A class is a description of an object. It defines what it knows, the fields of the class, and what it can do, its methods. This all comes together to define the interface of the object which the rest of the program will interact with to use that object. It usually also defines how a new instance is created via a constructor and some languages like C++ and C# also use them to define how they are destroyed via a destructor or finaliser.

Encapsulation is the main goal of Object-Oriented programming. It aims to encapsulate related data and functionality into classes of objects. This goes hand-in-hand with information hiding and interfaces. Data is typical hidden from the rest of the program, often by using some sort of 'private' keyword and the methods are the interface / the actions that can be used to access or alter that data. This encapsulates the two into a single object which can be passed around the program and used in a way that maintains flexibility and low coupling if done correctly. With the correct interface to an object, the data and behaviour of a class can be changed without needing to alter the rest of the program to facilitate these changes

An object is something that knows things and can do things. In Object-Oriented programming, objects are just about the single most important idea hence where it gets the name. In an object-oriented programming computation is achieved by sending messages to and from many different objects which is usually achieved by calling an object's method and using the result somewhere else in the program perhaps by sending that result as an argument to another object's method. An object is typically an instance of a class. The things that the object knows are the classes fields and the things it can do are the methods of that class. It's good OO design when an object can only be altered through its methods as methods should be the interface that describes all the legal actions that can be performed by or performed on an object. The 'Counter' class that we made in the 'Clock' program shows this principle. The Counter can only be incremented or reset and that is expressed with the 'Increment' and 'Reset' methods respectively.

Abstraction is the process of mapping real world concepts to something else that is simpler and/or easier to work with. The best abstractions are those that completely change the programmer's mental model of the problem. Abstraction is at the heart of Object-Oriented programming and Structured programming as they themselves are abstractions over procedural programming and machine code respectively. Before Structured programming became the norm, code was not organised into blocks with control flow statements used to control which blocks were executed and instead all the code was just a list of instructions and you could use 'goto' like instructions to move the instruction pointer. This pretty much always resulted in spaghetti code and so structured programming was proposed as the solution where these blocks were used as the fundamental unit of code that could then be jumped into and out-of using loops and if statements. Object-Oriented programming takes it further by turning the fundamental unit of code into a class that knows things and can do things. Instead of conceptualising a program as data being passed around different procedures it is now modelling a domain with classes that are themselves simple but collaborate with other classes to perform more complicated functionality by messaging specific instances of those classes. In the shapes programming that we made. We abstracted away the idea of pixels on a screen and instead worked with entire shapes using the 'Shape' class which made it easier to make visuals on the screen. Instead of having to manually set each pixel to a certain colour to make a Square we instead instantiate a 'Square' object and draw it to the screen.

Inheritance is used to model that certain classes are different kinds of a specific object or have the same basic knowledge and functionality. This is commonly referred as an 'is-a' kind of relationship. For instance if you had an 'Animal' base class from which a 'Cat' class inherited. A 'Cat' is a type of 'Animal'. Inheritance is very useful for when you have lots of different types of objects that share things in common and you want to be able to write code that handles the different types of data all together. For instance you could have an array of Animals with each element of that array being a different type of animal. A cat, a dog or a horse. Using inheritance allows the programmer to express this relationship and gain the flexibility that inheritance gives you. We used inheritance to model the idea of a 'LibraryResource' that could be anything that a library could contain. We made derived classes for games and for books and by modelling the program with inheritance we could easily aggregate all types of 'LibraryResources' in the 'Library' class. In my custom project, inheritance was used to model the different types of Abstract Syntax Tree nodes that the Virtual Machine could execute.

Being able to use each different type of AST node, or each type of 'LibraryResource' with the same code is an example of polymorphism. This is the power of inheritance and polymorphism. Being able to generically use an object so long as it implements some base functionality and being able to override base functionality with new ones allows for incredibly extensible and maintainable programs that don't have to be rewritten over and over again to add new functionality based on a different type of an object.

Collaboration is where the design of the program is really tested. The more complicated functionality of an object-oriented is done by getting different types of objects to collaborate to perform a bigger task. And if an object's interface isn't quite correct it can be very difficult to get different objects to collaborate in a way that is easy or maintains good design principles and instead end up with a great big pile of spaghetti that just barely works but in a way that no one can understand. In my custom project the Tokenizer and the Parser collaborate to generate an Abstract Syntax Tree that the Virtual Machine can then execute. To maintain a clean way for these three to collaborate to execute a program I used Tokens as an intermediate representation that makes the Parser's job much easier. These tokens are passed as a List to the Parser and provides a very clear way that the two communicate. The parser than returns an AST which can then be passed to the VM and executed. The Tokenizer and the Parser are fairly coupled which is to be expected as technically speaking the Tokenisation process is apart of parsing however the Parser and the VM although they collaborate, they aren't very highly coupled. So long as the Parser can generate valid Abstract Syntax Trees the Virtual Machine will still be able to execute the programs that the Parser generates.