Object Oriented Design & Programming

Object-Oriented Model

* In OOP, computation is represented as the interaction among or communication between objects.
* An object is an entity that contains both, the attributes that describe the state of a real-world object and the actions that are associated with the real-world object.
  + The attributes of an object encompasses the data/variables that characterize the state.
  + The state of an object encompasses all of the (usually static) properties of the object plus the current (usually dynamic) values of each of these properties.
  + The behaviour of an object encompasses the methods that represent the services & operations an object provides.
* Messages are requests from an object to another for the receiving object to produce some result.
* A class is a template/blueprint for objects. It contains data properties & methods. An object is a specific instance of a class.
* Abstraction – An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus, provides crisply defined conceptual boundaries.
* Encapsulation – Encapsulation builds a barrier to protect an object’s private data. Access to private data can only be done through public methods of the object’s class, such as accessors & mutators.
  + Information Hiding – Hides the details of implementation of the class from users.
* Inheritance – A mechanism that defines a new class that inherits the properties and behaviours (methods) of a parent class. Superclass/Base Class (Parent) 🡪 Subclass/Derived Class (Child). Any inherited behaviour may be redefined and overridden in the subclass. Avoids duplication of code.
  + Multiple inheritance is when a class inherits from more than one superclass. A problem arises when there is more than one property/method to inherit with the same name.
* Polymorphism – Same message can be sent to different objects with different results. Sending object does not need to know the class of the receiving object or how the object will respond.
* ***this*** – References the receiver object.
* ***static*** – Declares a class variable or class method. Applies to the whole class instead of individual objects.
* Object Composition – An object can include other objects as its data member(s). The class will contain object references as its instance variables. “has-a” relationship.

Inheritance

* An important OO feature that allows us to derive new classes from existing classes by absorbing their attributes and behaviours and adding new capabilities in the new classes. This enables code reuse and can greatly reduce programming effort. “is-a” relationship.
* The superclass is a generalization of the subclasses.
* The subclasses are specializations of the superclass.
* ***super*** – super() calls the superclass’ constructor and super.X() can be used to call a superclass’ method.
* Method Overloading – When a method is overloaded, it is designed to perform differently when supplied with different signatures i.e. same method name but different number of parameters or parameter types. *Not a behavior due to inheritance.*
* Method Overriding – A subclass inherits properties and methods from the superclass. When a subclass alters a method from a superclass by defining a method with exactly the same signature, it overrides that method. Can either be a refinement or a replacement of the superclass’ method.
* When a message is sent to an object, the search for a matching method begins at the class of the object → immediate superclass → and so on…
* Visibility Modifiers:
  + public (+) – Visible anywhere in an application.
  + private (-) – Visible only within that class.
  + protected (#) – Visible anywhere within the same package.
* A package contains a set of classes that are grouped together in the same directory. Non-private data can be accessed by any object in the same package.
* Packages allow the same class name to be used in two different packages. For eg: X.Deck & Y.Deck.
* ***final method*** – When a method is declared as final, it cannot be overridden in subclasses.
* ***final class*** – When a class is declared as final, it cannot be a superclass.
* This helps improve security by ensuring no change in behaviour & efficiency by reducing compile-time type checking and binding.
* Abstract Classes & Methods ({abstract}) – Abstract methods don’t have any implementation in the abstract class. The implementation must be provided by the subclass(es).
  + public abstract class Rectangle {}
  + public abstract double findArea();
* Multiple inheritance is not supported by Java. However, Java does support implementing multiple interfaces.
* An interface is like an abstract class except it contains only abstract methods and constants (with *static final*). The *abstract* keyword is not needed for an interface in Java.
* A class implementing an interface has to provide an implementation for all the abstract methods. Otherwise, the new class will also be abstract.
* Interfaces can inherit each other as per normal i.e. *extends*.
* An abstract class is a real base class but an interface is not.
* An abstract class can have object attributes (data members) and non-abstract methods but an interface cannot.
* A class *extends* an abstract class, however, it *implements* one or more interfaces.
* Both types cannot be instantiated as objects with *new*.
* A concrete class is a class with implementation for all methods.

Polymorphism

* In OOP, polymorphism means the ability of an object reference to refer to different types; knowing which method to apply depends on where it is in the inheritance hierarchy.
* When a program invokes a method through a superclass variable, the appropriate subclass version of the method is called, based on the type of the object reference stored in the superclass variable.
* The same method name & signature can cause different actions to occur, depending on the type of object on which the method is invoked.
* Binding – Refers which method to be called at a given time [i.e. connecting a method call to a method body].
* Static Binding – Occurs when the method call is bound at compile time.
* Dynamic Binding – The selection of the method body to be executed is delayed until execution time (based on the actual object being referred).
  + Java uses this by default for all methods except private, final & static.
* Upcasting – When an object of a derived class is assigned to a variable of a base class (or any ancestor class). However, subclass-only members cannot be referred to by a superclass variable.
* Downcasting – When an object of a base class is assigned to a variable of a derived class. This doesn’t make sense in many cases and may be illegal.
* *object instanceof ClassName* – will return true if *object* is an instance of *ClassName* or any descendent class of *ClassName*.
* Benefits of Polymorphism:
  + Simplicity – Code can ignore type-specific details and just interact with the base type of the family. Makes it easier to write and understand the code.
  + Extensibility – New functionality can be added by creating new derived classes without modifying other derived classes.
* Method overriding can be done by overriding methods of a superclass, implementing abstract methods of an abstract class or implementing methods of an interface.

Design Principles

* Symptoms of Rotting Design:
  + Rigidity – The tendency of software to be difficult to change, even in simple ways. Every change causes a cascade of subsequent changes.
  + Fragility – The tendency of software to break in many places every time it is changed. Breakage may occur in areas that have no conceptual relationship with the area that was changed.
  + Immobility – The inability to reuse software/module from other projects or from parts of the same project. The module may have too much baggage that it depends on.
* Good design & programming must be easy to read, easy to maintain and modify, efficient, reliable and secure.
* The main design goal of OOD is to make software easier to change i.e. minimise impact of change.
* Class Design Guidelines:
  + Design with Reusability, Extensibility & Maintainability in mind.
  + Achieve loose (low) coupling and high cohesion.
* A modular program has well-defined, conceptually simple and independent units interacting through well-defined interfaces. Achieved through encapsulation, low coupling & high cohesion.
* SOLID Design Principles:
  + Single Responsibility Principle – There should never be more than one reason for a class to change. If the class has more than one responsibility, then the responsibilities become coupled.
  + Open-Closed Principle – A module should be open for extension but closed for modification. We want to be able to change what the modules do, without changing the source code of the modules.
  + Liskov Substitution Principle – Subtypes must be substitutable for their base types. A user of a base class should continue to function if a derivative of that base class is passed to it.
    - Design by Contract: Methods should specify their pre- and post-conditions.
  + Interface Segregation Principle – Many client specific interfaces are better than one general purpose interface. Classes should not depend on interfaces that they do not use.
  + Don’t Repeat Yourself – Refactor to eliminate duplicated code and functionality.
  + Dependency Injection Principle – High level modules should not depend upon low level modules. Both should depend upon abstractions. This allows the simple reuse of high level modules.