OOP Concepts

Basically, the Object-Oriented Programming concepts, also referred to as pillars of OOP are:

* Encapsulation
* Inheritance
* Polymorphism
* Abstraction

Let’s see what each entail:

Encapsulation

Encapsulation is all about exposing a solution to a problem without requiring the consumer to fully understand the problem domain. Encapsulation is binding the data and behaviours together in a single unit. This prevents the client or the consumer of the module from knowing about the inside view where the behaviour is implemented. Encapsulation in Java is achieved by prepending the private modifier to the target property or method. When this is done, the data can only be accessed within the class implementation and cannot be accessed directly outside of the class. However, we may sometimes need to do something with that data. How then do we do that?? Well, this can be done through the use exposed functions (accessor and mutator methods). Hiding the internals of the object protects its integrity by preventing users from setting the internal data of the component into an invalid or inconsistent state. By convention, the default modifier for all properties should be private(which I saw a couple of you do). When a method is marked as private, it usually means it is going to be used in the definition of another method and is referred to as a helper method.

Inheritance

***Inheritance*** is a powerful feature of Object-oriented programming languages. Inheritance helps in organizing classes into a hierarchy and enabling these classes to inherit attributes and behaviour from classes above in the hierarchy. Inheritance describes an “IS A” relationship. A very general form of a class can be defined and compiled. Later, more specialized versions of that class may be defined by starting with the already defined class and adding more specialized instance variables and methods. The general class is referred to as a ***Base class*** or ***Superclass***, whereas the specialized class is referred the ***Derived class*** or ***subclass***. The specialized classes are said to ***inherit*** the methods and instance variables of the previously defined general class. A derived class automatically has all the fields and all the methods that the base class has, and can have additional methods and/or additional fields. Inheritance does not work backward. The parent won’t have properties of the derived class. The “extends” word depicts inheritance in Java.

Polymorphism

Polymorphism is the concept that there can be many different implementations of an executable unit and the difference happen all behind the scene without the user’s awareness. Inheritance allows you to define a base class and to define software for the base class. That software can then be used not only for objects of the base class but also for objects of any class derived from the base class. Polymorphism allows you to make changes in the method definition for the derived classes and to have those changes apply to the software written in the base class ***(method overriding***).

Types of polymorphism

In Java there are two types of polymorphism:

1. ***Compile-time polymorphism***: This type of polymorphism is when a method definition is associated with its invocation at compile time. This is achieved through method overloading (recall last year’s c++ with me 🙃) and operator overloading. Although operator overloading is not supported in Java. This type of polymorphism is also called ***static polymorphism*** or ***early binding***.
2. ***Run-time polymorphism***: This type of polymorphism is when a method definition is associated with its invocation at run time. This is achieved through method overriding. Run-time polymorphism is also known as ***dynamic binding***.

***Note***: To prevent polymorphism (***method overriding***), we prepend the “***final***” keyword to the method to denote that the method can’t be overridden. Likewise, an entire class could be marked final to depict that class cannot be inherited from or a property marked final to indicate that it can’t be altered.

Abstraction

***Abstraction*** lets you focus on what the object does instead of how it is done. The idea behind abstraction is knowing a thing on a high level. Abstraction helps in building independent modules which can interact with each other by some means. Abstraction aims to represent the essential feature without detailing the background implementation or internal working detail. We try to selectively focus on only those things that matter to us or in the case of programming, to our module. Modifying one independent module does not impact the other modules. The only knowledge one needs to know is what a module gives you. The person who uses that module does not need to bother about how the task is achieved or what exactly is happening in the background. A popular example is applying brakes to a vehicle. The braking system is abstracted and you are provided with a paddle for stopping your vehicle. Making changes to acceleration system does not affect the braking system they are independent. You also do not have to bother about the internal working of the brakes, you only have to press the brake pedal and be it disc brake or drum brake, the vehicle stops. Encapsulation aids in abstraction. They can be thought of as two sides of the same coin. The two ways abstraction can be achieved in Java is by using ***abstract classes*** and ***interfaces***.

***An abstract class*** is a class that has some methods without complete definitions. You cannot create an object using an abstract class constructor, but you can use an abstract class as a base class to define a derived class. An ***abstract method*** has a heading just like an ordinary method, but no method body. The syntax rules of Java require the modifier “***abstract***” and require a ***semicolon*** in place of the missing method body (no curlies). Consequently, abstract methods get defined in derived classes.

***Interfaces*** in Java are abstract types used to define the general behavior of a class. An interface contains static constants and abstract methods. Interfaces cannot be used to instantiate objects, but they can be used to implement classes, which in turn can be used to create objects. A class that implements an interface must define all the methods in the interface. The “***interface***” keyword is used to declare an interface. With interfaces, 100% abstraction can be achieved in Java. The “***implements***” keyword is used to used to implement the interface.

**Abstract class vs Concrete class**

An abstract class has at least one method being abstract (without a body), while a concrete class has no abstract class at all. In other words, the methods in a concrete class are all defined.