AQA Computer Science A-Level

4.1.2 Programming paradigms

Advanced Notes

Specification:

4.1.2.1 Programming paradigms:

Understand the characteristics of the procedural and object-oriented programming paradigms, and have experience of programming in each.

4.1.2.2 Procedural-oriented programming:

Understand the structured approach to program design and construction.

Be able to construct and use hierarchy charts when designing programs.

Be able to explain the advantages of the structured approach.

4.1.2.3 Object-oriented programming:

Be familiar with the concepts of:

- class
- object
- instantiation
- encapsulation
- inheritance
- aggregation
- composition
- polymorphism
- overriding

Know why the object-oriented paradigm is used.

Be aware of the following object-oriented design principles:

- encapsulate what varies
- favour composition over inheritance
- program to interfaces, not implementation

Be able to write object-oriented programs

Be able to draw and interpret class diagrams

The procedural programming paradigm

Programs written in the procedural programming paradigm are formed from sequences of instructions that are executed in the order in which they appear. Procedures like functions and subroutines form parts of the program and can be called from anywhere within the program, by other procedures or recursively.

Recursion

The process in which a block of code that is defined in terms of itself makes a call to itself.

Data is stored in procedural programs by constants and variables. A data structure is said to have a global scope if it can be accessed from all parts of the program and a local scope if it is only accessible from the structure within which it is declared.

Most of the programs that you may have written are likely to have been procedural.

Synoptic Link

Assignment, sequence, selection and iteration are defined in the notes for **abstraction and automation** under **theory of computation**.

The structured approach

Using the structured approach to program design and construction keeps programs easy to understand and manage. Four basic structures are used: assignment, sequence, selection and iteration.

Structured programs are said to be designed from the top down, meaning that the most important elements of a problem are broken down into smaller tasks, each of which can be solved in a block of code such as a procedure or module which goes on to form part of the overall solution.

Designing a program from the top down makes maintaining the program easier as navigation of different elements of the overall solution is improved. If all of a program's code is contained within the same module, finding the specific line of code that needs fixing can be incredibly difficult - especially in large projects.

When a program is split into modules, testing can be carried out on the individual modules before they are combined to form the overall solution. Furthermore, development can be split over a team of developers each of which is assigned a different module to work on.

Hierarchy charts

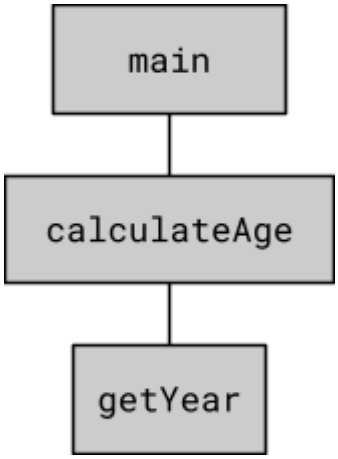
A hierarchy chart graphically represents the structure of a structured program. Each procedure is displayed as a rectangle which is connected to any other procedures that are used within it.

Example

```
SUBROUTINE Main()
  name ← INPUT
  yearBorn ← INPUT
  age ← calculateAge(yearBorn)
  IF age > 17 THEN
    OUTPUT name + "can drive"
  ELSE
    OUTPUT name + "can't drive"
  END IF
END SUBROUTINE
```

```
FUNCTION calculateAge(year)
  yearNow ← getYear()
  age ← yearNow - year
  RETURN age
END FUNCTION
```

```
FUNCTION getYear()
  RETURN system.year
END FUNCTION
```

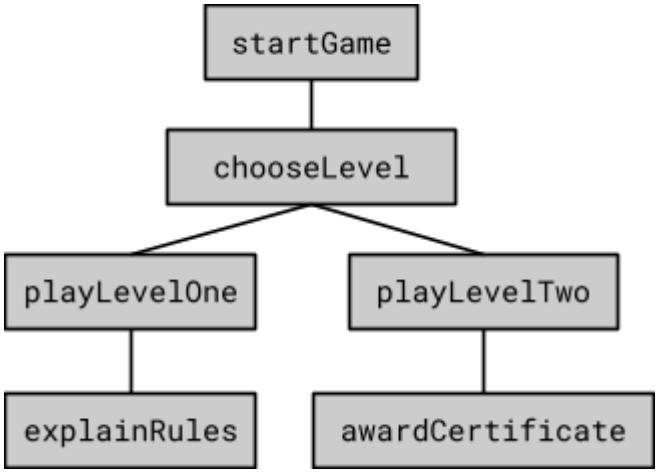


The three procedures above (`main`, `calculateAge` and `getYear`) form a program which could be represented by the hierarchy chart on the left.

Each rectangle represents a part of the overall program. The lines between the rectangles show the relationships that exist between the different parts of the program.

In more complicated programs, each rectangle can be linked to more than one other. This occurs when a procedure calls more than one other procedure.

The hierarchy chart below represents a more complicated program:



The hierarchy chart above shows that the procedure `startGame` calls `chooseLevel` which in turn calls both `playLevelOne` and `playLevelTwo` and so on.

The object-oriented programming paradigm

Objects

Rather than storing data with constants and variables like procedural programs, object-oriented programs use objects as containers of both data and instructions.

New objects are created from classes. The process of creating an object is called instantiation. It is possible for one program to have many different objects of the same class which will all have identical methods and procedures, however the actual values of their properties are unique to the object. For example, a program could have three different objects of the class `car` which all have a property called `manufacturer`. All three cars could have different `manufacturers`.

Classes & class definitions

An object is defined as an instance of a class. A class is like a blueprint for objects, they specify what properties (data) and methods (instructions) objects of their type will have.

For example, a class for creating car objects might have the properties `EngineSize` and `Colour` and the methods `StartEngine`, `ApplyHandbrake` and `TurnOnLights`.

A class can be expressed on paper as a class definition which lists a class' name, properties and methods. Exam questions often ask you to write or interpret class definitions.

The class definition for the car class could look like this:

```
Car = Class {
  Private:
    Manufacturer: String
    Model: String
    EngineCapacity: Float
    IsTaxed: Boolean
  Public:
    Function GetManufacturer
    Function GetModel
    Function GetEngineCapacity
    Function GetIsTaxed
    Procedure SetDetails
}
```

A method or property that is listed as private can only be accessed from within an object. Public methods allow an interface for accessing and modifying a class' private properties.

Encapsulation

Encapsulation is the name given to the process of combining methods and procedures to form an object in object-oriented programming. An object is said to encapsulate its contents, forming a single entity which encompasses all of the object's properties and methods.

Encapsulation allows the development of large programs to be split across a team of developers, each of whom can be allocated a class to develop. As long as a developer knows the methods which belong to other classes, they can develop their own class without knowing the specific implementation of methods in other classes.

Synoptic Link

This idea is essential to the design principle **program to Interfaces, not Implementation** which is covered later in these notes.

For example, suppose `MyCar` is an object of the class `Car`. The code `MyCar.StartEngine()` can be written by a developer working on another part of the program, and the developer doesn't need to know exactly how the `StartEngine` procedure works, just that it starts the car's engine.

Inheritance

A class can inherit another class. Inheritance allows one class to share the properties and methods of another class, while having its own properties and methods too.

```
DeLorean = Class (Car) {
  Private:
    ReactorOutput: Integer
    FluxCapacitorInput: Integer
  Public:
    Function GetReactorOutput
    Function GetFluxCapacitorInput
    Procedure SetDetails (Override)
}
```

Inheritance can be described as an "is a" relationship: a DeLorean **is a** car.

The example above shows the class description for `DeLorean`. The brackets after class (shown in red) indicate that the class inherits the `Car` class and therefore has all of the properties and methods that the `Car` class has as well as its own properties and methods that do not exist in the `Car` class such as `ReactorOutput`.

Polymorphism

The word polymorphism comes from the Greek for “many forms”. Polymorphism occurs when objects are processed differently depending on their class.

For example, if two objects with classes `Car` and `Lorry` both had a `ConsumeFuelForOneMile` method, it might decrease the Car’s `FuelLevel` property less than it would the Lorry’s. The method has taken the object’s class into account and treated them differently based on their class.

Overriding

An overridden method has the same name as a method in an inherited class but different implementation. The word override after the `SetDetails` method in the class description above indicates that the implementation of the `SetDetails` method in the `DeLorean` class differs from the implementation of the method with the same name in the `Car` class.

Association

If two objects are associated, they can be described as having a “has a” relationship. For example, objects of the classes `Car` and `Driver` could be associated as a car **has a** driver. An associated object forms part of its container object as a property. Just like a property of an object can be a string, it can be an object of another class.

There are two specific types of association that you need to know about: aggregation and composition.

Aggregation is the weaker of the two kinds of association. When an object is associated with another by aggregation, it will still exist if its containing object is destroyed.

> *For example: a car’s passengers are associated with the car by aggregation. If the car is scrapped, the passengers still exist independently.*

Composition is a stronger relationship between classes. If two objects are associated by composition and the containing object is destroyed, the associated object is also destroyed.

> *For example: a car’s wheels are associated with the car by composition. The wheels form an integral part of the car and scrapping the car destroys the wheels.*

Why is object-oriented programming used?

Object-oriented programming provides programs with a clear structure that makes developing and testing programs easier for developers. Using the paradigm also allows for large projects to be divided among a team of developers.

The use of classes allows code to be reused throughout the same program and even in other programs. This improves the space efficiency of code.

Object-oriented design principles

There are three design principles used in object-oriented programming that you need to be aware of. These are: encapsulate what varies, favour composition over inheritance and program to interfaces, not implementation.

Encapsulate what varies

When designing a program in the object-oriented paradigm, any requirements which are likely to change in the future should be encapsulated in a class so that any changes can be easily made when required.

Favour composition over inheritance

Wherever possible, composition should be used over inheritance. This is because composition is seen as a more flexible relationship between objects. Composition isn't always appropriate and inheritance should still be used in these situations.

Program to interfaces, not implementation

This design principle allows unrelated classes to make use of similar methods. An interface is defined as a collection of abstract procedures that can be implemented by unrelated classes.

When a new object is created, it can implement an interface which provides it with the correct properties and methods.

Writing object-oriented programs

The AQA specification requires you to have practical experience of coding user-defined classes, involving:

- abstract, virtual and static methods
- inheritance
- aggregation
- polymorphism
- public, private and protected specifiers.

We recommend that you use online documentation to find out what the syntax for this looks like in the programming language that you will use when you sit the exam.

Abstract, virtual and static methods

Abstract methods are declared in abstract classes and do not have an implementation. They serve as a blueprint for methods that must be implemented by subclasses. Abstract methods enforce a contract for subclasses to provide specific method implementations, ensuring consistency across different subclasses.

Virtual methods are methods in a base class that can be overridden by derived classes. This allows subclasses to provide their specific implementation while maintaining a consistent method interface. Virtual methods facilitate polymorphism, enabling method behaviour to vary based on the object that invokes the method.

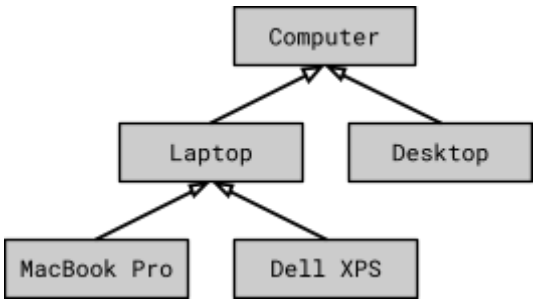
Virtual methods belong to the class itself rather than any instance of the class. They can be called on the class directly and do not require an instance to be invoked.

Class diagrams

Class diagrams can be used to visually represent the relationships that exist between classes. Classes are represented by boxes with different connectors representing different kinds of relationship.

Inheritance diagrams

A special type of class diagram, the inheritance diagram, shows the different inheritance relationships that exist between classes in an object-oriented program.



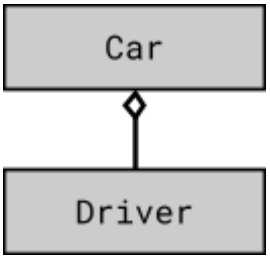
Inheritance is shown with unfilled arrows which point from an inherited class towards the class which inherits it. In this example, the `Laptop` class inherits the `Computer` class so an arrow points upwards from `Laptop` to `Computer`.

Inheritance arrows should always point upwards.

Association

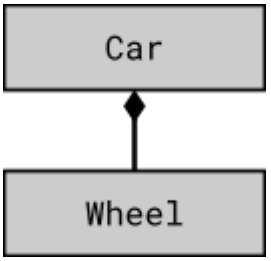
Association is shown in class diagrams with diamond headed arrows.

Aggregation



Aggregation (the weaker of the two types of association) is shown with an unfilled diamond headed arrow.

Composition



Composition (the stronger relationship) is shown with a filled diamond headed arrow.

Class diagrams with properties and methods

A class diagram can contain more information about classes than just their name. When this is done, a class is represented by three boxes. The uppermost box contains the class name, the middle box contains the class' properties and the bottom box contains the class' methods.

Student
- Name: String - Age: Integer
+ GetName + GetAge + SetDetails

A plus sign indicates that a property or method is public and a minus indicates that the property or method is private. A pound symbol (#, not £) indicates that a property or method is protected.

Protected properties and methods are accessible from within the object that declares them (just like if they were private) and also from any inherited objects.