# Unit 1a: Object Orientation

## Contents

## 1. Objects

From a modelling point of view an object is something of interest to a business system or application which captures state and behaviour. State represents what we know about an object i.e. its attributes or data; while behaviour is the procedures or methods which the object responds to i.e. actions which query or change attribute values.

### a. Example 1

An Employee object would require to know its employee number, name, job title, etc. The Employee object would also require to respond to queries for some, or all, of its data values, and requests to change those values.

Modelling real world objects involves making decisions about the aspects of the object which are important for the business system or application i.e. we wish to abstract state and behaviour which is important and ignore irrelevant features i.e. not relevant to the business system or application being modelled.

### b. Example 2

The Employee object would model a real life person for whom characteristics such height and weight might be important in other contexts but would not be relevant to modelling the employee in an employment context.

Of course, objects rarely stand in isolation and have relationships with other objects in the system – in fact, an object may be associated with or composed of other objects.

### c. Example 3

The business system may require Department objects where each Department object is associated with many Employee objects. We might also model an Employee’s address attribute as an Address object.

An OO application involves creating objects which invoke methods of other objects to query/modify attributes.

## 2. Classes

Modelling objects is really about identifying templates which allow applications to create object instances each of which holds its own data.

A class definition is used to identify fields (attributes) and methods which each object instance of that class has. Therefore, an object instance, of a class, holds its own data values for the attributes (fields) identified in the class definition.

### a. Example

An Employee class would specify the name and type of required attributes e.g. number, name, job title etc.; while an Employee object would follow the rules of the Employee class and hold actual values for that Employee e.g. EMP001, Sergi Busquets, Project Manager etc.

An OO application involves: creating objects from class definitions and allowing these objects to invoke (execute) methods of other objects to query/modify their fields.

Most OO languages, e.g. Java, are not pure and incorporate concepts from other paradigms such as:

- Sequence, Selection, Iteration;
- Modularisation.

## 3. Object Oriented Concepts

Object orientation contains a number of other important concepts.

### a. Data Encapsulation

This is about hiding the object's state from the rest of the system/application. Actually it's about controlling access to the attributes of an object instance. Ideally this should involve attributes being **private** to the object and allowing access/changes to attribute values only through the use of the object's methods. This has the advantage of controlling how data values in a business system can be retrieved/amended, and, additionally, it means that the way in which attributes are implemented can be hidden from the rest of the system and, therefore, can be changed with minimal knock on effects for the functioning of the application or other objects.

### b. Abstraction

In general, this is about reducing complexity by focusing on essential features. We have already seen that a class definition only defines attributes and methods which are required for the business system or application. However, we can take this a little further and recognize that similar and specialised classes might exist in the system; we can abstract the commonality to one type of class and specialize through extending that class definition.

### c. Inheritance

This is the mechanism for defining extensions i.e. generality and specialism. A class definition can be extended by creating a subclass definition which inherits attributes and methods from its parent class and adds the new attributes and methods required for the specialist object. This leads to a class hierarchy and allows for the creation of libraries of class definitions which can be extended for new scenarios.

### d. Polymorphism

Polymorphism recognizes that a method name should describe the action on the object which its code performs. Often information is passed into the method to allow it to carry out its task but sometimes the code required to implement the action is dependent on the number and type of data values supplied i.e. there is often the need to identify a basic action but have variations dependent on the data supplied – we really wish to use the same name for the action and we can do this provided the method signatures are different i.e. type or number of data arguments supplied or the return type of the method.

Additionally this allows a subclass to override a method which it has inherited from its parent class with a more specialised form. For example, a common method defined in a class is a **toString()** method which returns a string representation of the current state of the object i.e. its attribute values. A subclass definition will inherit this method but since the subclass adds additional attributes will require a new version which also provides the values of the new attributes.

### e. Modularity

While, traditionally, modules are thought of as divisions of functionality i.e. code elements, which combine to provide the overall functionality of a system/application; classes can be thought of as basic modules which interconnect to provide the necessary functionality. We are going to use an Application class which will create (code) objects to provide the functionality and model the (data) objects which hold the information to be processed. The aim of modularity is to define modules which are as independent as possible and which have minimal connections to other modules.

Benefits include individual module testing, reuse of modules between systems and applications, and pluggabilty i.e. the ability to be able to change a module implementation for a new one without affecting the implementation of other modules in the system.

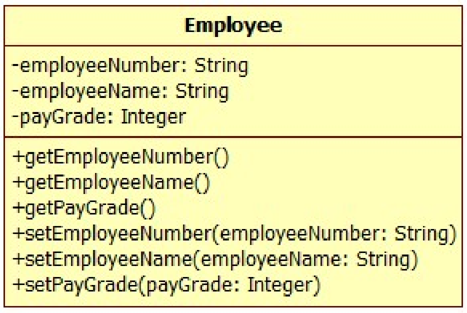
## 4. UML & Class Diagrams

The UML is the commonly used modelling language for modelling objects (classes) and their associations in business systems. There are a number of different forms of diagram used in the UML which focus on different views of the business system and its working.

Object (class) modelling is represented using a class diagram which specifies the name of the class; its attributes (and their type); and, methods and their signatures i.e. values which need to be supplied and any calculated output.

### a. Example

An Employee class might be represented in the class diagram below:



## 5. Discussion Questions

1. A customer brief should identify the functional (and non-functional) requirements of a required app/system. What role do the following UML diagrams play in the development of a software solution?
    a. Use case diagram
    b. Class diagram
2. Identify the main components of a class diagram which models data objects explaining how they are implemented in Java.