Exercise 1 - Defining Object-Oriented Programming Concepts

Encapsulation

Encapsulation is the concept of defining the scope of class/ object members, and how these classes/ objects are permitted to interact with one another. This in turn allows greater control over how these elements are accessed.

Being able to directly change a primitive variable in one object could lead to errors upon an invalid value being assigned; on the other hand, if the variable is restricted and an assessor function is used instead a specific type is defined in the parameter of said function. Upon trying to fit a boolean value into an int variable an error will occur immediately, as opposed to undefined behaviour happening later.

For example, if the program takes in user input to see what their favourite number is, some sanitising of the input is needed. If the input was directly fed into favouriteNumber there is the chance a string of characters being entered into there. Through hiding favouriteNumber from the class responsible for setting it, and using setFavouriteNumber() instead, the program may check to see if a number was truly entered and respond accordingly (probably through throwing an exception).

On the other hand, these hidden member variables need to be accessed as well, otherwise storing it is pointless. Due to them now being hidden away an accessor function to return the value is now needed.

These two concepts of assessor functions are colloquially known as 'getters' and 'setters' to return and assign the private members, respectively.

Inheritance

Inheritance allows for new classes to gain the functionality of already-existing classes. This in turn leads to a parent-class or base class, and a child-class or subclass.

The purpose of this is that a subclass may need to perform a lot of the same tasks as the base class, but with a few differences (via overriding base class functions) or additions (via defining new functions as normal).

For example there might be a Clock class. Clocks allow you to set the time, and display the time. In the event of another class needing a time variable and its getters/ setters, it would make sense to perform these tasks in the same manner to the Clock class to save you defining everything twice. Let's say this other class is an IlluminatedClock; it tells the time, lets you set the time, but it also lights up. Two of those functions are the same as a Clock so it makes sense to have IlluminatedClock inherit from Clock. Now all that needs defining in IlluminatedClock is the lightUp() function.

Polymorphism

Polymorphism is where the functionality of multiple classes vary, while the interface for these classes remains the same. Due to the functionality of classes being defined in the classes themselves through encapsulation, it's no longer necessary to create long function names to disambiguate one function from another.

Continuing the Clock and IlluminatedClock example, let's say there's now a DigitalClock. This DigitalClock allows you to set the time, but the way is displays the time is different (assuming Clock is analogue). Through overriding the displayTime() function of Clock in DigitalClock to display the time digitally it is possible to later invoke displayTime() without the compiler querying which displayTime() you want to execute. This saves on defining two different functions to needlessly disambiguate (displayAnalagueTime() and displayDigitalTime() are needlessly lengthy function names).

Abstraction

Abstraction builds on inheritance and polymorphism. In many cases, when a class inherits from another class, it is considered to be-a of the base class. Sometimes there are many classes inheriting from the same base class, which indicates there are some/ many similarities between them all. Abstraction is the term for this base class.

In the Clock, IlluminatedClock, and DigitalClock examples, abstraction is to some extent nearly taking place. For abstraction to occur, the abstract class needs to not be instantiable, thus making Clock more of a blueprint than anything. Due to the apparent range of clock types there are available using Clock as a base class would provide sufficient functionality to set and read the time.

Having clock types inherit from Clock to define how the time is set and read leads to the abstraction, an IlluminatedClock is-a Clock, a DigitalClock is-a Clock, and to retain the old analogue style clock there would now be an AnalogueClock inheriting from Clock.

While some multiple inheritance would occur if this design was taken further to have a light-up digital clock, the concept should be clear by now. The base class defines what all the subclasses are implementations of.