2.4 Programming Paradigms and OOP

OCR H446 · Calculators not allowed · about 75 minutes · Write-in edition

Total Marks

Name: _____ Date: ____ / ____ / _____

Answer ALL questions. Show all your working.

1 A small program for a Bristol newsagent totals up the cost of a customer's magazines. It is written using OCR's Exam Reference Language conventions (assignment <-, iteration FOR ... NEXT, selection IF ... ENDIF).

```
PROCEDURE CalculateTotal(price, quantity)
  total <- price * quantity
  RETURN total
ENDPROCEDURE

PROCEDURE PrintReceipt(item, price, quantity)
  total <- CalculateTotal(price, quantity)
  OUTPUT item, " x ", quantity, " = GBP ", total
ENDPROCEDURE

CALL PrintReceipt("Weekly Puzzle Magazine", 2.50, 4)
```

(a) Identify the programming paradigm used in the code above. **(1)**

(b) Give two reasons for your answer to part (a), referring directly to the code. **(2)**

(c) State one change that would need to be made to this code for it to become object-oriented. **(1)**

(Total for Question 1 is 4 marks)

---

2 A secondary school in Norwich wants a pseudocode class to represent a book held in its school library.

(a) Write pseudocode for a class LibraryBook with four private attributes: title (STRING), author (STRING), isbn (STRING) and isOnLoan (BOOLEAN); and a constructor PROCEDURE NEW(t, a, i) that sets title, author and isbn from its parameters and sets isOnLoan to FALSE. **(3)**

(b) Write a public procedure BorrowBook() for LibraryBook that sets isOnLoan to TRUE and outputs a confirmation message if the book is not already on loan; otherwise it should output a message stating the book is already on loan and leave isOnLoan unchanged. **(3)**

(Total for Question 2 is 6 marks)

3 A payroll system for a chain of shops in Glasgow uses a class Employee, shown below, and needs a subclass Teacher for staff at the company's in-house training academy.

```
CLASS Employee
  PRIVATE name : STRING
  PRIVATE salary : REAL

  PUBLIC PROCEDURE NEW(n, s)
    name <- n
    salary <- s
  ENDPROCEDURE

  PUBLIC FUNCTION GetDetails() RETURNS STRING
    RETURN name + ", salary GBP " + STR(salary)
  ENDFUNCTION
ENDCLASS
```

(a) Complete the class header line below so that Teacher inherits from Employee and adds one new private attribute subject (STRING). **(2)**

```
CLASS Teacher ______
  PRIVATE subject : STRING
  ...
ENDCLASS
```

(b) Write pseudocode for Teacher's constructor PROCEDURE NEW(n, s, subj), which should call the Employee constructor to set name and salary, then set subject from its own parameter. **(3)**

(c) Explain what is meant by 'method overriding', using a version of GetDetails written for Teacher as your example. **(2)**

(Total for Question 3 is 7 marks)

4 A graphics package for a design studio in Manchester defines a superclass Shape and two subclasses, Circle and Rectangle, each overriding a function CalculateArea.

```
CLASS Shape
  PUBLIC FUNCTION CalculateArea() RETURNS REAL
    RETURN 0
  ENDFUNCTION
ENDCLASS

CLASS Circle INHERITS Shape
  PRIVATE radius : REAL
  PUBLIC PROCEDURE NEW(r)
    radius <- r
  ENDPROCEDURE
  PUBLIC FUNCTION CalculateArea() RETURNS REAL
    RETURN pi * radius ^ 2
  ENDFUNCTION
ENDCLASS

CLASS Rectangle INHERITS Shape
  PRIVATE width : REAL
  PRIVATE height : REAL
  PUBLIC PROCEDURE NEW(w, h)
    width <- w
    height <- h
  ENDPROCEDURE
  PUBLIC FUNCTION CalculateArea() RETURNS REAL
    RETURN width * height
  ENDFUNCTION
ENDCLASS
```

A list shapes contains a mixture of Circle and Rectangle objects. The following loop is used to print each shape's area:

```
FOR EACH s IN shapes
  OUTPUT s.CalculateArea()
NEXT s
```

(a) Explain what is meant by polymorphism in object-oriented programming, referring to the Shape, Circle and Rectangle classes above. **(3)**

(b) Using the loop given in the question, explain why the code that processes the shapes list does not need to know whether each item is a Circle or a Rectangle. **(3)**

(Total for Question 4 is 6 marks)

5 A weather-logging tool for a network of stations across the UK uses functions to process a list of daily temperature readings (in degrees Celsius).

```
globalTotal <- 0

FUNCTION AddToGlobalTotal(x) RETURNS REAL
  globalTotal <- globalTotal + x
  RETURN globalTotal
ENDFUNCTION

FUNCTION CelsiusToFahrenheit(c) RETURNS REAL
  RETURN (c * 9 / 5) + 32
ENDFUNCTION
```

(a) Define what is meant by a 'pure function' in functional programming. **(2)**

(b) State one benefit of avoiding mutable/global state when writing functions. **(2)**

(c) Identify which of the two functions given in the question, AddToGlobalTotal or CelsiusToFahrenheit, is impure, and give a reason for your answer. **(2)**

(Total for Question 5 is 6 marks)

6 A vehicle-hire company in Oxford models its fleet using the classes below.

```
CLASS Vehicle
  PROTECTED make : STRING

  PUBLIC PROCEDURE NEW(m)
    make <- m
  ENDPROCEDURE

  PUBLIC FUNCTION Describe() RETURNS STRING
    RETURN make + ": generic vehicle"
  ENDFUNCTION
ENDCLASS

CLASS Car INHERITS Vehicle
  PUBLIC FUNCTION Describe() RETURNS STRING
    RETURN make + ": car, 4 wheels"
  ENDFUNCTION
ENDCLASS

CLASS Motorbike INHERITS Vehicle
  PUBLIC FUNCTION Describe() RETURNS STRING
    RETURN make + ": motorbike, 2 wheels"
  ENDFUNCTION
ENDCLASS
```

The following driver program is then run:

```
fleet <- []
fleet.APPEND(NEW Vehicle("Generic Co"))
fleet.APPEND(NEW Car("Skoda"))
fleet.APPEND(NEW Motorbike("Triumph"))

FOR EACH v IN fleet
  OUTPUT v.Describe()
NEXT v
```

(a) State the three lines of output produced by the driver program, in the order they are produced. **(4)**

(b) Explain why the second and third lines of output use the overridden versions of Describe() in Car and Motorbike, rather than the version defined in Vehicle, even though the loop only ever calls v.Describe(). **(4)**

(Total for Question 6 is 8 marks)

7 A small business owner in Cardiff is developing a new student grade-tracking system for a secondary school. They are deciding whether to use the procedural paradigm or the object-oriented paradigm to build it. The system must store many students' names, subjects and marks, calculate averages and grade boundaries, and will be extended over several years by different developers as new features (e.g. attendance tracking) are added.

(a) Compare and contrast the procedural and object-oriented approaches for developing this grade-tracking system, discussing at least two relevant factors (for example: maintainability, code reuse, protection of data, or suitability for a system that will be extended by a team of developers over time). **(8)**

(Total for Question 7 is 8 marks)

8 A developer is designing a class hierarchy for a delivery-tracking system and is considering whether a DeliveryDriver class should inherit from both a Person class and a VehicleOperator class (multiple inheritance), or whether it would be better to keep the hierarchy simpler.

(a) Discuss one potential problem that can arise from using multiple inheritance (a class inheriting directly from two or more superclasses). **(3)**

(b) Discuss one potential problem that can arise from overusing deep inheritance hierarchies in general (a long chain of subclasses inheriting from subclasses), even without multiple inheritance. **(3)**

(Total for Question 8 is 6 marks)

9 A leisure centre in Swansea wants a new booking system for its fitness classes. Classes have different formats: some are individually instructed sessions (e.g. personal training), and some are group sessions (e.g. spin classes, yoga) with a fixed maximum number of participants. The centre expects to add further class formats in future years and wants the system to be easy to maintain and extend.

(a) Design an object-oriented model for this booking system. Your answer should describe (in words and/or pseudocode) at least: a base class shared by all class formats, two subclasses with genuinely different (polymorphic) behaviour, appropriate encapsulation with validated attributes, and a justification of why an object-oriented approach suits a system that will keep being extended with new class formats. **(10)**

(Total for Question 9 is 10 marks)

Mark scheme · 2.4 Programming Paradigms and OOP

Question 1

(a) procedural (programming) cao **B1**

(a) Answer: Procedural programming.

(b) the code is organised as procedures (CalculateTotal, PrintReceipt) that are called in sequence/one from another (oe) **B1**

(b) there are no classes or objects defined; data (price, quantity, total) is passed directly as parameters rather than being bundled with methods (oe) **B1**

(b) Answer: The code uses two procedures that call each other in sequence, and there are no classes or objects; data is simply passed between procedures as parameters.

(c) define a class (e.g. Receipt or Item) that bundles the item's data (item, price, quantity) together with methods such as CalculateTotal and PrintReceipt (oe) **B1**

(c) Answer: Create a class, for example Receipt, whose attributes hold item, price and quantity, and whose methods include CalculateTotal and PrintReceipt.

Question 2

(a) declares all four attributes as PRIVATE with appropriate types (title, author, isbn as STRING; isOnLoan as BOOLEAN) **M1**

(a) constructor correctly assigns the three parameters to title, author and isbn **M1**

(a) constructor correctly initialises isOnLoan to FALSE; overall class syntax logically complete (ignore minor syntax errors) **A1**

(a) Answer:

```
CLASS LibraryBook
  PRIVATE title : STRING
  PRIVATE author : STRING
  PRIVATE isbn : STRING
  PRIVATE isOnLoan : BOOLEAN

PUBLIC PROCEDURE NEW(t, a, i)

    title <- t
    author <- a
    isbn <- i
    isOnLoan <- FALSE
  ENDPROCEDURE
ENDCLASS
```

(b) uses an IF statement to test isOnLoan = FALSE (or NOT isOnLoan) **M1**

(b) TRUE branch sets isOnLoan <- TRUE and outputs a confirmation message **M1**

(b) ELSE branch outputs a message that the book is already on loan and does not change isOnLoan; overall logic complete (ignore minor syntax errors) **A1**

(b) Answer:PUBLIC PROCEDURE BorrowBook()

```
  IF isOnLoan = FALSE THEN
    isOnLoan <- TRUE
    OUTPUT "Book borrowed successfully"
  ELSE
    OUTPUT "This book is already on loan"
  ENDIF
ENDPROCEDURE
```

Question 3

(a) uses an inheritance keyword linking Teacher to Employee (e.g. INHERITS) **M1**

(a) CLASS Teacher INHERITS Employee cao **A1**

(a) Answer: CLASS Teacher INHERITS Employee

(b) constructor header PUBLIC PROCEDURE NEW(n, s, subj) with three parameters **M1**

(b) calls the superclass constructor, e.g. SUPER.NEW(n, s), to set name and salary **M1**

(b) assigns subject <- subj after the SUPER call; overall logic complete (ignore minor syntax errors) **A1**

(b) Answer:PUBLIC PROCEDURE NEW(n, s, subj)
SUPER.NEW(n, s)

```
  subject <- subj
ENDPROCEDURE
```

(c) method overriding is when a subclass defines its own version of a method that already exists in its superclass, replacing that inherited behaviour when called on an object of the subclass (oe) **B1**

(c) e.g. Teacher could define its own GetDetails() that also includes subject in the returned string, so calling GetDetails() on a Teacher object uses Teacher's version rather than Employee's (oe) **B1**

(c) Answer: Method overriding is when a subclass provides its own implementation of a method inherited from its superclass, so that version runs instead. For example, Teacher could override GetDetails() to also include subject, so calling GetDetails() on a Teacher object returns extra information not present in Employee's version.

Question 4

(a) polymorphism means objects of different classes can be treated through a common interface/superclass, while each responds to the same method call in its own way (oe) **B1**

(a) here, Circle and Rectangle both inherit from Shape and both override CalculateArea, so both types of object can be treated as Shape objects (oe) **B1**

(a) calling CalculateArea() on a Circle object runs Circle's version, and calling it on a Rectangle object runs Rectangle's version, even though the same method name/call is used (oe) **B1**

(a) Answer: Polymorphism means that objects of different classes can be treated through a shared superclass/interface, but each still responds to a call in its own way. Here, Circle and Rectangle both override CalculateArea from Shape, so calling CalculateArea() on either type of object uses that object's own version of the method, even though the same method name is used throughout.

(b) every object in shapes is treated simply as a Shape, since Circle and Rectangle both inherit from Shape (oe) **B1**

(b) the loop calls s.CalculateArea() the same way for every element, without checking or needing to know its exact class (oe) **B1**

(b) because of polymorphism/dynamic (run-time) method binding, the correct overridden version of CalculateArea for that object's actual class is automatically called (oe) **B1**

(b) Answer: Because Circle and Rectangle both inherit from Shape, every element of shapes can be treated as a Shape and called in exactly the same way with s.CalculateArea(). The program does not need to check each object's exact class, because polymorphism (dynamic method binding) ensures the correct overridden version of CalculateArea runs automatically for whichever type of object it actually is.

Question 5

(a) a function whose return value depends only on its input parameters/arguments (oe) **B1**

(a) and which has no side effects, i.e. it does not change any state/data outside itself (e.g. global variables) (oe) **B1**

(a) Answer: A pure function is one whose result depends only on its input arguments, and which causes no side effects, meaning it does not change any variable or state outside the function.

(b) the function's behaviour is easier to predict/test/debug, because calling it does not depend on, or change, any state elsewhere in the program (oe) **B1**

(b) it makes the function safer to run in parallel/concurrently, since there is no shared state that could be altered by another process at the same time (oe) **B1**

(b) Answer: It makes functions easier to test and reason about, since their output only depends on their own inputs; it also makes them safer to run in parallel, since there is no shared state for two calls to interfere with.

(c) AddToGlobalTotal cao **B1**

(c) it reads and permanently changes the global variable globalTotal, which is a side effect outside the function, so the same call can return a different result each time depending on globalTotal's current value (oe) **B1**

(c) Answer: AddToGlobalTotal is impure, because it changes the external variable globalTotal each time it is called, so the value it returns depends on more than just its own parameter x, and it has a side effect outside the function.

Question 6

(a) "Generic Co: generic vehicle" **B1**

(a) "Skoda: car, 4 wheels" **B1**

(a) "Triumph: motorbike, 2 wheels" **B1**

(a) all three lines given in the correct order matching the order objects were appended to fleet **B1**

(a) Answer:Generic Co: generic vehicle
Skoda: car, 4 wheels
Triumph: motorbike, 2 wheels

(b) each element of fleet is declared/stored as a Vehicle type but actually refers to an object of its real (sub)class, Car or Motorbike (oe) **B1**

(b) when v.Describe() is called, the program does not simply use Vehicle's method because the object's actual class defines its own version (oe) **B1**

(b) this is dynamic (run-time) method binding/dispatch, where the version of the method that is run is decided by the object's actual class at run time, not by the declared type used to store the reference (oe) **B1**

(b) this is exactly what allows polymorphism: the same call v.Describe() behaves differently depending on which class of object v actually refers to at that point in the loop (oe) **B1**

(b) Answer: Each element of fleet is stored using the general Vehicle type, but it actually refers to an object of its true class, Car or Motorbike. When v.Describe() is called, dynamic (run-time) method binding means the version that runs is chosen according to the object's real class, not its declared type, so Car's and Motorbike's own overridden Describe() methods are used. This is exactly what allows the same call v.Describe() to behave differently for each object, which is polymorphism in action.

Question 7

(a) **L1** Basic, isolated points are made about procedural and/or object-oriented programming, with little or no reference to the grade-tracking scenario. The answer may simply define the paradigms with no comparison. **1-2**

(a) **L2** Some relevant points of comparison are made between procedural and object-oriented approaches (e.g. data protection, code reuse), with limited application to the specific scenario of a school system extended over time by multiple developers. **3-5**

(a) **L3** A clear, well-balanced comparison is given, correctly weighing at least two relevant factors (such as encapsulation/data protection, code reuse via inheritance, and maintainability by multiple developers over time), with a reasoned overall judgement clearly applied to the school grade-tracking scenario. **6-8**

(a) Indicative content:

- A procedural approach would store student data in arrays/records and use separate procedures/functions (e.g. CalculateAverage, AssignGrade) to act on that data; this can be quick to write initially but the data is not protected, so any procedure anywhere in the program could accidentally corrupt a student's marks.
- An object-oriented approach would define a Student class encapsulating each student's own name, subjects and marks as private attributes, with public methods (e.g. AddMark, CalculateAverage) controlling how that data is read or changed, reducing the risk of accidental corruption.
- Inheritance in OOP would allow, for example, a SixthFormStudent subclass to reuse and extend the base Student class (adding UCAS-related attributes) without duplicating existing code, whereas a procedural approach would likely need separate, duplicated procedures or extra conditional branches for different student types.
- Because the system will be extended over several years by different developers, OOP's encapsulation makes it easier for a new developer to safely add features (e.g. attendance tracking) as a new class or subclass without fully understanding or risking every existing procedure, whereas procedural code with shared global data can be harder to safely extend as the system grows (increased coupling).
- A counter-point: for a small, simple, one-off program the procedural approach may be quicker to write and easier for a single developer to follow, since OOP's class design adds some initial overhead; the choice depends on the scale and expected lifetime/team size of the project.
- Overall judgement: given that this system is expected to grow over several years with multiple developers, the object-oriented approach is generally more appropriate, mainly due to better data protection (encapsulation) and easier code reuse/extension (inheritance and polymorphism).

(a) Answer: Best answers weigh encapsulation/data protection and reuse via inheritance (favouring OOP for a multi-year, multi-developer school system) against the simplicity of procedural code for small one-off tasks, reaching a reasoned overall recommendation for this scenario.

Question 8

(a) ambiguity can arise if two superclasses each define a method (or attribute) with the same name (oe) **B1**

(a) it becomes unclear which of the two inherited versions the subclass should use, which version is called can depend on the language/implementation, and this can lead to unexpected/inconsistent behaviour (oe) **B1**

(a) this is sometimes called the 'diamond problem', where the same ultimate ancestor is reached via two different inheritance paths (oe); accept any other valid problem, e.g. increased design complexity and difficulty tracing where behaviour originates **B1**

(a) Answer: If Person and VehicleOperator both defined a method with the same name, it would be ambiguous which version DeliveryDriver should inherit, potentially causing unexpected or inconsistent behaviour; this kind of clash is sometimes called the diamond problem, and it also makes the class hierarchy harder to understand and maintain.

(b) a change made to a superclass near the top of the hierarchy can unexpectedly affect the behaviour of every subclass beneath it (sometimes called the 'fragile base class' problem) (oe) **B1**

(b) it can become difficult for a developer to trace which class in a long chain actually defines/overrides a given method's behaviour, making the code harder to understand and maintain (oe) **B1**

(b) classes lower down become tightly coupled to design decisions made far above them, making the hierarchy harder to safely restructure or extend later (oe); accept any other valid, well-explained problem **B1**

(b) Answer: Changing a superclass high up in a long inheritance chain can unintentionally break or change behaviour in many subclasses beneath it, which is known as the fragile base class problem; deep hierarchies can also make it hard to trace which class actually defines a given method's behaviour, and they create tight coupling that makes the design harder to safely change later.

Question 9

(a) **L1** A basic outline is given of some classes or attributes, with little structure, no working inheritance/polymorphism, and little or no reference to encapsulation or to why OOP suits this scenario. **1-3**

(a) **L2** A reasonably structured design is given, including a base class and at least one subclass, with some correct use of encapsulation (e.g. private attributes) and at least a basic attempt to link the design to the future-extension requirement, but polymorphism and/or validation are incomplete or only partly justified. **4-6**

(a) **L3** A well-structured, coherent design is given: a clear base class (e.g. FitnessClass) with at least two subclasses showing genuine overridden/polymorphic behaviour, correctly encapsulated attributes with sensible validation, and a clear, well-reasoned justification, referring specifically to future extension by new class formats, of why the object-oriented approach is appropriate here. **7-10**

(a) Indicative content:

- A base class FitnessClass could hold shared private attributes such as className (STRING), instructorName (STRING) and maxParticipants (INTEGER), with a constructor that validates maxParticipants is greater than 0 (e.g. outputting an error and refusing to set it otherwise), and a public method such as CalculateCapacityRemaining() giving common shared behaviour.
- A subclass PersonalTrainingSession could inherit from FitnessClass and override a method such as CheckAvailability() to simply check whether the one participant slot (maxParticipants = 1) is already booked, returning TRUE or FALSE.
- A subclass GroupFitnessClass could inherit from FitnessClass and override CheckAvailability() differently, comparing the current number of bookings against maxParticipants (which may be much larger, e.g. 20 for a spin class), returning TRUE only if there is a free space.
- Encapsulating attributes (e.g. maxParticipants, currentBookings) as PRIVATE, accessed only via validated public methods (e.g. a BookPlace() method that refuses a booking if the class is already full), protects the booking data from being corrupted by code elsewhere in the system, for example accidentally allowing more bookings than maxParticipants.
- Because both subclasses inherit shared attributes and behaviour from FitnessClass, adding a further future class format (e.g. an online-only class with different capacity rules) would only require writing one new subclass that reuses the existing base class, rather than rewriting shared logic such as instructor details or basic validation from scratch.
- Because CheckAvailability() (and similar methods) are called polymorphically wherever a list of mixed FitnessClass objects is processed (e.g. when displaying all of today's classes with their availability), new class formats can be added in future without needing to change the code that processes and displays the list, since each object automatically uses its own overridden method.
- Overall justification: the object-oriented approach is well suited here because encapsulation protects booking data from invalid states, and inheritance plus polymorphism mean the system can be extended with new class formats over time while reusing existing, tested code and without having to modify the existing processing logic each time a new format is added.

(a) Answer: Strong answers define a FitnessClass base class with encapsulated, validated attributes (e.g. maxParticipants), at least two subclasses (e.g. PersonalTrainingSession and GroupFitnessClass) overriding a method such as CheckAvailability() with genuinely different logic, and a clear justification that inheritance/polymorphism let new class formats be added in future years by writing new subclasses rather than modifying existing, shared, tested code.