OOPS:
modularize
manageable and predictable
Better maintainability
more reuse of code

Features of OOPS:
Emphasis on Data rather than procedures
objects
functions that operate on data are tied together
objects may communicate with each other through functions
new data and functions can be added whenever necessary

Concepts:
Classes and objects.
Methods
Encapsulation
Association, aggregation and composition
Inheritance
Polymorphism
Abstraction
Modularity
Coupling
Cohesion
Interfaces

Encapsulation -

- Encapsulation is defined as the wrapping up of data under a single unit. It is the mechanism that binds together code and the data it manipulates.Other way to think about encapsulation is, it is a protective shield that prevents the data from being accessed by the code outside this shield.
- Technically in encapsulation, the variables or data of a class is hidden from any other class and can be accessed only through any member function of own class in which they are declared.
- As in encapsulation, the data in a class is hidden from other classes, so it is also known as data-hiding.
- Encapsulation can be achieved by: Declaring all the variables in the class as private and writing public methods in the class to set and get the values of variables.

Inheritance

- 2 or more entities that are different but share many common features.
- features common to all classes are defined in superclass.
- the classes that inherit common features from super classes is called subclass.

why inheritance?

- classes often share capabilities.
- reuse code:
  - to improve maintanability
  - reduce cost
  - improve real world modeling.
- Make classes more flexible.

Abstraction

- Abstraction provides the freedom to defer implementation decisions by avoiding commitment to details.
- Data Abstraction is the property by virtue of which only the essential details are displayed to the user.The trivial or the non-essentials units are not displayed to the user.

Association:

- Relationship between objects.
- one to many, ...

Aggregation:

- Specialized form of associated.
- 'has a' relationship
- unidirectional association. e.g. department can have students, but not vice versa
- In aggregation, both the both the entities can survive individually.

Why Aggregation?

- Code reuse is best achieved through aggregation.

Composition:

- Restricted form of aggregation.
- 2 entities are highly dependent on each other.
- Represents "part-of" relationship
- when there is composition b/w 2 entities, the composed object cannot exist without other entity.

Aggregation vs composition:

- child can exist without parent in aggregation.
- child cannot exist without parent in composition.

<table>
<tr><td>

- Data Abstraction may also be defined as the process of identifying only the required characteristics of an object ignoring the irrelevant details.

</td><td>

- has-a vs part-of
- aggregation is weak whereas composition is strong association.

</td></tr>
<tr><td>

Polymorphism

- Many Forms
- overloading and overriding

Polymorphism refers to the ability of OOPs programming languages to differentiate between entities with the same name efficiently. This is done by Java with the help of the signature and declaration of these entities.

</td><td>

Overloading:

- method can have different definitions by defining different types of parameters.

Overriding:

- Subclass and parent class has the same methods, parameter types and return types.

</td></tr>
<tr><td>

Interfaces:
Concept of abstraction and encapsulation.
Defines the abstract methods, inputs and outputs, but not implementation.

</td><td rowspan="2">

Composition vs inheritance:

- Prefer composition when not all superclass functions are reused by subclass.
- Inheritance leads to tight coupling b/w subclass and superclass. Harder to maintain.
- Inheritance is easier to use than composition.
- Composition make the code maintainability in future, especially when your assumption breaks.
- Liskov substitution principle:

</td></tr>
<tr><td>

Modularity
Logical component of a large program can each be implemented separately.
Has benefits not just for organizing the implementation, but for fixing problems later.

</td></tr>
<tr><td>

Coupling:
Defines how dependent one object on another.
Coupling is a measure of strength of connection between any 2 system components.
The more any one component knows about the other components, the tighter the coupling.

</td><td>

Cohesion:
How narrowly defined an object is?
Cohesion is a measure of how logically related the parts of individual components are to each other. and to the overall components.

</td></tr>
<tr><td colspan="2">

Low coupling and high cohesion is good object oriented design.

</td></tr>
<tr><td>

SOLID: Most popular sets of design principles in OOP.

</td><td rowspan="3">

Open/Closed Principle:

- “Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.”
- “A class is closed, since it may be compiled, stored in a library, baselined, and used by client classes. But it is also open, since any new class may use it as parent, adding new features. When a descendant class is defined, there is no need to change the original or to disturb its clients.”

Later renamed polymorphic open/closed principle:

- It uses interfaces instead of superclasses to allow different implementations which you can easily substitute without changing the code that uses them. The interfaces are closed for modifications, and you can provide new implementations to extend the functionality of your software.
- The main benefit of this approach is that an interface introduces an additional level of

</td></tr>
<tr><td>

Single responsibility:

- A class should have one, and only one, reason to change.

Why?

- it makes your software easier to implement and prevents unexpected side-effects of future changes.
- You can avoid these problems by asking a simple question before you make any changes: What is the responsibility of your class/component/microservice?
- If your answer includes the word “and”, you’re most likely breaking the single responsibility principle.

</td></tr>
<tr><td>

Liskov substitution principle:

- Let Φ(x) be a property provable about objects x of type T. Then Φ(y) should be true for objects y of

</td></tr>
</table>

type S where S is a subtype of T.

- The principle defines that objects of a superclass shall be replaceable with objects of its subclasses without breaking the application. That requires the objects of your subclasses to behave in the same way as the objects of your superclass.
- That means you can implement less restrictive validation rules, but you are not allowed to enforce stricter ones in your subclass. Otherwise, any code that calls this method on an object of the superclass might cause an exception, if it gets called with an object of the subclass.

Dependency Inversion principle:

- High-level modules, which provide complex logic, should be easily reusable and unaffected by changes in low-level modules, which provide utility features. To achieve that, you need to introduce an abstraction that decouples the high-level and low-level modules from each other.
- High-level modules should not depend on low-level modules. Both should depend on abstractions.
- Abstractions should not depend on details. Details should depend on abstractions.

DDD (Domain-Driven Design):

- DDD is about making pragmatic decisions.
- Try not to 'force' a pattern into the model, and, if you do 'break' a pattern, be sure to understand the reasons and communicate that reasoning too.
- It is said that DDD is object orientation done right but DDD is a lot more than just object orientation.
- DDD also encourages the inclusion of other areas such as Test-Driven Development (TDD), usage of patterns, and continuous refactoring.

Representing a model:

- Domain-Driven Design is all about design and creating highly expressive models. DDD also aims to create models that are understandable by everyone involved in the software development, not just software developers.
- Since non-technical people also work with these models, it is convenient if the models can be represented in different ways. Typically, a model of a domain can be depicted as a UML sketch, as code, and in the language of the domain.

Ubiquitous language:

- Reveal the Intention not the Implementation.
- Aim for Deep Insights.

abstraction which enables loose coupling.

Interface Segregation principle:

- “Clients should not be forced to depend upon interfaces that they do not use.”
- the goal of the Interface Segregation Principle is to reduce the side effects and frequency of required changes by splitting the software into multiple, independent parts.
- Extends Single responsibility principle.
- By following this principle, you prevent bloated interfaces that define methods for multiple responsibilities.

Dependency Injection:

- Transferring the task of creating the object to someone else and directly using the dependency is called DI.

There are 3 types of DI:

- Constructor
- Setter
- interface

Inversion of control:

- This states that a class should not configure its dependencies statically but should be configured by some other class from outside.
- It is the fifth principle of **S.O.L.I.D**
- According to the principles, a class should concentrate on fulfilling its responsibilities and not on creating objects that it requires to fulfill those responsibilities. And that’s where **dependency injection** comes into play: it provides the class with the required objects.

Benefits of using DI:

- Helps in Unit testing.
- Boilerplate code is reduced, as initializing of dependencies is done by the injector component.
- Extending the application becomes easier.
- Helps to enable loose coupling, which is important in application programming.

Application Architecture:

- Typically, a layered architecture can be used to isolate the domain from other parts of the system.
- Each layer is aware of only those layers below it. As such, a layer at a lower level cannot make a call (i.e. send a message) to a layer above it. Also, each layer is very cohesive and classes that are located in a

- Refactor the Language.
- Work with Concrete Examples.

Strategic design:

- Strategic design is about design in the large, and helps focus on the many parts that make up the large model, and how these parts relate to each other.
- In DDD, these smaller models reside in bounded contexts. The manner in which these bounded contexts relate to each other is known as context mapping.

- Bounded Contexts:
  - Contexts can be created from (but not limited to) the following:
    - how teams are organized
    - the structure and layout of the code base
    - usage within a specific part of the domain

- Context Maps:
  - Context mapping is a design process where the contact points and translations between bounded contexts are explicitly mapped out. Focus on mapping the existing landscape, and deal with the actual transformations later.

Modeling the design:

- Dealing with Structure:
  - Entities
  - Cardinality of Associations
  - Services
  - Aggregates

- Dealing with Life Cycles:
  - Factories
  - Repositories

- Dealing with Behavior:
  - Behavior design patterns

particular layer pay strict attention to honoring the purpose and responsibility of the layer.

- User Interface -
  - Responsible for constructing the user interface and managing the interaction with the domain model. Typical implementation pattern is model-view-controller.
- Application -
  - Thin layer that allows the view to collaborate with the domain. Warning: it is an easy 'dumping ground' for displaced domain behavior and can be a magnet for 'transaction script' style code.
- Domain -
  - An extremely behavior-rich and expressive model of the domain. Note that repositories and factories are part of the domain. However, the object-relational mapper to which the repositories might delegate are part of the infrastructure, below this layer.
- Infrastructure -
  - Deals with technology specific decisions and focuses more on implementations and less on intentions. Note that domain instances can be created in this layer, but, typically, it is the repository that interacts with this layer, to obtain references to these objects.

Note: Information gathered in this document has been collected from various sources on Internet.

Sources:

https://dzone.com/refcardz/soa-patterns?chapter=1
https://dzone.com/refcardz/continuous-delivery-patterns?chapter=1
https://dzone.com/refcardz/designing-quality-software?chapter=1
https://dzone.com/refcardz/contexts-and-depencency?chapter=1
https://dzone.com/refcardz/patterns-modular-architecture?chapter=1
https://dzone.com/refcardz/software-configuration?chapter=1
https://dzone.com/refcardz/getting-started-domain-driven?chapter=1