). The subclass adds its own attributes and methods, and can change (**override**) the ones it takes. In pseudocode it is `CLASS EBike INHERITS Bike`, and the subclass's constructor calls the parent's with `SUPER.NEW(...)`.

Polymorphism means a method with the **same name** behaves **differently** in different classes of an inheritance family, because each subclass redefines (overrides) it. Calling `HireCost(3)` on a `Bike` and on an `EBike` runs different code.



In a **class diagram** each class is a box in three parts: the name, then the attributes with their data types, then the methods. In an **inheritance diagram** an arrow goes **from each subclass to its superclass**, and a subclass box lists **only what it adds or overrides**.

Containment (aggregation) is a "has a" link rather than an "is a" link: one class contains objects of another class as attributes. A `Depot` object might hold an array of `Bike` objects; a bike is not a kind of depot, so this is containment, not inheritance.

In pseudocode a class looks like this:

```

CLASS Bike
  PRIVATE BikeID : STRING
  PRIVATE DailyRate : REAL
  PRIVATE OnHire : BOOLEAN

  PUBLIC PROCEDURE NEW(NewID : STRING, NewRate : REAL)
    BikeID ← NewID
    DailyRate ← NewRate
    OnHire ← FALSE
  ENDPROCEDURE

```

```

PUBLIC FUNCTION GetBikeID() RETURNS STRING
    RETURN BikeID
ENDFUNCTION

PUBLIC PROCEDURE SetOnHire(Status : BOOLEAN)
    OnHire ← Status
ENDPROCEDURE

PUBLIC FUNCTION HireCost(Days : INTEGER) RETURNS REAL
    RETURN Days * DailyRate
ENDFUNCTION
ENDCLASS

Bike1 ← NEW Bike("B07", 14.50)      // instantiate an object
OUTPUT Bike1.HireCost(3)           // 43.5

```

In Python the attributes are made private with two underscores (`self.__BikeID`) and the constructor is `__init__`; in Java and Visual Basic you write `private` and `public`.

Declarative programming

A declarative program is a knowledge base of **clauses**, each ending in a full stop.

- A **fact** states something that is true: `planet(mars)`. means "Mars is a planet"; `orbits(phobos, mars)`. means "Phobos orbits Mars". Names of things (constants) start with a lower-case letter.
- A **rule** states when something is true, from other facts: `moon_of_planet(M, P) IF orbits(M, P) AND planet(P)`. The rule's head is true whenever everything after `IF` is true. `AND` (or a comma), `OR` and `NOT` join the conditions.
- A **goal** (query) asks the language to find values that make it true. Names that start with a capital letter are **variables**. With the facts `orbits(phobos, mars)`. `orbits(io, jupiter)`. `orbits(deimos, mars)`., the goal `orbits(M, mars)` returns `M = phobos, deimos`, in the order the facts appear.

A goal can have two parts joined by `AND`, using the same variable in both, such as `orbits(M, P) AND planet(P)`. `NOT` succeeds when the fact cannot be found: `NOT orbits(titan, mars)` is true because no such fact exists.

File processing

A program that keeps data after it stops must store it in a file. Every file is **opened** in a mode, used, then **closed** so the data is saved and the file is released. In pseudocode:

Operation	Pseudocode	What it does
Open to read	<code>OPENFILE "Log.txt" FOR READ</code>	read lines from the start

Operation	Pseudocode	What it does
Open to write	OPENFILE "Log.txt" FOR WRITE	makes a new, empty file: anything already in it is lost
Open to append	OPENFILE "Log.txt" FOR APPEND	keeps what is there and adds lines at the end
Read a line	READFILE "Log.txt", Line	puts the next line into Line
Write a line	WRITEFILE "Log.txt", Line	writes Line as a new line
End of file?	EOF("Log.txt")	TRUE once there is nothing left to read
Close	CLOSEFILE "Log.txt"	always, when you have finished

A **text file** is read and written one line at a time, from the start, so text-file processing is **serial** (records in the order they were added) or **sequential** (records in order of a key field). To read the whole file: WHILE NOT EOF(...), READFILE, process the line, ENDWHILE.

A **random file** holds **records** of a user-defined type at numbered positions, and any record can be reached **directly** without reading the ones before it. The position often comes from a hashing function applied to the record's key.

Operation	Pseudocode
Open	OPENFILE "Stock.dat" FOR RANDOM (for reading and writing records)
Move to record position Pos	SEEK "Stock.dat", Pos
Read the record there into a variable	GETRECORD "Stock.dat", Item
Write a variable as the record there	PUTRECORD "Stock.dat", Item

Always SEEK before each GETRECORD or PUTRECORD, so you know which position is used. The variable must have the record's type, declared with TYPE ... ENDTYPE, and a field is reached with a dot: Item.Quantity.

Exceptions

An **exception** is an unexpected event that happens **while a program is running** (at run time) and interrupts its normal flow; if nothing deals with it, the program stops (crashes). Typical causes:

- **programming errors** that the compiler did not catch: dividing by zero, using an array index outside the bounds;
- **user errors**: typing letters where a number is expected, giving the name of a file that does not exist;
- **hardware or connection problems**: a disk that fails, a printer or network connection that is lost, a device that is full.

Exception handling is code that catches the exception when it happens and responds to it, for example by showing a helpful message, asking again or using a default value, so the program carries on rather than stopping part-way. Use it wherever something **outside the program's control** can go wrong: opening and reading files, converting what a user types, dividing by a value that could be zero, using hardware or a network.

```
try:
    File = open("Scores.txt", "r")
    Line = File.readline()
    Score = int(Line)
    File.close()
except FileNotFoundError:
    print("Scores.txt could not be found")
except ValueError:
    print("The file does not start with a whole number")
```

The code that might fail goes in the `try` block, including opening **and closing** the file, so a failure never leaves the program trying to close a file that was never opened. Each `except` block (catch in Java, Catch in Visual Basic) handles one kind of exception.

Key facts and formulas

There is no formula list for 9618. The assembly instruction set is printed in each question; learn everything else.

Formula	Status
Assembly instruction table (LDM, LDD, LDI, LDX, ADD, SUB, STO, <label>: <data>, # for denary)	Given (printed in the question)
Immediate: operand is the value. Direct: operand is the address. Indirect: operand is the address of the address. Indexed: operand + IX. Relative: offset from a base address	Learn it
Class: template of attributes and methods. Object: an instance of a class	Learn it
Encapsulation: data and methods in one class, attributes <code>PRIVATE</code> , reached through methods	Learn it
Getter returns an attribute's value; setter changes it	Learn it
Inheritance: subclass takes the superclass's attributes and methods and can add or override them	Learn it
Polymorphism: same method name, redefined to behave differently in subclasses	Learn it
Containment (aggregation): a class holds objects of another class ("has a")	Learn it
Fact $f(a, b)$, rule $\text{head}(X) \text{ IF } p(X) \text{ AND NOT } q(X)$, goal $f(X, b)$ returns X = ...	Learn it

Formula	Status
OPENFILE ... FOR READ / WRITE / APPEND / RANDOM, READFILE, WRITEFILE, EOF (), CLOSEFILE	Learn it
SEEK file, position, GETRECORD file, variable, PUTRECORD file, variable	Learn it
Exception: unexpected run-time event; causes: programming, user, hardware errors	Learn it

How to do it

In the Paper 3 questions we have tagged for this topic (44 questions, 2021–2025, several of them repeated on two or three paper variants), the types came up this often. 14 of the 44 questions mix two types, so the counts add up to more than 44.

Question type	Questions
Define or match OOP terms	12
Declarative facts, rules and goals	9
Complete file-handling pseudocode	9
Explain exceptions and exception handling	7
Complete a class or inheritance diagram	7
Write assembly code with labels	5
Describe or identify programming paradigms	5
Name addressing modes and give the ACC	2
Write a class in pseudocode	2

1. Define or match OOP terms (12 questions)

1. Use OOP words in every definition: **class**, **object**, **attribute (property)**, **method**. Never explain a term with its own name ("a getter gets...", "inheritance is when a class inherits...").
2. Give one full idea per mark. Short versions to build on:
 - **Instance**: one object created from a class (an occurrence of an object).
 - **Encapsulation**: data and methods bundled into one class, with the attributes hidden so only the class's own methods reach them.
 - **Inheritance**: a subclass **derives** all the attributes and methods of its superclass, and can add to them or override them.
 - **Polymorphism**: a method of the same name is **redefined in subclasses**, so it behaves differently depending on the object's class.
 - **Getter / setter**: a method that **returns / updates** an attribute's value.

- **Property / attribute:** data, with a data type, that each object of the class holds. **Method:** a procedure or function defined in the class that acts on its attributes.
- 3. **Structure of a class:** attributes with their data types; methods that act on them; getters and setters; a constructor that creates objects.
- 4. **Object against class:** the class is the description, each object is one thing built from it that holds its own values. Anchor each point in an example, such as Bike against Bike1 and Bike2 (see the table in "Understand it").
- 5. **Why attributes are PRIVATE:** so they can only be reached through the class's own methods (encapsulation), never "for security".

2. Declarative facts, rules and goals (9 questions)

1. **Adding facts:** copy the exact predicate names and argument order from the given clauses, all lower case, one fact per piece of information, each ending in a full stop. A new person, thing or category needs its own fact too (`student(lena)` . as well as `choice(lena, art)` .).
2. **Result of a goal:** go down the clauses **in order** and list every match: `X = first, second`.
3. **Writing a goal:** use the variable name you are given, with a capital letter. For "all X that are both...", join two parts with AND and the **same** variable: `plant(P) AND too_damp(P, bathroom)`.
4. **Writing a rule:** turn the sentence into `head(A, B) IF ... AND ... AND NOT ...,` one condition per phrase:
 - "P is a plant" → `plant(P)`;
 - "R is not too damp for P" → `NOT too_damp(P, R)`.

Use exactly the variable letters given in the head, and add nothing else.

3. Complete file-handling pseudocode (9 questions)

1. **Open** the file before the loop, in the right mode: `FOR RANDOM` for a .dat file of records, `FOR READ/WRITE/APPEND` for a text file. If the program copies from one file to another, **open both** and **close both**.
2. **Random file, one record:** `SEEK file, position` then `GETRECORD file, variable` to read, or `SEEK` then `PUTRECORD file, variable` to write. The position may come from `INPUT`, a loop counter or a hash function (`Pos ← Hash(Key)`).
3. **Random file, search:** a loop `WHILE NOT Found AND Pos ≤ Max`, with `SEEK`, `GETRECORD`, a comparison, and `Pos ← Pos + 1`; after the loop, `IF NOT Found THEN OUTPUT ...`.
4. **Empty slot:** a record whose key field is " " is free: store the new record there with `PUTRECORD` and set the found/stored flag.
5. **Text file copy:** `WHILE NOT EOF(source), READFILE source, Line`, test or change the line, `WRITEFILE target, Line, ENDWHILE`.
6. **Close** every file after the loop, never inside it.

Fill each gap with a whole line (the mark scheme gives one mark per completed line) and use the question's identifiers exactly, including the file name in quotes.

4. Explain exceptions and exception handling (7 questions)

- **What is an exception:** something the program did not plan for, such as a missing file or a zero divisor, that turns up only at run time and breaks the normal order of execution; if nothing deals with it, the run ends there.
- **What is exception handling:** code that catches the exception as it happens and deals with it, so the program can keep going, for example with `try ... except` (or `catch`) and an error message.
- **Causes / examples:** give **different kinds:** a programming error (divide by zero, array index out of range), a user error (wrong data type, wrong file name), a hardware error (disk failure, lost printer connection). For "a reason it causes a problem", say what goes wrong: a division by zero has no answer, so the calculation cannot finish and the run ends with an error message instead of a result.
- Do not describe **interrupt** handling: that is a different topic.

5. Complete a class or inheritance diagram (7 questions)

1. **Attributes:** one per piece of data, a sensible name and a data type that fits the data: a phone number is `STRING` (it can start with 0 and is never used in sums), a date is `DATE`, a whole-number scale is `INTEGER`, yes/no is `BOOLEAN`.
2. **Methods:** a **setter** `SetX(NewX : TYPE)` for every value the question says is assigned, a **getter** `GetX( )` for every value to be returned, and `Constructor( )` (or `NEW`) if objects are created with values. The name after `Set/Get` must match the attribute exactly.
3. **Inheritance diagram:** put the shared attributes and methods in the superclass box only; each subclass box has just its **extra** attributes and the methods it **overrides** (such as its own constructor and display method). Draw an arrow from **each** subclass to the superclass.

6. Write assembly code with labels (5 questions)

1. Read each bullet in turn and turn it into one or two instructions:
 - "store the denary value n as X" → `LDM #n, STO X;`
 - "store the contents of location a as X" → `LDD a, STO X;`
 - "add (subtract) the value in X" → `ADD X (SUB X);` "subtract the contents of location a" → `SUB a;`
 - "the value at the address held in X" → `LDI X;`
 - "store the result in Y" → `STO Y.`
2. Keep track of what is in the ACC: an `ADD` or `SUB` uses whatever the last load left there, so load first.
3. Set up every label in the **separate** label table, away from the code: `X: 250`. A variable the program fills (the result) is set up too, with its final value if the question asks for values.
4. Work the values through to fill the table, using the memory contents given.

7. Describe or identify programming paradigms (5 questions)

- **Imperative:** a sequence of commands, executed in order, that change the program's state (its variables); says **how** to solve the problem; may use procedures and functions.
- **Declarative:** facts and rules; the program says **what** result is wanted, and a goal is answered by searching them.
- **Recognising code:** facts like `planet(mars) .` → declarative; a FOR loop with assignments → imperative; opcodes such as `LDD` or `ADD #1` → low-level; `class`, `private`, a constructor → object-oriented. Name the paradigm; don't describe what the code does.

8. Name addressing modes and give the ACC (2 questions)

Look at the opcode: `LDM #n` is **immediate** (ACC = n); `LDD a` is **direct** (ACC = contents of a); `LDI a` is **indirect** (go to a, read the address there, then load from that address); `LDX a` is **indexed** (load from a + IX). Write both the mode's name and the number in the ACC.

9. Write a class in pseudocode (2 questions)

1. `CLASS Name ... ENDCLASS` (don't forget `ENDCLASS`).
2. Every attribute `PRIVATE Attribute : TYPE`, in the names and types given.
3. The constructor `PUBLIC PROCEDURE NEW(...)` with one parameter for each value supplied when an object is made; assign each to its attribute. Attributes that always start the same (a count of 0, an empty string) are set in the constructor **without** a parameter.
4. Getters as `PUBLIC FUNCTION GetX() RETURNS TYPE` with `RETURN X`; setters as `PUBLIC PROCEDURE SetX(NewX : TYPE)`.
5. A subclass: `CLASS Sub INHERITS Super`, its constructor takes the parent's values as well as its own and calls `SUPER.NEW(...)`; don't declare the parent's attributes again.

Worked examples

The bold codes show where marks are earned, as in Cambridge mark schemes. The number is how many marks.

- **B1** mark for a correct result on its own, with no method needed.
-

Example 1

A processor has one register, the ACC, and an index register, IX. Its instruction set includes `LDM #n`, `LDD <address>`, `LDI <address>`, `LDX <address>`, `ADD <address>`, `SUB <address>`, `STO <address>` and `<label>: <data>`, as in the exam's table. Memory holds:

Address	300	301	305	320
Contents	40	320	301	95

(a) IX contains 5. State the addressing mode and the contents of the ACC after each of `LDM #301`, `LDD 301`, `LDI 305` and `LDX 300`. **[4]**

(b) Write assembly code, using only the instructions above, to:

- store the denary value 12 as a named constant `Fee`;
- load the value at the address held in location 301;
- subtract the constant;
- add the contents of location 300;
- store the result in a variable `Net`.

Show the labelled locations and their values. **[6]**

(a)

- `LDM #301`: immediate, `ACC = 301`. **B1**
- `LDD 301`: direct, `ACC = 320` (the contents of 301). **B1**
- `LDI 305`: indirect; 305 holds 301, and 301 holds 320, so `ACC = 320`. **B1**
- `LDX 300`: indexed; $300 + 5 = 305$, which holds 301, so `ACC = 301`. **B1**

(b)

Opcode	Operand	ACC afterwards
LDM	#12	12
STO	Fee	12
LDI	301	95
SUB	Fee	83
ADD	300	123
STO	Net	123

B1 for `LDM #12` and `STO Fee`; **B1** for `LDI 301` (301 holds 320, and 320 holds 95); **B1** for `SUB Fee`; **B1** for `ADD 300`; **B1** for `STO Net`.

Label	Contents
Fee:	12
Net:	123

B1 for both labelled locations, away from the code, with $95 - 12 + 40 = 123$ in `Net`.

Example 2

A cycle-hire company stores its bikes as objects of the class `Bike`. Each bike has an ID (for example "B07"), a daily hire rate in dollars and whether it is out on hire. Electric bikes are objects of the subclass `EBike`, which also stores the battery range in whole kilometres. An electric bike costs its daily rate for each day **plus** a charging fee of \$6.00 per hire.

(a) Draw the inheritance diagram for `Bike` and `EBike`, showing attributes with data types and methods, including a constructor, getters and setters where they are needed. **[4]**

(b) Write pseudocode for the class `EBike`. **[5]**

(c) `Bike1 ← NEW Bike("B07", 14.50)` and `Ebike1 ← NEW EBike("E02", 22.00, 70)`. Give the value returned by `Bike1.HireCost(3)` and by `Ebike1.HireCost(3)`, and name the OOP feature this shows. **[3]**

(a) The second diagram in "Understand it" (leave out `CargoBike`). `Bike`: `BikeID : STRING`, `DailyRate : REAL`, `OnHire : BOOLEAN`; `Constructor()`, `GetBikeID()`, `SetOnHire()`, `HireCost()`. **B1** for attributes with sensible types, **B1** for the methods. `EBike`: `BatteryRange : INTEGER`; `Constructor()`, `GetBatteryRange()`, `HireCost()`. **B1** for only the extra attribute and the overridden methods, **B1** for the arrow from `EBike` to `Bike`.

(b)

```

CLASS EBike INHERITS Bike
    PRIVATE BatteryRange : INTEGER

    PUBLIC PROCEDURE NEW(NewID : STRING, NewRate : REAL, NewRange : INTEGER)
        SUPER.NEW(NewID, NewRate)
        BatteryRange ← NewRange
    ENDPROCEDURE

    PUBLIC FUNCTION GetBatteryRange() RETURNS INTEGER
        RETURN BatteryRange
    ENDFUNCTION

    PUBLIC FUNCTION HireCost(Days : INTEGER) RETURNS REAL
        RETURN SUPER.HireCost(Days) + 6.00
    ENDFUNCTION
ENDCLASS

```

B1 `CLASS EBike INHERITS Bike...ENDCLASS`; **B1** `PRIVATE BatteryRange : INTEGER`; **B1** a constructor with all three parameters that calls `SUPER.NEW`; **B1** the getter; **B1** `HireCost` overridden, reusing the parent's version.

(c) `Bike1.HireCost(3) = 3 × 14.50 = 43.50`. **B1** `Ebike1.HireCost(3) = 3 × 22.00 + 6.00 = 72.00`.

B1 The same method name gives different behaviour in the subclass: **polymorphism**. **B1**

Example 3

A declarative language holds these facts about house plants.

```
01 plant(fern).
02 plant(cactus).
03 plant(basil).
04 plant(orchid).
05 room(kitchen).
06 room(hallway).
07 room(bathroom).
08 room(conservatory).
09 sunny(kitchen).
10 sunny(conservatory).
11 too_damp(cactus, bathroom).
12 too_damp(cactus, kitchen).
13 too_damp(basil, bathroom).
```

Clause 09 means "the kitchen is sunny"; clause 11 means "the bathroom is too damp for a cactus".

(a) An aloe is a plant, and the bathroom is too damp for it. Write the additional clauses. **[2]**

(b) The goal `too_damp(cactus, R)` returns `R = bathroom, kitchen`. Write the result of the goal `too_damp(P, bathroom)` for the original clauses 01–13. **[1]**

(c) Write a goal, using the variable `Spot`, to find every sunny room that is not too damp for a cactus, and give its result. **[3]**

(d) `P` can be placed in `R` if `P` is a plant and `R` is a room and `R` is not too damp for `P`. Write this as a rule `can_place(P, R)`, and give the result of the goal `can_place(cactus, R)`. **[5]**

(a) 14 `plant(aloe)`. **B1** 15 `too_damp(aloe, bathroom)`. **B1**

(b) `P = cactus, basil` **B1** (clauses 11 and 13, in that order).

(c) `sunny(Spot) AND NOT too_damp(cactus, Spot)` **B1** for `sunny(Spot)`, **B1** for the `NOT` part with the same variable. The kitchen is sunny but too damp, so `Spot = conservatory`. **B1**

(d)

```
can_place(P, R)
    IF plant(P) AND room(R) AND NOT too_damp(P, R).
```

B1 `plant(P)`, **B1** `room(R)`, **B1** `NOT too_damp(P, R)`, **B1** correct IF, AND and full stop with nothing extra. Rooms are tried in order; the kitchen and bathroom fail, so `R = hallway, conservatory`. **B1**

Example 4

A gym stores its 300 lockers in the random file `Lockers.dat`, one record per locker, at the position equal to the locker number. A free locker has `Member = ""`.

(a) Write pseudocode to declare the record type `TLocker`, with fields `LockerNo` (a whole number), `Member` (a name) and `Paid` (yes or no). **[2]**

(b) Write pseudocode that asks for a locker number and a member's name. If the locker is free, it gives it to the member with `Paid` set to `FALSE`; otherwise it outputs a message. **[6]**

(c) Give one exception that could happen when this code runs, and describe how exception handling would help. **[3]**

(a)

```
TYPE TLocker
    DECLARE LockerNo : INTEGER
    DECLARE Member : STRING
    DECLARE Paid : BOOLEAN
ENDTYPE
```

B1 TYPE ... ENDTYPE, **B1** all three fields with the right types.

(b)

```
DECLARE Locker : TLocker
DECLARE Number : INTEGER
DECLARE Name : STRING
OPENFILE "Lockers.dat" FOR RANDOM
OUTPUT "Locker number (1 to 300): "
INPUT Number
OUTPUT "Member's name: "
INPUT Name
SEEK "Lockers.dat", Number
GETRECORD "Lockers.dat", Locker
IF Locker.Member = "" THEN
    Locker.LockerNo ← Number
    Locker.Member ← Name
    Locker.Paid ← FALSE
    SEEK "Lockers.dat", Number
    PUTRECORD "Lockers.dat", Locker
ELSE
    OUTPUT "Locker ", Number, " is already taken"
ENDIF
CLOSEFILE "Lockers.dat"
```

B1 OPENFILE ... FOR RANDOM; **B1** SEEK to the number input; **B1** GETRECORD into a TLocker variable; **B1** the test for " "; **B1** setting the fields then SEEK and PUTRECORD; **B1** the message and CLOSEFILE after the IF. With locker 17 free, the record at 17 becomes (17, the name, FALSE); with locker 18 taken, the message appears and nothing is written.

(c) For example, the user types "seventeen" for the locker number (or the file Lockers.dat cannot be found). **B1** Without handling, this run-time error would stop the program. **B1** With a try ... except block round the input and file operations, the error is caught and a message is output asking for a whole number (or saying the file is missing), so the program keeps running. **B1**

Common mistakes

- **Defining a term with its own name.** "Inheritance is when a class inherits from another" earns nothing; say the subclass **takes (derives)** the attributes and methods of the parent. The same goes for "a getter gets a value": say it **returns** it.
- **Vague OOP answers.** Use the words class, object, attribute and method. "Private stops hackers" is about security, not encapsulation; the point is that attributes are reached **only through the class's own methods**.
- **Inheritance diagrams without arrows**, with arrows the wrong way, or with the parent's attributes copied into every subclass. Arrows go from subclass to superclass, and each subclass lists only what is new or overridden.
- **Class diagrams:** a phone number as INTEGER, a setter with no parameter, only one setter or getter when two are asked for, and getter or setter names that don't match the attribute.
- **Writing a class:** leaving out ENDClass, renaming the given attributes or parameters, and making a fixed starting value (such as a count of 0) a constructor parameter.
- **Declarative syntax:** capital letters on constants, missing full stops, a changed predicate name, or a two-part goal with no AND between the parts. In a rule, use exactly the variables in the head and add nothing more.
- **Assembly:** a missing # (so LDM 250 or ADD 20 uses an address, not the number), no labelled locations for constants and variables, and forgetting that ADD needs a value already in the ACC. Immediate addressing is the mode most often not recognised.
- **File handling:** not closing the file, closing it inside the loop, opening only one of two files, forgetting SEEK before GETRECORD or PUTRECORD, and opening with WRITE when data must be kept (it empties the file).
- **Exceptions:** describing interrupt handling instead. And in code, open and close the file **inside** the try block, so a failed open is not followed by closing a file that is not open.
- **Paradigms:** describing what a code example does instead of naming its paradigm, or swapping imperative (how, step by step) with declarative (what, facts and rules).

How to get the marks

- **"Define" / "Describe" an OOP term:** a definition in OOP words (1 mark), then what it allows or an example (a second mark if there are two). For a two-mark term like inheritance: what the subclass takes, **and** from which class, **and** that it can extend or override.
- **"Identify" / "State":** the term or value alone, such as "instance" or "encapsulation".
- **Gap-fill pseudocode:** one mark per completed line; copy identifiers and file names exactly, keep THEN, ENDIF, ENDWHILE, and use ← for assignment.
- **Class diagrams** are marked point by point: the attributes with correct types, the setters, their parameters with types, the getters, and a fully correct pair for one attribute. Every named method needs its brackets.
- **Declarative:** each extra clause is a mark; the rule earns one mark per condition plus one for the operators and punctuation being all correct.
- **Assembly:** each correct instruction in the right order earns its mark, and separate marks go to setting up the labels and their values. Keep the code and the label table apart.
- **"Explain" exception handling:** what it does (responds to the unexpected event at run time) **and** why (so the program does not stop unexpectedly), plus an example if asked.
- **Examples** must be of different kinds when two are asked for: "divide by zero" and "dividing a number by 0" are one example.

Quick check

1. Memory holds 84 at address 80, 3 at 81, 90 at 82, 12 at 83 and 57 at 84, and IX holds 2. Give the contents of the ACC after each of `LDM #82, LDD 82, LDI 80` and `LDX 81`, and name each mode.
2. A team splits a project four ways. Name the paradigm each part uses: (i) the program lists what is true about train routes and answers questions such as "which towns can be reached from Leeds?" by searching them; (ii) a routine on a small device runs `LDD 90, ADD #1, STO 90` to count button presses; (iii) each ticket is stored with its own fare and date, and code outside can reach them only through `GetFare()` and `SetDate()`; (iv) a list of fares is totalled by a loop that adds each one to a running total, step by step.
3. Using the facts `likes(ana, chess). likes(ben, chess). likes(ana, tennis). likes(cho, tennis). plays(ben, chess).`, give the result of the goal `likes(X, tennis)`. Then write a rule `might_join(P, G): P might join G if P likes G and P does not already play G`, and give the result of `might_join(X, chess)`.
4. The text file `Scores.txt` holds 3 lines. A program opens it `FOR WRITE`, writes 2 lines and closes it; then opens it `FOR APPEND`, writes 1 line and closes it. How many lines does the file hold now, and why?
5. A program runs `Bike1 ← NEW Bike("B07", 14.50)`, then `Bike2 ← NEW Bike("B12", 16.00)`, then `Bike2.SetOnHire(TRUE)`, using the `Bike` class above. (i) How many classes and how many objects are there? (ii) What is `Bike1`'s `OnHire` value now, and why? (iii) Why would the line `Bike2.OnHire ← FALSE`, written outside the class, be rejected, and what should be written instead?

Answers

1. `LDM #82`: immediate, 82. `LDD 82`: direct, 90. `LDI 80`: indirect; 80 holds 84, which holds 57, so 57. `LDX 81`: indexed; $81 + 2 = 83$, which holds 12, so 12.
2. (i) Declarative. (ii) Low-level. (iii) Object-oriented. (iv) Imperative (procedural).
3. `X = ana, cho.might_join(P, G) IF likes(P, G) AND NOT plays(P, G)`. Ana and Ben like chess, but Ben already plays it, so `X = ana`.
4. 3 lines. Opening `FOR WRITE` empties the file, so its 3 old lines are lost and it then holds 2; `APPEND` adds 1 line at the end, making $2 + 1 = 3$.
5. (i) One class, `Bike`, and two objects (instances), `Bike1` and `Bike2`. (ii) `FALSE`: the constructor set it to `FALSE`, and the setter changed only `Bike2`'s copy, because each object holds its own attribute values. (iii) `OnHire` is `PRIVATE` (encapsulation), so code outside the class cannot reach it directly; use the setter, `Bike2.SetOnHire(FALSE)`.

Past questions to try

- [9618/31/M/J/25 Q12](#): looking up a record by position in a random file.
- [9618/33/O/N/24 Q9](#): why attributes are private, and completing a class diagram.
- [9618/32/M/J/24 Q10](#): new facts, a goal's result and a rule with `NOT`.
- [9618/33/O/N/25 Q13](#): assembly code with a constant and three labelled variables.