

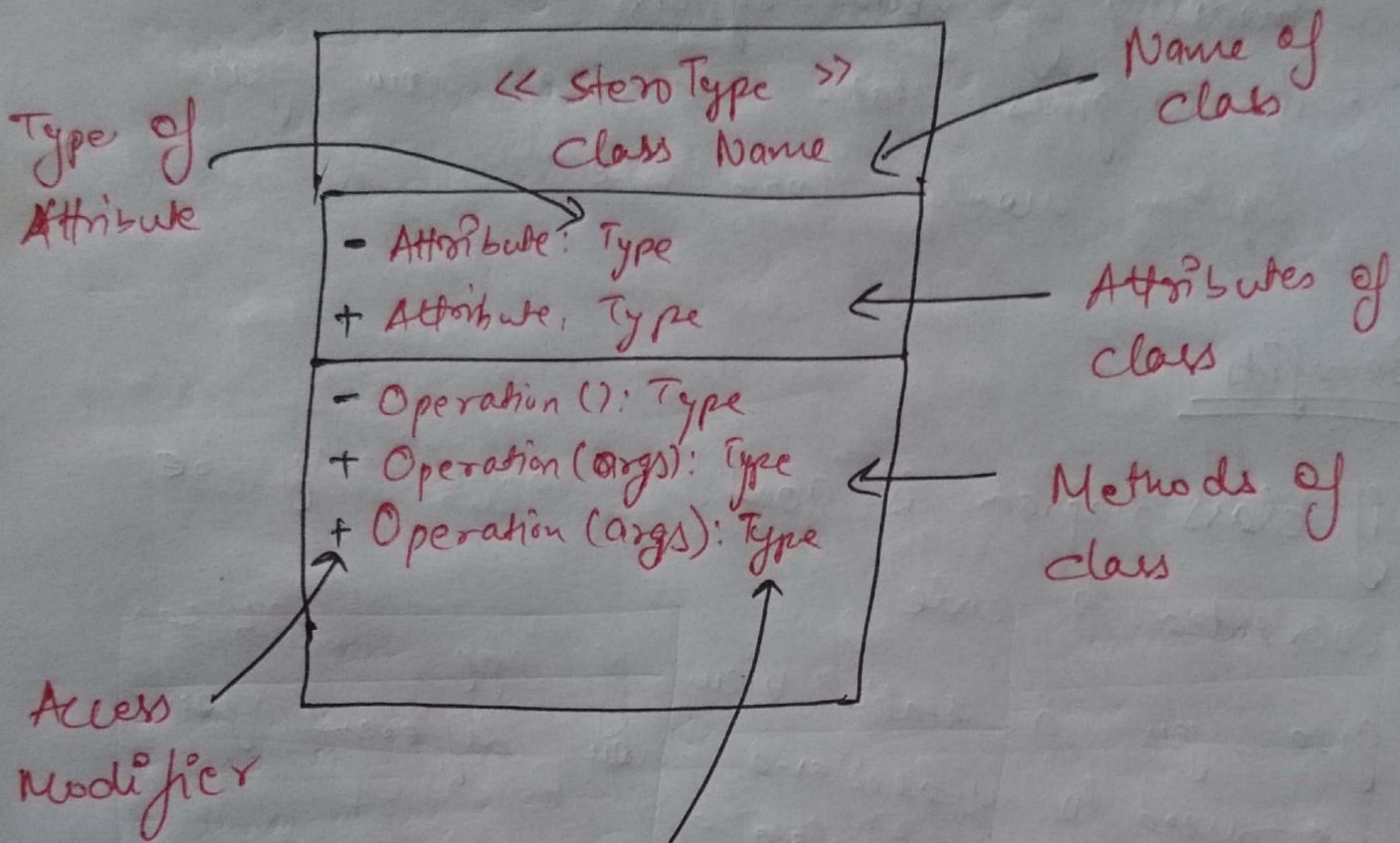
Class Diagram

- supporting system for object oriented modeling of your system
- It ~~also~~ gives static view of how your system would look like

→ Benefits

- If class diagram is good, it can directly be converted to code
- Give you a static view of app, which can be used to understand, how different your entities will co-relate
- It's ~~drawn~~ at base level, useful for both development & production & can also help in reverse engineering & forward engineering

→ class



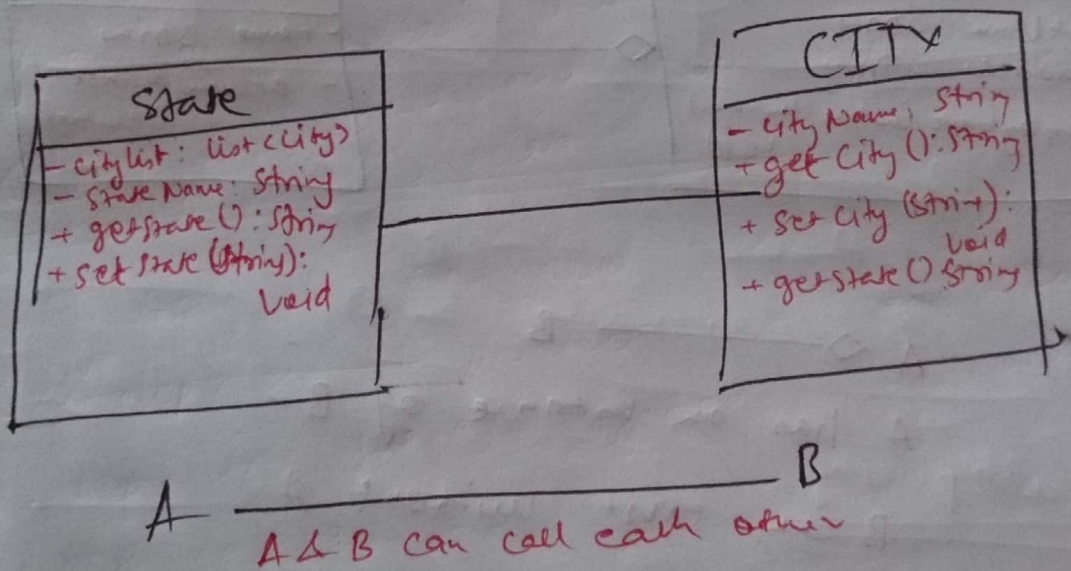
Return Type of Method

+	: public
-	: private
#	: protected
~	: Package

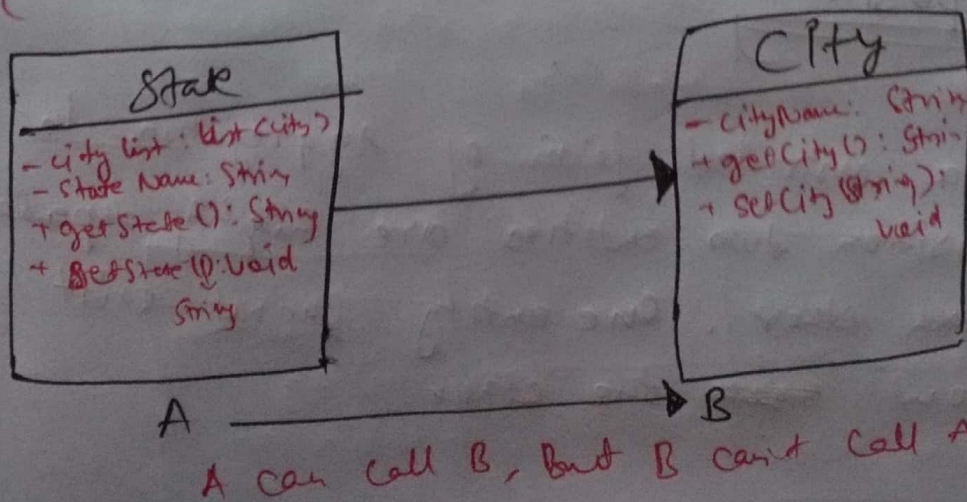
→ Association

- if two classes in a model needs to communicate with each other, there must be a link b/w them. This link can be represented by association.
- Associations can be represented in a class diagram by a line b/w these classes with an arrow indicating the navigation direction.

→ Bi directional



→ Unidirectional

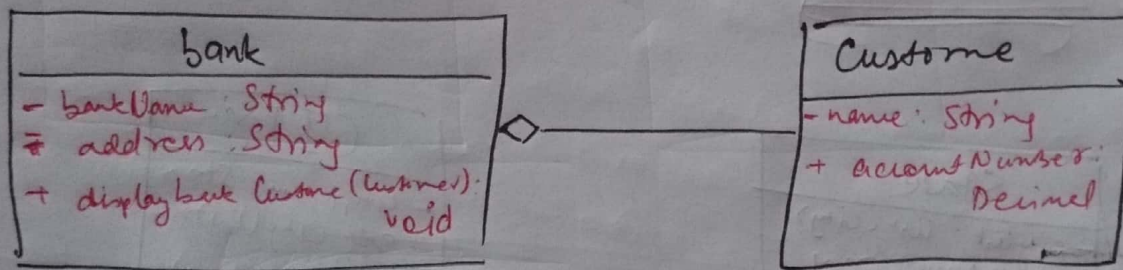


⇒ Aggregation

→ Aggregation represents the has-A relationship.

→ It is a one-way relationship and called unidirectional association. For example, Bank can have customers but vice versa is not possible & that's why its unidirectional in nature.

→ Both class's objects will not affect each other.



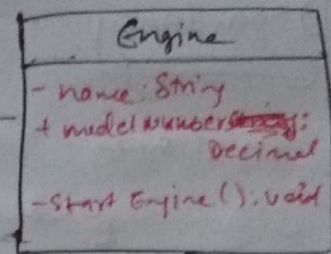
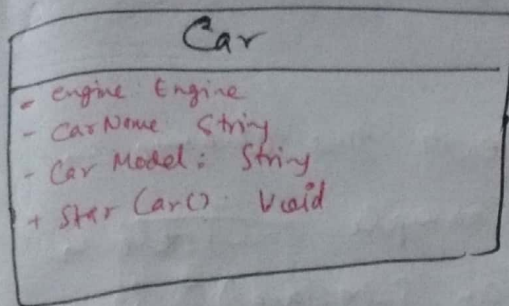
A ◇ — B

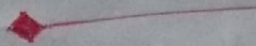
A has an instance of B

B can exist without A

⇒ Composition

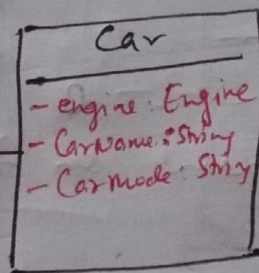
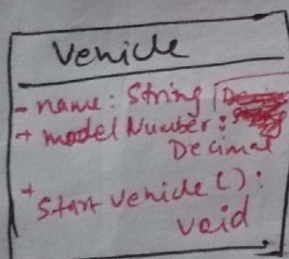
→ It is a restricted form of Aggregation. In composition two entities are highly dependent on each other. One entity cannot exist without ~~each~~ the other.




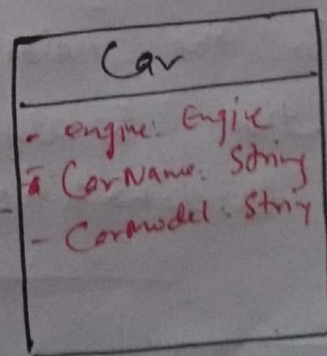
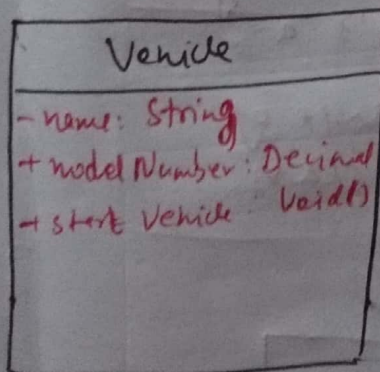
A  B
 A has an instance of B
 B can't exist without A


⇒ Generalization

→ It is a mechanism for combining similar classes of objects into a single, more general class.
 → It identifies commonalities among a set of entities.



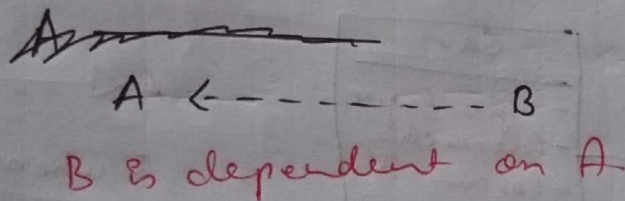
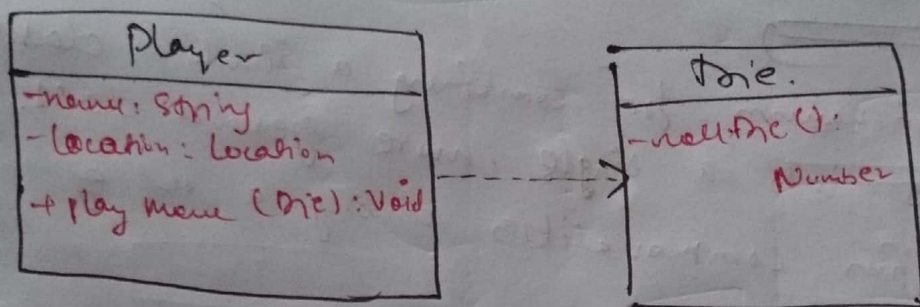
A  B
 B inherits from A
 B is a A



A  B
 B implements A

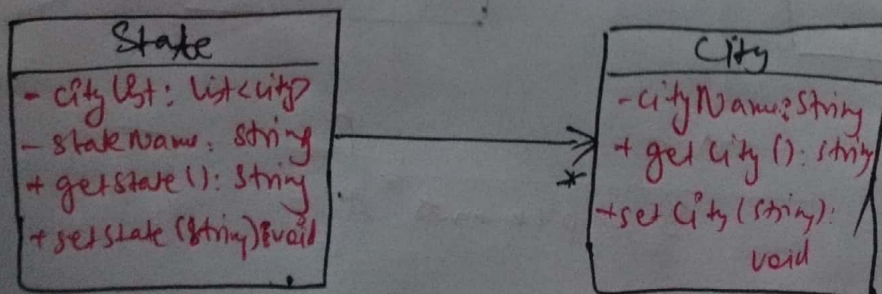
⇒ Dependency

⇒ A change in class affects the change in its dependent class. Example. Circle is dependent on shape (an interface). If you change shape, it affects Circle too. So circle has a dependency on shapes.

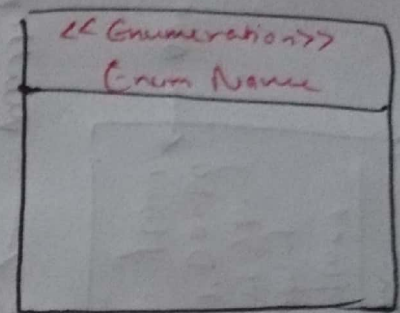
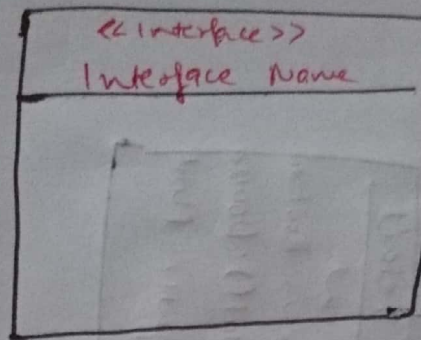
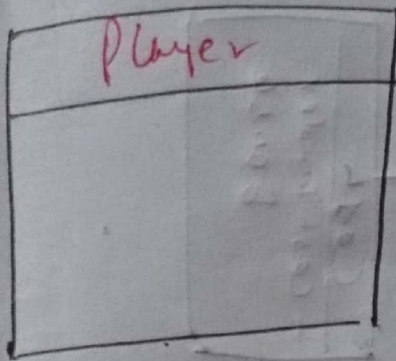


⇒ Multiplicity

Multiplicity indicates how many instances of class participate in the relationship. It's a constraint that specifies the range of permitted cardinalities two class



⇒ Abstract Class, Interface, Enum.



⇒ Class diagram for "Place an order".

Involved Entities/objects:

Customer

Order

Order Details

Item

Payments

types of payment.

