Object Oriented Programming – Definitions for Quick Reference

A Class

A class can be thought of as a software blueprint for implementing objects of a given type; therefore a class can be defined as a template that is used to create objects of its type, which specifies the data fields (properties) and methods (procedures and functions) that objects will have.

An Object

An object is simply known as an instance of a class. It has a collection of data fields (properties) and the methods (procedures and functions) that operate on these data fields.

Encapsulation

Encapsulation is the ability to hide or isolate the internal components (data) and operations (methods) of an object, allowing the user only to communicate through the public interface. Encapsulation is a key principle of the OOP methodology. If an object is properly encapsulated, it is said to be a "black box" that reveals just the information needed to reuse its members through its public interface, and that hides any private implementation details.

Inheritance

This is the relationship among classes wherein one class shares the data structure and behaviour of another. In other words, a class has the same properties and methods as its parent class, and so it inherits the data fields (properties) and the methods of the parent.

Polymorphism

This is a term derived from the Greek for "many forms". In OOP it refers to the programming language's ability to process objects differently depending on class or data type.

Inheritance based polymorphism - describes a situation when a method in the base class is inherited in the derived class, but is redefined to suit the data and purpose of the derived class.

Overriding methods will allow a derived class to alter the behaviour that is inherited from the base class, but they maintain the same method call and signature. (In other words, its return type and parameter list must stay the same but the algorithms can be different).

Interface based polymorphism- this can be achieved by creating an interface and implementing it differently in several classes. Therefore a method from the interface can be invoked using the same call but the algorithms can be different.

In most OOP languages, the correct method is chosen because method calls are determined by the actual object and not the object reference.

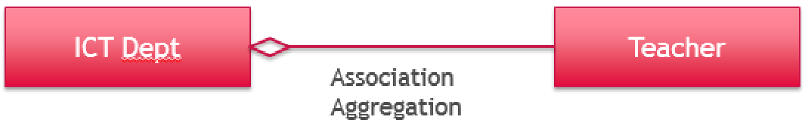
NOTE : Method **Overloading** is when two or more methods in the same class have the same name but a different parameter list.

Aggregation

This is when an object may be built up from other objects, and is generally based on the way in which objects are related.

Association Aggregation is when an object which contains other objects is destroyed, the other objects will still exist.

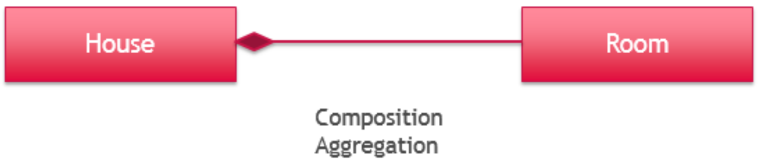
For example, if we have a teacher who belongs to a department say ICT, and the ICT department is destroyed, there is still a teacher object which exists.



Composition

This is when an object that contains other objects is destroyed; then the objects will no longer exist.

For example if we have a house object which contains room objects, if we delete the house then the rooms can no longer exist.



Advanced Design Principles

The following design principles are recognised as producing object oriented programs which produce more elegant solutions to problems.

Encapsulate What Varies
This is the idea that where there is a variance, then there should be a class for it.
This is based on encapsulation and information hiding, so that the properties and methods are appropriate to the real world scenario they reflect.

Favour Composition over Inheritance
Composition allows classes to instantiate other classes within them to achieve the desired functionality, as opposed to using inheritance**.** *Favour composition over inheritance* is the design principle in which composition is preferred, because this is considered to be a more flexible relationship between two classes and is more readily adapted in the future.

NOTE : Generally we can describe composition (when objects contain other objects) as a “has-a” relationship. Inheritance is typically described as an “is-a” relationship.

Classes making use of composition are thought to be easier to maintain and can be tested separately. This is because if inheritance is used, a base class cannot be tested separately from the derived class, but composition allows each class to be tested separately.

Classes are often easier to understand in this format and this approach can also be less prone to errors. So, with this principle, whilst classes are related and can use each other’s instance variables, properties and methods are not inherited. This can help prevent potential errors in altering the base class methods.

Program to interfaces, not implementation
This principle allows programs to be written based on the interface as opposed to each individual implementation of a class. This allows us to alter individual classes, perhaps updating a method or adding additional functionality. However, this is done with reference to the interface, and therefore has little or no effect on the other classes in the program.