Unit 1d: Imperative Approach (Object Oriented)

1 Introduction

While object oriented concepts had first been discussed in the 60s and programming languages such as LISP and Smalltalk developed, widespread discussion and adoption did not happen to the 90s.

This paradigm is based on the concept of objects which may contain data, in the form of fields, and behaviour, in the form of methods. Complex problem domains can be *modelled* as the *collaboration* between a collection of simpler objects each with its own *responsibilities* and the capability to *communicate* with each other using *messages*, or method calls.

Key concepts include encapsulation, composition, inheritance, and, polymorphism.

In essence this approach involves the encapsulation of data and code that access/manipulates that data into a indivisible unit called an object.

2 Objects

OO ideas become popular as a way of representing real life objects encapsulating state and behaviour. This is now the dominant way of modelling business systems. While pure OO languages exist, most languages additionally incorporate procedural ideas, and, commonly now also supply functional programming support e.g. Java and C#.

3 Class Oriented

Most OO languages (such as C#, Python) are actually *class-oriented*: classes define what types of objects can be created and objects are instances of those classes. The type system typically includes classes as types. Behaviour/code reuse is implemented through inheritance of classes.

```
class Car{
   string manufacturer;
   string model;


Car myCar = new Car("Ford", "Focus");
```

Objects are instances of classes; a class is a user-defined data type that represents the shape of an object i.e. its constituent parts, while objects are instances (variables) of the class type and are allocated memory locations to store the data associated with the object.

3.1 Object State

In class-oriented languages, a class definition will specify the form of the object state in terms of fields (properties/attributes), each of which has its own data type. A field can be a simple value such as a numeric value, involve a composition or aggregation relationship with other objects e.g. an **Address** object or a **List** of associated objects.

Typically the state will be hidden from any external code using an access modifier such as **private**; access will only be allowed through a **public** interface – Information Hiding.

3.2 Behaviour

Object behaviour is specified through public methods which can be used to define an interface for external code – although some **private** methods may be defined for internal use. Methods, like fields, may be associated with the class itself though typically they are object components.

Categories of methods include:

- constructors
- getters
- setters
- string representation
- collection methods
- comparison methods

Components of a method include:

- method name
- parameters – data required
- return type – void if method does not return a value
- access modifier – scope of access (**public**, **private**, **protected**)
- method body – statements to be executed

A method is really a procedure that is internal to an object/class.

3.3 Message passing

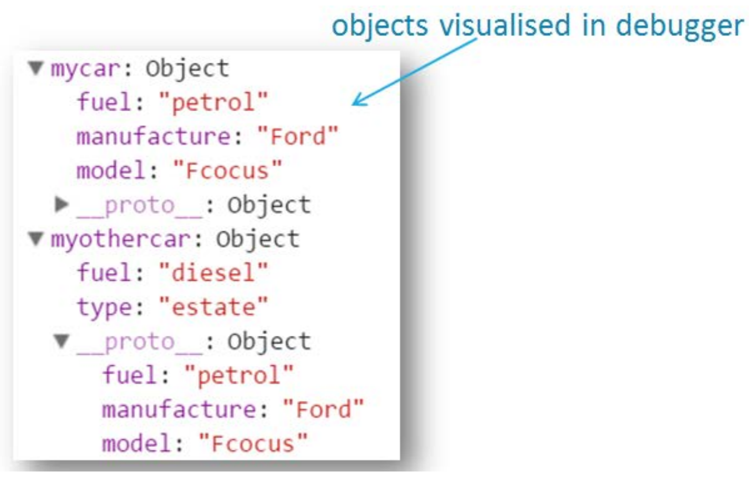
OO programs execute through objects passing messages to other objects i.e. an object requests that another object executes one of its **public** methods, perhaps passing data to it and receiving a result back.

4 Prototype Oriented

Some OO languages (notably JavaScript) are not class-oriented, but are *prototype-oriented* i.e. no classes, objects are created arbitrarily and you can add new fields (or behaviour) at run time. Behaviour reuse is achieved by cloning existing objects that serve as *prototypes*.

```
var mycar = {
"manufacturer": "Ford",
"model": "Focus"
}
```

```
mycar.fuel = "petrol"
var myothercar = Object.create(mycar)
myothercar.fuel = "diesel"
myothercar.type = "estate"
```



```
▼mycar: Object
   fuel: "petrol"
   manufacture: "Ford"
   model: "Fcocus"
  ▶__proto__: Object
▼myothercar: Object
   fuel: "diesel"
   type: "estate"
  ▼__proto__: Object
     fuel: "petrol"
     manufacture: "Ford"
     model: "Fcocus"
```

5 Inheritance

Inheritance relationships allow specialization (and generalization) i.e. we can create a subclass which inherits features from a superclass and adds specialized features – fields and methods. Methods can be overridden in a subclass to provide a version of the method suitable for the specialized objects – typically, dealing with the extra subclass fields.

Most OO languages support single inheritance i.e. a subclass can only have a single direct parent class; however, multiple inheritance can be a desirable feature of modelling real life objects and later in the module we will look at how this can be achieved and its consequences.

5.1 Abstraction

Inheritance naturally leads to a class hierarchy where the class(es) towards the top of the tree identify commonality and the classes towards the bottom provide detailed implementation. It can often be useful to specify commonality for a number of subclasses without specifying any implementation of behaviour or default behaviour. An abstract class specifies the basic property and behaviour of its subclasses but does not allow instantiation i.e. no objects can be created from the abstract class; it exists for organizational purposes. Similarly, we can specify the shape of a method without a body; subclasses provide detailed implementation of the abstract method. Often we specify an Interface of abstract methods which allows us to identify common behaviour without implementation; a class implements an interface. Interfaces can provide a documented contract allowing us to see what a class can do without being concerned about how it does it.