Lesson 17: Article 6: Object Oriented Programming

Object oriented programming is another programming paradigm which is an extension to the ordinary Pascal programming language. Object pascal helps programmers to approach a problem in a different way from the ordinary procedural programming paradigm. When programming using procedural approaches to a problem, one can only use calls to procedures and functions in order to perform some operation. In object oriented programming, one can define a particular object.

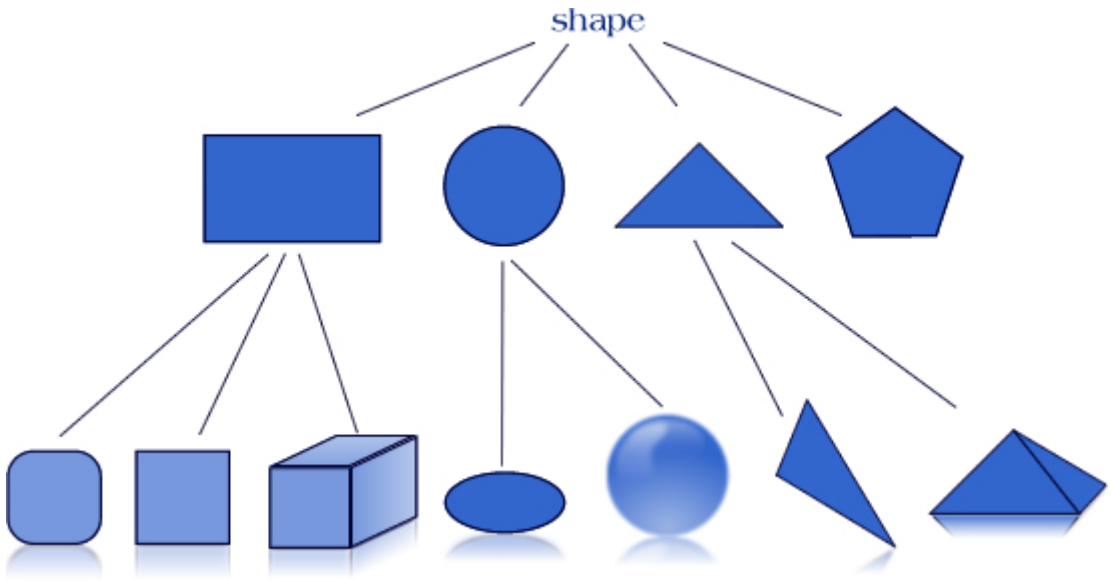
When we watch around us, we all see objects of different types: your computer that you are using right now, televisions, mobiles, cup, persons, cars, animals etc... Objects may be categorized into two parts: moving and non-moving. Moving objects can move on their own whereas non-moving objects can't move or they require a force in order to be moved. All objects have attributes which uniquely identify them from other objects. Such attributes may be the size, weight, colour, dimensions etc... Apart from attributes, objects may also have their own behavious which seperate their unique identity from other objects as well; (such as a person walks, sits on a chair, eats, runs etc.., an aeroplane takes off, flies, auto-pilots, start engines, drive on the runway etc...). Although there are many objects out there that are distinct from each other, some may exhibit similar behaviour and share similar characteristics. This can be seen in humans and monkeys.

Classification of Objects

When coming to allocate objects into groups using a tree hierarchy, one can start from the very top - i.e. the most general, to the very bottom of the tree - i.e. the most specific. Let's take for example a vehicle object. When someone talks with you about vehicles, it may come to your mind a car, or maybe a motorcycle. A truck, trailer and a bus are also vehicles. They are all objects having wheels that can be driven by one driver and transport some type of medium around the land - a bus carries people whereas a truck carries goods. These objects can all be defined using object orientation by declaring their properties (using variables) and behaviours (using methods - ie. procedures and functions).

Objects that more or less have similar characterstics can be categorized into a tree hierarchy that are most general at the top and are specified further down the tree. To simplify what is being explained, let's take a shape as an example. A shape can be any of rectangle, circle, triangle, cube, sphere and so on. But you cannot specify that a shape is a rectangle. So we have got an abstract idea of a shape - something that has got an enclosed body. A shape always has got points of origin, corners or curves. For example, a rectangle has got 4 vertices, and a circle has got a point of origin around which the line of circle is drawn.

Not all shapes have got the same characteristics. As you know, the more we go down the tree hierarchy the more we specify.



The class hierarchy above shows how the very general shape object is classified at every stage by other different shapes. The more we go down from the top, the more we specify the object. If we take the path of the rectangle, it is divided into three specific objects - rounded rectangle, square and a cuboid.

In this context, it is important to introduce the concept of **Inheritance**. In the diagram above, inheritance is shown as one goes down the classification tree - hence an object becomes more specific. The rectangle shape inherits methods (e.g. *draw()*)and properties (e.g. a record structure variable that stores x and y i.e. *Point(x,y)*) from the shape object. The rounded rectangle, square and cuboid inherit methods and properties of the rectangle. It is implicit that these three specific shapes automatically inherit methods and properties of the object that was inherited by the rectangle i.e. the shape object. The same applies to the other group of shapes of the same level.

It is possible that the shape object may include methods (i.e. functions and procedures) that need not necessarily be implemented but only declared. This concept is known as **Polyphormism**. For a clear and simple example of polyphormism is the following: we assume that all shapes need to be drawn. The shape object need only to declare a method called *draw()*. But this is not implemented since we don't know which type of shape we should draw. However, whichever object inherits the shape object, needs to explicitly implement the *draw()* function. For example, the rectangle object inherits the shape object, therefore it must implement the *draw()* function which draws a rectangle on the screen - otherwise a compiler error results.

Hopefully, I have introduced you to some important concepts of object orientism. You will meet them and use them inevitably when you'll be doing object oriented programming.