A2 Computing : CPT4 - all you need about OOP

The AQA 2006 specification gives:

Object-oriented programming.
Candidates should be familiar with the concept of an object, an object class, encapsulation, inheritance, polymorphism, associations and containment (fixed and variable aggregation), and event-driven programming.

This has to be studied in the context of the structured approach to programming we looked at in the AS year. *REVISE STRUCTURE DIAGRAMS, including interfaces.*

Structured Programming approach

- The method studied is called *Top-down design*, also known as *Stepwise refinement* or *Functional decomposition*.
- This method involves:
  - Split the (large) problem into a series of smaller problems.
  - Work on each of these separately, splitting them into smaller problems,
  - - until the smaller problems are small enough to be easily solved.
  - The small problems are coded as procedures or functions.
  - If a procedure or function requires data from elsewhere, then that data is passed via parameters.
- It has the characteristics that
  - data and procedures remain separate
  - the programmer makes the relationship between data and procedures by parameter passing
  - Top-down design focuses on the *tasks* that the code will perform, rather than the structure of the data to be processed.
  - Although the coding is modular, the modules are produced specifically for a particular problem solution. Re-use of the code is possible, but it may need modification and re-testing in a new situation.

Object Oriented Programming approach

Whilst the structured approach helps to reduce errors and increase reliability by allowing the programmer to focus on small tasks, it does make re-use of code difficult. Programmers need to recode similar problems from scratch if using this method. OOP's biggest advantage is that it aims to make code reusable. It does this by encapsulating a set of data items together with code to process that data into one item called an *object*. The data items can be processed *only* by using the encapsulated code - they are inaccessible by any other method. So once such code has been tested, it is known to be reliable, and will not need recoding in the future. The data and procedures are now firmly linked as an object. A programmer can use an object within any number of new programs without having to worry about how the processing is done - the object's code is already written and can be immediately used. Furthermore, a programmer needing a variation on an existing object can simply create a new object based on the existing one, inheriting from it all the data items and procedures that don't need changing (this is re-used) but allowing the addition of new data items and procedures.

Advantages of OOP over structured approach to program design include:

- it produces reusable code/objects because of encapsulation and inheritance.
- the data is protected because it can be altered only by the encapsulated methods.
- it is more efficient to write programs which use pre-defined objects.
- the storage structure and/or procedures within an object type could be altered if required without affecting programs that make use of that object type.
- new functions can easily be added to objects by using inheritance
- the code produced is likely to contain fewer errors because pretested objects are being used.
- less maintenance effort will be required by the developer because objects can be reused.

Many CPT4 papers have asked for 2, 3 or 4 of these points.

Encapsulation
is the combining of data items (or attributes / properties) with procedures (methods) for manipulating those data items, into a single unit called an object.

Object
An object consists of a data structure plus a set of procedures for manipulating that data structure.
The data can be manipulated only by those procedures contained within the object.

Class (Object class / Object Type)
A class is a template defining a set of objects which share a common data structure (set of attributes / properties) and a common behaviour (set of methods).

It follows from these definitions that a *class* is an object type, that is, it is the template which defines the structure of the data within an object, and contains the code for manipulating that data. An *object* is actually an instance of such a type. (Compare a `type TStudent = record` declaration, which would be followed by a list of field names plus types. This is a template for a student record user-defined data structure. However, `var student1, student2 : TStudent;` declares variables student1 and student 2 to be record variables, each created according to the TStudent template - student1 and student2 are each instances of type TStudent. In the same way, `type TButton = class` defines a class - an object type, as a template, then `var button1:TButton` will declare button1 as an object, an instance of TButton).

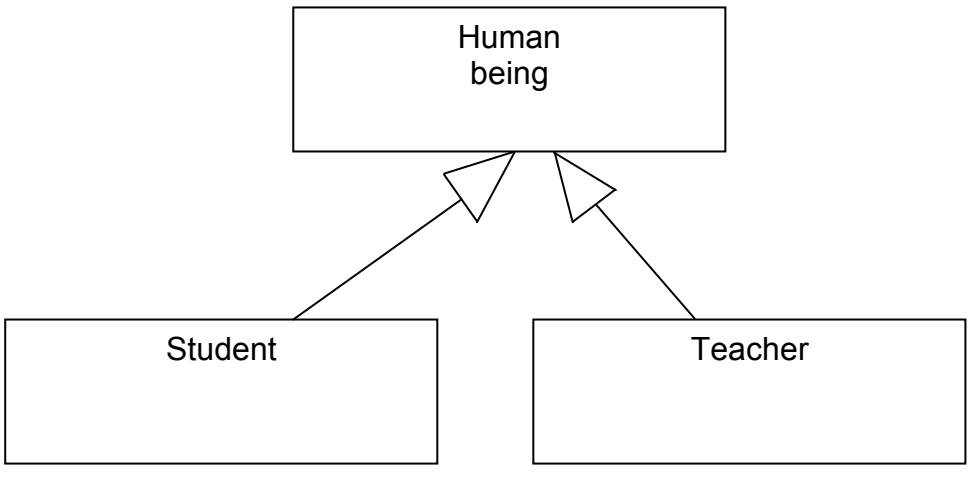
Inheritance
Inheritance is a relationship among classes/objects wherein one class/object shares the structure and/or behaviour defined in one (single inheritance) or more (multiple inheritance) other classes/objects.

alternative definition:

Inheritance is defining an object then using it to build a hierarchy of descendant objects, with each descendant inheriting access to all its ancestors' code and data.

This again allows code already created to be reused in a situation where objects are needed which are similar to or derived from previously created objects – it makes program development easier because material is being reused.

Inheritance can be depicted by using *inheritance diagrams*, e.g.



Notice with an inheritance diagram that:

- it depicts an *"is a"* relationship – a Student is a human being, a Teacher is a human being (?!). You could in fact label the arrows with the words "is a" to make this point clearly.
- The parent goes at the top, with descendants below.
- The link is an arrow pointing (upwards) towards the parent.
- The arrow head is an open triangle. (This conforms to the latest UML – unified modelling language – ideas, replacing a blocked-in triangle. AQA requires an arrow, pointing the correct way, but will not penalise you for using the wrong type of arrow).
- the diagram above shows inheritance of objects. An object is depicted by a rectangle containing the name of the object. A class is depicted by a rectangle divided into three sections. The top section contains the class name; the middle section contains its attributes and the bottom section its methods.

Most CPT4 papers have asked for an inheritance diagram. This could be an easy 2 or 3 marks.

Polymorphism
Polymorphism is the ability of an inherited class to customise an inherited method. Two different classes may inherit a particular method from a common parent, but they may re-define it in different ways.

alternatively

Polymorphism is giving an action one name that is shared up and down a class hierarchy, with each class in the hierarchy implementing the action in a way appropriate to itself.

For example, an "animal" is a class, one of whose methods is "eat". A bird is a descendant of class "animal", and will inherit its "eat" method. A "mammal" is also a descendant of "animal", and will also inherit its "eat" method. However, a mammal eats in a manner which is different from that of a bird, so the "eat" method would have to be implemented in a way appropriate to each descendant. We have an example of an action with the name "eat" which is shared up and down the class hierarchy, but re-defined in a different way. "Eat" is an example of a *polymorphic* method. (You could continue this – human and dog are descendants of mammal – and also will need to redefine their eat methods). [*Polymorphic* means "many shapes or forms (Greek)].

Object Oriented Programming Languages
For a programming language to be classified as object oriented, it must satisfy three requirements:

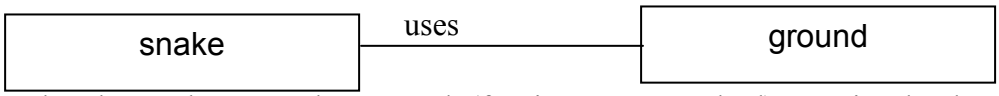
1. the language must support objects that are data structures, with an interface of named operations. (i.e. it must support encapsulation).
2. each object must have an associated class. (i.e. the language must support type declarations of classes, allowing objects to be created as instances of a class)
3. the language must support the creation of new classes which can inherit attributes and methods from existing classes (i.e. it must support inheritance).

(N.B. Some sources don't include requirement 2 above, but instead point out that the language must support the re-definition of inherited methods, i.e. must support polymorphism)

Relationships between classes
We have already come across inheritance as a particular type of relationship between different classes (a generalisation/specialisation relationship = "is a"). There are others which are on the CPT4 specification. Note that this is an area of Computing where the terminology means different things to different people. For CPT4 purposes, beware what you read!

Association
This is basically a link between objects, which allows *messages* to be sent between them, so that they can interact. Pictorially, an association is shown as a simple line between the object or class symbols. It may be

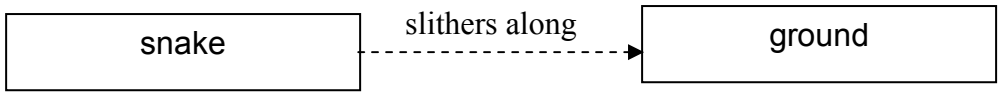
a one-to-one or one-to-many association (use 1 for one, N for many). It may be a *dependency*, also called a *uses* relationship. E.g.



In this example, the snake uses the ground (for its move method). It is clearly not an inheritance relationship, nor is it a containment relationship (see below).

Another example could be that of a person - pet relationship - the person owns/uses/keeps the pet, the pet is dependent on the person, but the pet is not a descendant of the person, and is not a component part of the person - it's a dependency relationship, and, in this case, also a one to many relationship.

The label "uses" on the association line is generally applicable - it's a uses relationship. However, you could label the association with a more specific verb, e.g. (for the snake example) "slithers along". If so, the direction of the association could also be indicated by using an arrow:

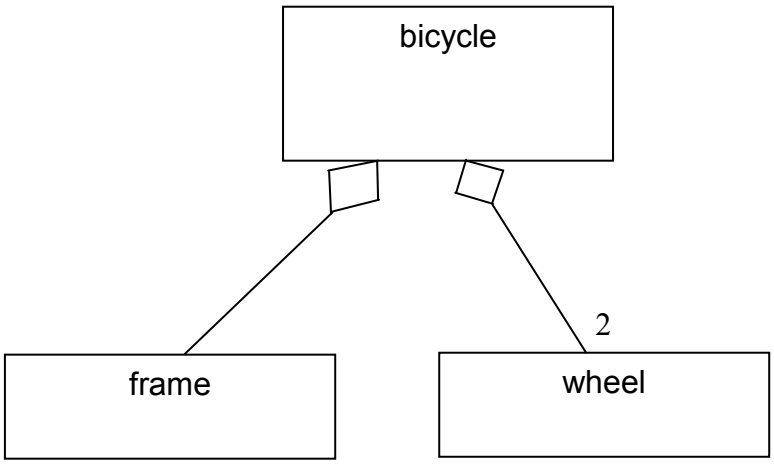


Containment

This is a relationship between classes/objects where one class is composed wholly or partly of other classes. A Form is a container object. When you place components (which are themselves objects) on a form, the new form created exhibits *containment.* Containment is a "has a" relationship.

- **Fixed Aggregation** is a form of containment where the number and identity of the contained objects is always the same. The attributes of an object showing fixed aggregation may be declared as other objects; memory requirements for such an object would be known at design time. Fixed aggregation is also known as *Composition*.
- **Variable Aggregation** is a form of containment where the number and identity of the contained objects may vary at runtime. In this case, the attributes of an object showing variable aggregation would include pointers to the contained objects rather than the actual objects; such pointers can be set to NULL.

Containment signifies a "has a" relationship. A form has a button, a form has an edit box, a car has an engine etc. Pictorially, containment is indicated by a diamond-shaped arrow head, and the arrow points towards the container object.



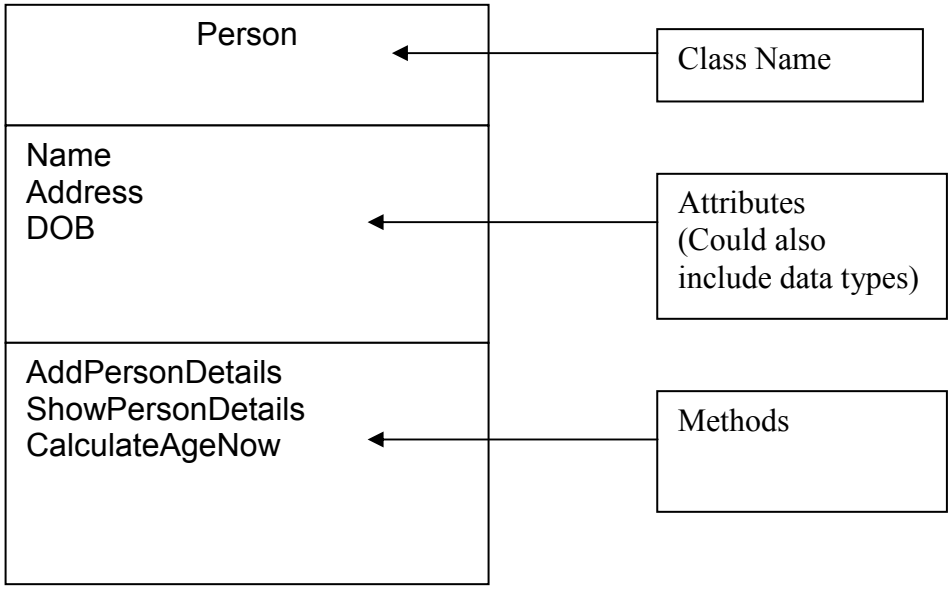
(Note that aggregation, association, containment have recently been added to the CPT4 specification; older papers will not have questions on this area).

Object oriented analysis (this may also be relevant to CPT6 if you choose to use OOP in your project - highly recommended if appropriate).

1. Determine the objects in the domain of the problem.
2. Determine the relationships between the objects. Show these using inheritance, association and containment diagrams.
3. Determine the attributes and behaviours of each object.
4. Draw the object-oriented analysis diagrams. This includes class diagrams.

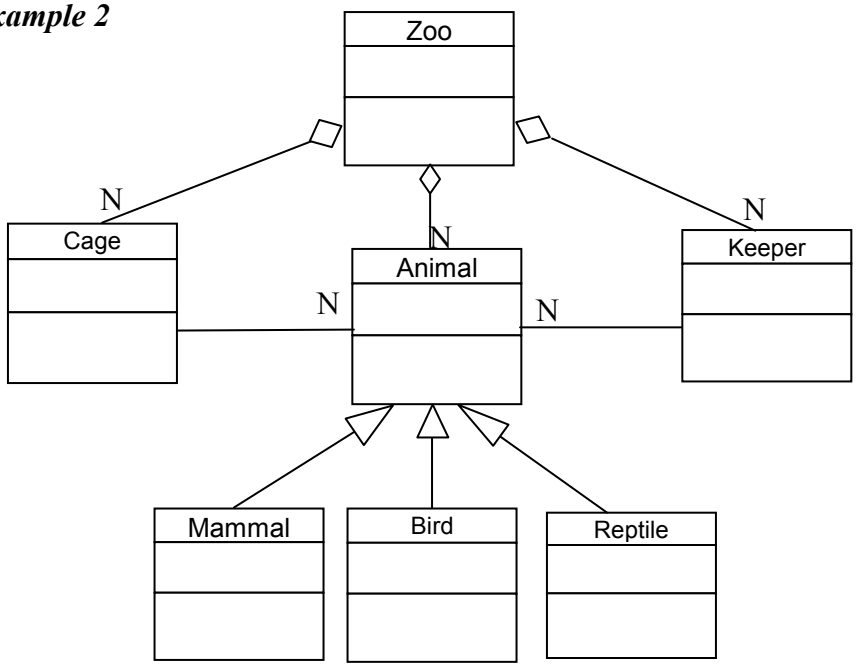
Class Diagrams

Example 1



Note that "Person" is an appropriate class name in general, but if you are programming in Delphi, for instance, you ought to subscribe to the convention that type names begin with T. A class is a type (an object type) so it would better be "TPerson". A reminder also that the methods are the publicly available procedures (behaviours) which manipulate the internal attributes (data fields). The attributes ought to be declared as *private*, i.e. they can't be directly accessed by users of the class - they must be accessed through the methods. This is essential in order to allow for reuse of the code and data structures without re-testing. It also follows that the methods can't be used to manipulate data outside the object. A book class in a library can have a ReadTitle method (this accesses the internal "title" field), but a LoanBook method would not be possible if such a method referred to a borrower (it's not an attribute of Book).

Example 2



A zoo contains animals,cages, keepers.
A keeper looks after many animals.
A cage houses animals.
A mammal / a bird / a reptile is an animal.

This diagram does not show the attributes or methods, but it could.

Class Definitions
These would be written in the design stage. A class definition declares:

- the name of the class
- the parent class from which it is inherited (or says that it is a base class)
- the attributes of the class. These would normally be private, so the keyword `private` would be expected also.
- the methods to be made available (including input and output parameters). These would be `public` since they are the procedures to be used by programmers which will access the attributes. By making the attributes private, you are ensuring that the only way they can be accessed is via the public methods.

> Items declared as **private** cannot be seen / referred to except in the unit that actually implements the class. In other words, when you subsequently use a class in another unit, you can't refer to private items. Items declared as **public** on the other hand are available for use - they are part of the visible interface seen by any program using the class.

Class definitions are probably best written in the format you would actually use in a Delphi program, e.g.

```
type Tcounter = class                                   <-- parent class
    public
     procedure initialisecounter(startvalue:integer);   <-- methods
     procedure incrementcounter;                        <-- methods
     function getcounter:integer;                       <-- methods
    private
     MyCounter : integer;                               <-- attributes
  end;

type Tcounterbeep = object(Tcounter)                    <-- inheritance
    public
     procedure incrementcounter;                        <-- polymorphic method
  end;
```

(In later versions of Delphi, and for more complex work such as using OOP as the fundamental programming method in your project, you may need to learn about constructors, destructors, virtual methods and overriding. This is well documented in various books and online tutorials, but is unlikely to appear in CPT4 questions).

Event-driven programming
Simple Pascal programs are basically a sequence of instructions which run from start to finish, when the program terminates. With event-driven programming, a program consists of (many) procedures which are effectively "event handlers". An event is an occurrence resulting from some user interaction, a system happening, or particular code logic. The occurrence of a particular event causes its associated procedure to be run. Thus in event-driven programming, the order of execution of procedures is determined by the order of the events, not by the order in which the procedures are written. OOP and event-driven programming go together, because each object can have its own events, e.g. a button will have an OnClick event, etc. Objects communicate using events; an event causes a message to be produced, which is converted to a procedure/function call.

Windows operating system is basically a collection of objects which communicate in this way via messages. Many Delphi components provide "wrappers" for Windows event handlers; Delphi itself allows programmers to access the Windows Application Programming Interface conveniently.