Unit D: Object orientated programming part 1

Object Orientated programming is a method of designing software that makes it easier for programmers working in teams to produce large, complex applications, by providing modularity not available in traditional procedural programming.

While you will be shown some programming code samples, you will learn more about the implementation of writing object orientated code in part 2.

1. Introducing OOP

Object orientated programming (OOP) is built on the idea of an "object". Instead of modelling your software around the individual instructions you want it to perform, your software is modelled around the objects that will interact with each other within your program. Object orientated programming draws inspiration from the real-world idea of objects... things... tangible "stuff" such as a person, a car, a book, or a house.

The idea is that different objects of the same category can be differentiated and described using the same properties and actions. For example, a "person object" could be described using their name, age, height, or education. For an "car object", you might be more interested in its make, model, colour, year of manufacture and service history. The properties (pieces of information - think your classic programming "variables") that are of interest about a person differ from the properties that you are interested in for a car. Object orientated programming provides a structured approach by which you define a class of objects that can be used for describing people, and another class of objects for describing cars. This method of classification of objects is, not coincidentally, known as a class.

Importantly, this approach to programming is not limited to only classifying physical objects. It can be equally used for virtual or intangible objects such as accessing a file on the hard drive, a node on the network, or to draw a shape on screen.

As well as properties, each class of object also has different behaviours or actions they have in common. Whereas a person might engage in conversation, work, study or exercise; a car would have actions such as acceleration, turning and breaking.

Object orientated programming allows you to create an abstraction for the properties and behaviours of different classes of objects together into one unified block of programming code. It provides a modular way of organising and separating your projects.

You can think of object orientated programming as allowing you to create your own data types and the functionality that goes along with them. Instead of being constrained to just writing programs using integers, floats and strings, you can now create a Person variable or a Car variable, and define the functions that control the calculations that can be performed with it.

An example using Strings

You will have been using objects and classes already in your programs without realising it. For instance, the String data type is object based. Let's consider the following two strings.

```
# Python
```

```
person1 = "Alan Turing"
person2 = "Grace Hopper"
print("Great computer scientists",person1,person2)
```

```
// Java
class Example {
    public static void main(String[] args) {
        String person1 = "Alan Turing";
        String person2 = "Grace Hopper";
        System.out.println("Great computer scientists "+person1+" "+person2);
    }
}
```

To operate on these variables, you would generally use the variable name, followed by a dot and a function name. That function would return a result specific to the particular string you used it on.

```
# Python
person1 = "Alan Turing"
person2 = "Grace Hopper"
person1_upper = person1.upper()
person2_upper = person2.upper()
print("Great computer scientists",person1,person2)
```

```
// Java
class Example {
    public static void main(String[] args) {
        String person1 = "Alan Turing";
        String person2 = "Grace Hopper";
        String person1Upper = person1.toUpperCase();
        String person2Upper = person2.toUpperCase();
        System.out.println("Great computer scientists "+person1Upper+"
"+person2Upper);
    }
}
```

The above examples will print output of "ALAN TURNING" and "GRACE HOPPER" respectively. Notice that you didn't have to provide a parameter to the .upper() containing data you wanted it to manipulate. The function already knew the data it was working on as it is "attached" to the person1 and person2 String variables. It already has access to the internal data stored within the variable.

This is object orientated programming at its core. It allows you to create data types and write functions that are attached to and have access to the internal information stored within them, thereby controlling how your program interacts with that data.

2. Defining classes

In the same way you need a blueprint to build a house, so you can't create objects without its class. The class is the template, or the blueprint, from which an object is derived. The class is the place where you will define the internal data stored by individual objects, and create the functions that will manipulate and maintain that data.

If a car is a class of object, the old red Ford Pickup owned by your neighbour, or the bright yellow Ferrari you see on the highway, are individual instances of a car. Individual cars on the road are objects, whereas the "class" is the overall category.

The process of defining a class usually has two key steps:

1. Identify the information you want to store about individual objects.
2. Identify the interactions you want other code to have on this information.

To do this we use the idea of abstraction which is the first of four core principles of object orientated programming.

3. Abstraction

Abstraction is the act of creating the programmatic model to represent our objects. It is the process of simplifying your scenario so you are only considering the detail that is relevant and ignoring the rest.

A commonly used analogy to explain the idea of creating abstractions is a map. Consider the following three maps of New York City.

1

2

3

All three maps ostensibly show the same thing: New York City; yet each is very different. The first map is useful if you want to see the major roads of Manhattan, the second is useful for riding the subway, and the third is useful for knowing where the various boroughs are located within the city. There are other maps that would specialise in topographical features, parks, waterways, buildings, or tourist attractions.

No map can possibly contain everything and still serve the task of being a simple and clear reference. Additionally while all these maps may be "of" New York City, none of them "are" New York City, they are a model of the city. They have information for a particular purpose and strip away unnecessary details. This is an abstraction.

When using the principle of abstraction to design classes for our programs, it is important to identify the information important to your problem so you can accurately design a model. It doesn't matter if you are designing a class to represent a person, a car, or something less tangible such as a sales transaction or a library loan, the process is the same.

The actual properties or attributes you define will vary depending on the context of the problem. To return to the example of a car, the properties of a "Car" class in use by the Government Department of Motor Vehicles would be different to the properties of a "Car" class in use by a used car showroom, or that of a car rental firm. The reason why there is no universally available "Person" class for you to use already is because it will vary considerably based on the scenario. A school looking to create a "Person" class for their student management software would have very different needs to the payroll software managing employees at a business, which would also be very different to the needs of a retail business wanting to keep customer data.

---

[1] https://www.openstreetmap.org/search?query=new%20york%20city#map=12/40.7336/-73.9194
[2] https://en.wikipedia.org/wiki/File:New_York_City_Subway_Map.svg
[3] https://commons.wikimedia.org/wiki/File:5_Boroughs_Labels_New_York_City_Map.svg

Exercise 1

You have been tasked with designing a Person class for a school student management system, a business payroll system and a retail store's customer system.

1. Identify 3 pieces of information all three systems may require that are the same across all the scenarios (for example, person name). For each item, decide the data type best suited to store it (integer, float, string, boolean, etc)
2. Identify 3 pieces of information for each system that are unique to each scenario (for example, the payroll scenario would need to store the salary of a person). For each item, decide the data type best suited to store it (integer, float, string, boolean, etc)

4. Encapsulation

Encapsulation is the idea that programming code outside of the class should not have direct access to the data within the class. Think of the term as referring to the data being protected by a capsule wrapped around it! The only way to permeate the capsule is through the methods the class allows.

When you are creating a small project it may not seem important if you read or write variables directly from outside the class. Once you scale, however, and are working on projects that involve many weeks of coding with multiple people working on it, it becomes critically important to adhere to encapsulation. There are all manner of possible complications that could cause bugs in your program.

An example is to consider what happens if our example Person class stores a "date of birth". As you begin programming you quickly realise that dates are complicated. Dates can be stored in all manner of ways using strings, integers or with specialised objects of their own. Even storing a date as a string is not clear as there are several commonly used styles in use such as dd/mm/yy (British), mm/dd/yy (USA), or yyyy-mm-dd (ISO). If your object is storing a date, there is likely code that depends on it being styled a particular way so that it can perform calculations on it, such as to determine a person's age. This code could fail if the date of birth property was accidently set to a value using the incorrect style. The principle of encapsulation exists for the purpose of preventing these unintended side effects.

Encapsulation allows the programmer to prevent properties from being updated by code outside the class, thereby protecting it from invalid, error inducing values. It also allows the object the opportunity to keep information secure from other parts of the program. If other objects can't obtain information without going through the object functions, then it means the object itself has the opportunity to decide whether or not to allow that information to flow out.

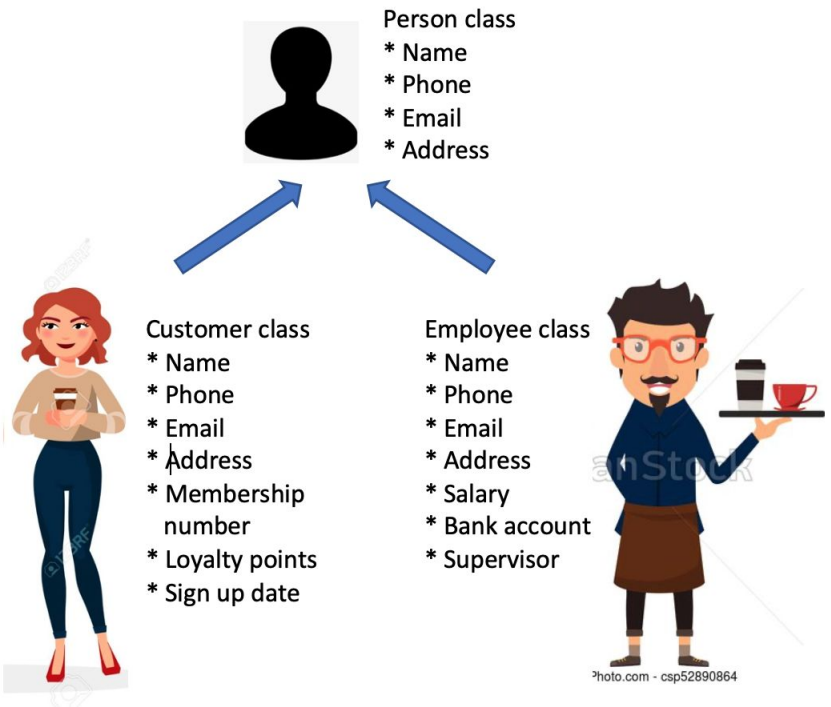
Successful encapsulation of your classes is at the heart of what allows modularity in object orientated programming. It means that as a programmer, you don't need to know or care how the Person class is storing its name. This complexity is hidden from you. You simply call the .get_name() function and trust it to reliably provide the name information to you. Internally the class may store it as one string, or it could split given-name and family-name into separate strings and then combine them together when you make your function request, or it could use some other method entirely different. In this way, encapsulation also provides flexibility for the author of the Person class to change how the internals of the class work. Provided the functions behave the same, no external code should break as a result.

5. Inheritance

Drawing from biology, the principle of inheritance is intended to ease code reuse when constructing a class that can be thought of as being derived from another class. When a class inherits from another, it takes in all the properties and methods of its parent.

- Programming note: A Python class can inherit from multiple parent classes, whereas a Java class can only inherit from one parent class.
- Terminology note: Parent classes may also be referred to as super classes; child classes that inherit from a parent may also be referred to as a subclass.

For example, think of how the Person class would work for a coffee shop. The business might have a loyalty scheme requiring the storing of customer details in the system. It would likely also need to store employee information in order to pay them. While customers and employees would have very different functions within the program, there would be some commonalities as well such as name and contact details. Inheritance allows the programmer to put these commonalities into a parent class called Person, while putting the specialist functionality into the Customer and Employee classes.



As the above diagram illustrates, in this scenario the Person class would contain the programming code responsible for any persons name, email address and phone number. The Customer class will inherit these basic properties and functions from Person and then extend upon them by adding a membership number, the points the customer has accrued, and their sign up date. The Employee class will likewise inherit the basic properties and functions of Person and extend those by adding salary and bank account details along with information about an individual employee's supervisor.

Exercise 2

Creating diagrams styled on the above one for Person / Customer / Employee, (a) identify which is the parent class and which are the child classes, and (b) properties that would likely exist in the parent class, properties that would likely exist in the children classes.

1. Human, Super Hero
2. Car, Motorcycle, Vehicle, Truck, Bus
3. Square, Rectangle, Box
4. Student, Teacher, Person

6. Polymorphism

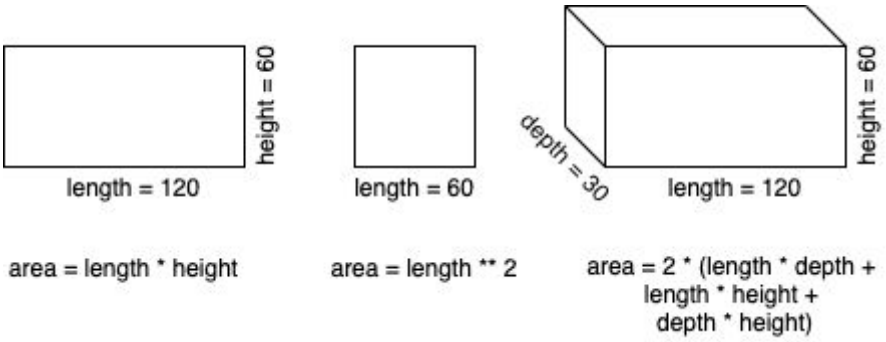
Another principle of object orientated programming is that of polymorphism. Polymorphism is another term that computer science has taken from biology and refers to something that can take many forms (poly...many; morph...change form).

There are two ways of getting polymorphic behaviour when programming with objects: Overriding and overloading.

6.1 Overriding

Overriding occurs when a child class creates a property or method of the same name as the parent class, thereby overriding it. You have actually already seen polymorphism by overriding but not had it pointed out to you that that is what was happening.

Consider the earlier example of geometric shapes that used inheritance. You hopefully identified that Rectangle was the super class, and Square and Box were child classes. Consider the function .get_area() for all three classes. This function would be calculated differently in all three cases....



Overriding means the same get_area() method will be used in all three classes. You can see in the following programming code that executing these three methods looks the same even though they are each performing a different calculation. (note: this code will not execute by itself without the corresponding code for the Rectangle, Square and Box classes. You will create that code in part 2.

```python
# Python
rect = Rectangle(120, 60)
sq = Square(60)
box = Box(120, 60, 30)
print( rect.get_area(), sq.get_area(), box.get_area() )
```

```java
// Java
class Example {
    public static void main(String[] args) {
        Rectangle rect = new Rectangle(120, 60);
        Square sq = new Square(60);
        Box box = new Box(120, 60, 30);
        System.out.println( rect.getArea() );
        System.out.println( sq.getArea() );
```

```
        System.out.println( box.getArea() );
    }
}
```

Given that the Square class inherits the .get_area() function from Rectangle, if you couldn't use overriding, you would be forced to create a new function name such as .get_square_area(). This situation would quickly become unworkable as you are forced to think, "which area function do I need for this circumstance?" The point of inheritance is to help keep things simple. Overriding means that our Square and Box classes can create their own .get_area() functions that will be used instead of the one they inherit from Rectangle.

6.2 Overloading

Overloading occurs when the one method can take multiple forms. This is generally when the combination and significance of the parameters required can vary. Overloading allows you to make the same function name available to achieve the same purpose, while accepting different inputs.

To demonstrate the usefulness of overloading, consider the Customer class and the fact it requires a membership date. To provide flexibility to whatever external code may need to use the Customer class, the programmer could elect to provide several different valid ways of setting this property such as by providing a string formatted with an ISO date, or by providing day, month and year as integers. The method signatures could be described as follows:

- `set_membership_date( iso_string : string )`
- `set_membership_date( day : int, month : int, year : int )`

Without overloading, the author of the Customer class would have to provide different method names for each scenario, such as `.set_membership_date_by_iso()` and `.set_membership_date_by_ints()`.

Polymorphism through both overriding and overloading allows programmers to avoid unnecessary complexity by using the same name for properties and methods, where their purpose is the same.

7. Class relationships & hierarchy

When you begin using object orientated programming, some of your objects will be dependent on other objects and so become linked. There are several different class relationships within Computer Science. The ones you will use in this course are inheritance, dependency, association and aggregation.

7.1 UML class diagrams

A UML (Unified Modelling Language) class diagram is a useful tool used to visually depict the definitions of classes and the relational links between them.

To describe an individual class with UML, it is drawn as a three row table. The first cell contains the class name, the second contains the properties / instance variables, and the third contains the methods / functions for your class. Refer to the general structure and example for a Rectangle class below.

General structure

Class name
- property: type - property: type - property: type
+ method(type): type + method(type): type + method(type): type

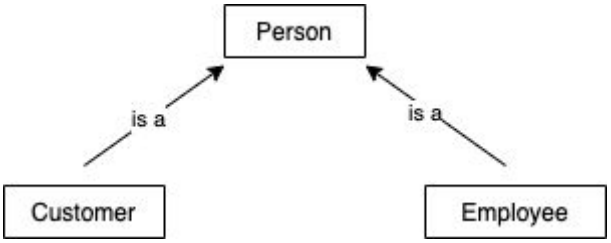
Example

Rectangle
- length: float - height: float
+ get_area(): float + get_perimeter(): float

The data type for all properties, method parameters and method return types should be specified. Ignore the plus and minus symbols for now. They refer to access modifiers which you will learn about in part 2.

7.1 Inheritance

You have already seen inheritance. The dependent nature of the relationship between two classes where one inherits from the other should be immediately obvious. A hierarchy of these class relationships can be drawn as an inheritance tree in a UML Class diagram with a solid arrow indicating inherited dependence. The keyword "is a" is written over the arrow. When UML diagrams are being used to describe class relationships it is common to only draw the cell containing the class name without the properties and methods as the following demonstrates.

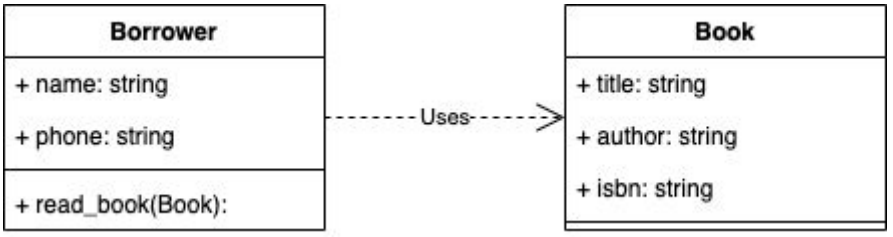


Using our previous retail store customer and employee system example, the above simplified UML Class diagram illustrates that as Customer class "is a" Person, it is dependent on the functionality within Person in order to work correctly. Person, however, is not dependent on Customer for it to function correctly by itself.

7.2 Dependency

Dependency is an umbrella term used to denote one class being dependent on another. Inheritance, association and aggregation are specialised forms of dependency.

An example of dependency at work would be to consider a library loans system. A person borrowing from a library, uses the book and then returns it. In this way, a temporary connection exists between Borrower and Book. Once Borrower is done with the Book, the system can break the connection.



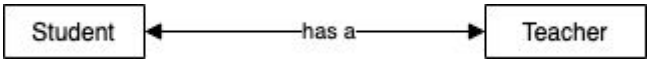
The UML Class diagram for generalised dependency would be denoted with a dashed arrow with the keyword "uses" overlaid.

Programmatically, this will typically show itself through a method that requires the dependency as a parameter but that is not subsequently stored as an instance variable.

7.3 Association

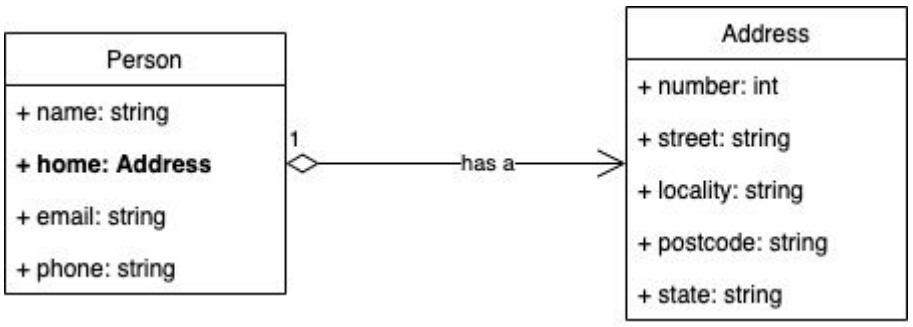
Students and Teachers have an associative relationship. The objects are separate entities from each other (and can exist without each other), but associate and interact with each other in a relationship that may be one-to-one, one-to-many, or many-to-many. For instance, one student may have many teachers, and one teacher may have many students.

The UML Class diagram would denote this relationship with a double-headed arrow and the keyword "has a" as shown.



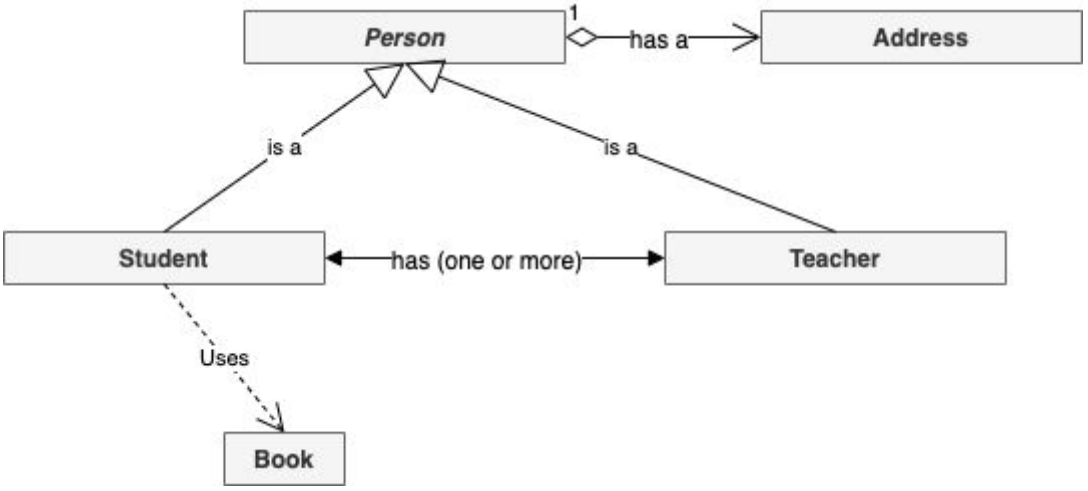
7.4 Aggregation

Aggregation is a special case of dependency. It describes when a connection between objects is mandatory for at least one of them. An example is that in the case of a Person class, consider that a Person requires an Address but an Address does not require a Person. This would be depicted in a UML Class diagram by using an arrow with a diamond on the end connecting the dependent class as shown. The keyword "has a" is written over the arrow.



Programmatically, this will typically show itself through one class having an object of the other as an instance variable.

All four relationship types that have been discussed can be brought together into one complete UML diagram as follows



Past paper questions for review

(refer to separate document)