Object Oriented Programming

Object-oriented programming is a programming paradigm that uses abstraction (in the form of classes and objects) to create models based on the real world environment. An object-oriented application uses a collection of objects, which communicate by passing messages to request services. Objects are capable of passing messages, receiving messages, and processing data. The aim of object-oriented programming is to try to increase the flexibility and maintainability of programs. Because programs created using an OO language are modular, they can be easier to develop, and simpler to understand after development.

Object-Oriented Programming vs. Procedural Programming

Programs are made up of modules, which are parts of a program that can be coded and tested separately, and then assembled to form a complete program. In procedural languages (i.e. C) these modules are procedures, where a procedure is a sequence of statements. In C for example, procedures are a sequence of imperative statements, such as assignments, tests, loops and invocations of sub procedures. These procedures are functions, which map arguments to return statements.

The design method used in procedural programming is called Top Down Design. This is where you start with a problem (procedure) and then systematically break the problem down into sub problems (sub procedures). This is called functional decomposition, which continues until a sub problem is straightforward enough to be solved by the corresponding sub procedure. The difficulties with this type of programming, is that software maintenance can be difficult and time consuming. When changes are made to the main procedure (top), those changes can cascade to the sub procedures of main, and the sub-sub procedures and so on, where the change may impact all procedures in the pyramid.

One alternative to procedural programming is object oriented programming. Object oriented programming is meant to address the difficulties with procedural programming. In object oriented programming, the main modules in a program are classes, rather than procedures. The object-oriented approach lets you create classes and objects that model real world objects.

Classes and Objects

A class is a collection of objects that have common properties, operations and behaviours. A class is a combination of state (data) and behaviour (methods). In object-oriented languages, a class is a data type, and objects are instances of that data type. In other words, classes are prototypes from which objects are created.

For example, we may design a class Human, which is a collection of all humans in the world. Humans have state, such as height, weight, and hair color. They also have behaviour, such as walking, talking, eating. All of the state and behaviour of a human is encapsulated (contained) within the class human.

An object is an instance of a class. Objects are units of abstraction. An object can communicate with other objects using messages. An object passes a message to another object, which results in the invocation of a method. Objects then perform the actions that are required to get a response from the system.
Real world objects all share two characteristics; they all have state and behaviour. One-way to begin thinking in an object oriented way is to identify the state and behaviour of real world objects. The complexity of objects can differ, some object have more states and more complex behaviours than other object. Compare the state and behaviour of a television to the states and behaviours of a car.

Software objects are conceptually similar to real world objects. An object stores its state in fields, and it exposes its behaviours through its methods.

A fundamental principle of object oriented programming is encapsulation; the hiding of an objects internal state and requiring all interaction to be performed using the objects methods. (Think of the classes used in VB.Net)

**Classes and Objects Example - class Bicycle**

There are many types of bicycles, but we can say that each bicycle was built from the same blueprint (or prototype) and each bicycle contains the same components. Your bicycle (a specific bicycle object) is an instance of a class of objects known as bicycles. Describe the state and behaviour of a bicycle.

**Fields and Methods**

Objects in a class have a number of shared properties/features/attributes. Fields are the variables contained in a class (encapsulated) that are used to represent these properties. For example, class Student may have an integer field to represent graduation year, or a String field to represent the student’s Last Name.

Before fields in an object can be assigned values, the object must first be constructed/created. In Java (as an example of an OO language) a special function called a **constructor** is used to create and initialize an instance of a class. The statement:

```
new Student();
```

creates an instance of the class Student, or in other words, a Student object. Each class has operations associated with it. Each Student object has a number of distinct behaviours, such as study, sleep, work, etcetera. The operations associated with a class, together with the attributes of a class are encapsulated within the class. These operations are called the methods of a class. Methods are functions that represent the operations associated with a particular class. Fields and methods are referred to as class **members**.

**Encapsulation**

**Definition:** the ability of an object to hide its data and methods from the rest of the world - one of the fundamental principles of OOP. Because objects encapsulate data and implementation, the user of an object can view the object as a black box that provides services. Instance variables and methods can be added, deleted, or changed, but as long as the services provided by the object remain the same, code that uses the object can continue to use it without being rewritten.
**Coupling** – the degree to which a module (class) depends on other classes. In a good design, you should try to minimize coupling. Classes should be self-contained units that have a low dependence on other classes, meaning understanding and using one class should not require an understanding of another. With low coupling a change in a module will not require changes in other modules.

Related to the concept of coupling is information hiding. Information hiding is the hiding of the implementation in a class or module that are most likely to change; this protects other parts of the program from change if the implementation is changed. Protecting an implementation involves providing a stable interface which other classes can use to access services provided by the class. Often encapsulation and information hiding are used interchangeably.

Example: Car and Driver

**Cohesion** – is the measure within a module (class) of how well the members work together to provide a specific piece of functionality. Cohesion is measured by how strongly related and focused the responsibilities of a single class are. In good design, the cohesion should be maximized. Cohesion is decreased if the methods of a class have little in common, or methods carry out many activities.

If cohesion is low, modules may be difficult to understand, maintenance is difficult, because change may affect many modules, and modules cannot be reused, because it is unlikely that another application will have a use for the random grouping of functionality.

Example: Oven that has built-in radio and alarm clock.

*A well-designed system should maximize cohesion, and minimize coupling.*

**Inheritance**

Multiple classes may share some of the same characteristics, and have things in common with one another, but also may have certain properties that make them different. Object oriented programming languages allow classes to inherit commonly used state and behaviour from other classes.

Classes in Java occur in *inheritance hierarchies*. These hierarchies consist of parent child relationships among the classes. Inheritance is used to specialize a parent class, but creating a child class (example). Inheritance also allows designers to combine features of classes in a common parent.

Example: Student

**Benefits of Object Oriented Programming**

1. **Modularity**: The source code for a class can be written and maintained independently of the source code for other classes. Once created, an object can be easily passed around inside the system.
2. **Information-hiding**: By interacting only with an object's methods, the details of its internal implementation remain hidden from the outside world.

3. **Code re-use**: If a class already exists, you can use objects from that class in your program. This allows programmers to implement/test/debug complex, task-specific objects, which you can then use in your own code.

4. **Easy Debugging**: If a particular object turns out to be a problem, you can simply remove it from your application and plug in a different object as its replacement. This is analogous to fixing mechanical problems in the real world. If a bolt breaks, you replace it, not the entire machine.

**Summary of Object-Oriented Concepts**

1. Everything is an object.

2. Computation is performed by objects communicating with each other, requesting that other objects perform actions. Objects communicate by sending and receiving messages. A message is a request for action, bundled with whatever arguments may be necessary to complete the tasks.

3. Each object has its own memory, which consists of other objects.

4. Every object is an instance of a class. A class simply represents a grouping of similar objects, such as Integers or lists.

5. The class is the repository for behaviour associated with an object. That is, that all objects that are instances of the same class can perform the same actions.

6. Classes are organized into a singly rooted tree structure, called the inheritance hierarchy. Memory and behaviour associated with instances of a class are automatically available to any class associated with a descendant in this tree structure.

**Programming paradigms**

A **programming paradigm** is a model of programming based on distinct concepts that shapes the way programmers design, organize and write programs. A multi-paradigm programming language allows programmers to choose a specific single approach or mix parts of different programming paradigms. C++ as a multi-paradigm programming language supports single or mixed approaches using Procedural or Object-oriented programming and mixing in utilizations of Generic and even Functional programming concepts.

**Procedural programming**

**Procedural programming** can be defined as a subtype of imperative programming as a programming paradigm based upon the concept of procedure calls, in which **statements** are structured into procedures (also known as subroutines or functions). Procedure calls are modular
and are bound by scope. A procedural program is composed of one or more modules. Each module is composed of one or more subprograms. Modules may consist of procedures, functions, subroutines or methods, depending on the programming language. Procedural programs may possibly have multiple levels or scopes, with subprograms defined inside other subprograms. Each scope can contain names which cannot be seen in outer scopes.

Procedural programming offers many benefits over simple sequential programming since procedural code:

- is easier to read and more maintainable
- is more flexible
- facilitates the practice of good program design
- allows modules to be reused in the form of code libraries.

Note:
Nowadays it is very rare to see C++ strictly using the Procedural Programming paradigm, mostly it is used only on small demonstration or test programs.

Object-oriented programming

Object-oriented programming can be seen as an extension of procedural programming in which programs are made up of collection of individual units called objects that have a distinct purpose and function with limited or no dependencies on implementation. For example, a car is like an object; it gets you from point A to point B with no need to know what type of engine the car uses or how the engine works. Object-oriented languages usually provide a means of documenting what an object can and cannot do, like instructions for driving a car.

Objects and Classes

An object is composed of members and methods. The members (also called data members, characteristics, attributes, or properties) describe the object. The methods generally describe the actions associated with a particular object. Think of an object as a noun, its members as adjectives describing that noun, and its methods as the verbs that can be performed by or on that noun.

For example, a sports car is an object. Some of its members might be its height, weight, acceleration, and speed. An object's members just hold data about that object. Some of the methods of the sports car could be "drive", "park", "race", etc. The methods really do not mean much unless associated with the sports car, and the same goes for the members.

The blueprint that lets us build our sports car object is called a class. A class does not tell us how fast our sports car goes, or what color it is, but it does tell us that our sports car will have a member representing speed and color, and that they will be say, a number and a word, respectively. The class also lays out the methods for us, telling the car how to park and
drive, but these methods can not take any action with just the blueprint - they need an object to have an effect.

**Encapsulation**

«No component in a complex system should depend on the internal details of any other component.»

Dan Ingalls (Smalltalk Architect)

Encapsulation, the principle of information hiding (from the user), is the process of hiding the data structures of the class and allowing changes in the data through a public interface where the incoming values are checked for validity, and so not only it permits the hiding of data in an object but also of behavior. This prevents clients of an interface from depending on those parts of the implementation that are likely to change in future, thereby allowing those changes to be made more easily, that is, without changes to clients. In modern programming languages, the principle of information hiding manifests itself in a number of ways, including encapsulation and polymorphism.

**Inheritance**

Inheritance describes a relationship between two (or more) types, or classes, of objects in which one is said to be a "subtype" or "child" of the other, as result the "child" object is said to inherit features of the parent, allowing for shared functionality, this lets programmers re-use or reduce code and simplifies the development and maintenance of software.

Inheritance is also commonly held to include subtyping, whereby one type of object is defined to be a more specialized version of another type (see Liskov substitution principle), though non sub-typing inheritance is also possible.

Inheritance is typically expressed by describing classes of objects arranged in an inheritance hierarchy (also referred to as inheritance chain), a the tree like structure created by their inheritance relationships.

For example, one might create a variable class "Mammal" with features such as eating, reproducing, etc.; then define a subtype "Cat" that inherits those features without having to explicitly program them, while adding new features like "chasing mice". This allows commonalities among different kinds of objects to be expressed once and reused multiple times.

In C++ we can then have classes that are related to other classes (a class can be defined by means of an older, pre-existing, class). This leads to a situation in which a new class has all the functionality of the older class, and additionally introduces its own specific functionality. Instead of composition, where a given class contains another class, we mean here derivation, where a given class is another class.

Multiple inheritance
Multiple inheritance is the process by which one class can inherit the properties of two or more classes (variously known as its base classes, or parent classes, or ancestor classes, or super-classes).

In some similar language, multiple inheritance is restricted in various ways to keep the language simple, such as by allowing inheritance from only one real class and a number of "interfaces", or by completely disallowing multiple inheritance. C++ places the full power of multiple inheritance in the hands of programmers, but it is needed only rarely, and (as with most techniques) can complicate code if used inappropriately. Because of C++’s approach to multiple inheritance, C++ has no need of separate language facilities for "interfaces"; C++'s classes can do everything that interfaces do in some related languages.

**Polymorphism**

Polymorphism allows a single name to be reused for several related but different purposes. The purpose of polymorphism is to allow one name to be used for a general class. Depending on the type of data, a specific instance of the general case is executed.

The concept of polymorphism is wider. Polymorphism exists every time we use two functions that have the same name, but differ in the implementation. They may also differ in their interface, e.g., by taking different arguments. In that case the choice of which function to make is via overload resolution, and is performed at compile time, so we refer to static polymorphism.

*Dynamic polymorphism* will be covered deeply.