AGENDA

1.> UML DIAGRAMS

2.> CLASS DIAGRAMS

Start by 9:05 PM

*] UML DIAGRAMS:

•> communication
→ Picture > 1000 words.

•> communication with
Product Managers / Leaders

↳ Mostly; technical communication

*] Day 2 Day life communication :

↓↓ ↓↓
Managers / leadership / PM

Purposes -
1.> Manager: tasks / Planning
2.> PM: Product related stuff
3.> client: understanding req.
4.>

focus: show / discuss Implementation of system with peers.

*1 WAYS TO COMMUNICATE :

1.> Words -

Meet / email / slack.

Problems:

↘

wasn't clear / Improper english etc

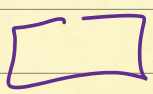
2.> Pictures / Graphs

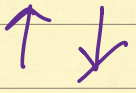
Picture > 1000 words

-> Easy to understand anything
-> less Ambiguity
-> Easy to visualize

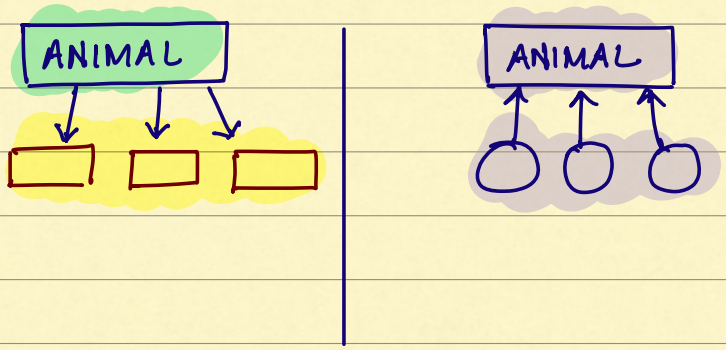
↘

PROBLEMS:





1.> NO standardization



Leads to confusion

∴ Hence, we should ensure some std. is done / followed.

*] UNIFIED MODELLING LANG:

Provides std. way on how to implement diff s/w concepts in diagrams

-> Has defined representation for classes / interfaces / modifiers etc

⟶ follow this & problem solved.

*] USES :

1.> day 2 day life / design docs / reviews
2.> Interviews

*> TYPES OF UML :

STRUCTURAL UML	BEHAVIOURAC UML
•> class diagram	•> Activity
•> Package	•> usecase
•> Object	•> sequence
•> Component	

*] USE CASE DIAGRAM:

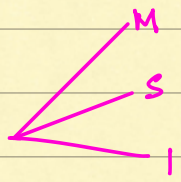
- talks about features / func supported by s/w system
- who can use the func

eg: ① create Mentor session

who can create : student

② Functionality.

login() - everyone can use.



*] HOW TO CREATE :

(5 keywords) *

1> System Boundary : [RECTANGLE]

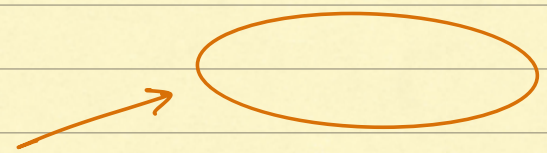
Every feature supported - Goes Inside Rect

-> Represents scope of my system

Exception : using Rzpay - then shouldn't be added.

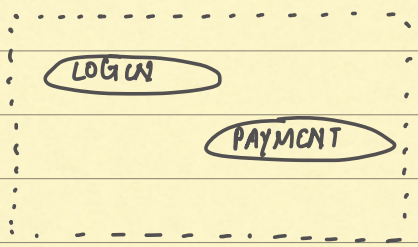
∴ No outsourced feature is included.
ONLY OUR SYSTEM

2-> usecase [OVAL]



Every usecase is feature / behaviour
(Action in our system)

:- usecase should be verb



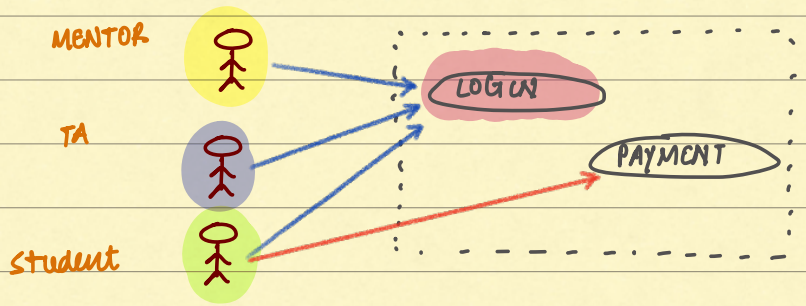
Represented in OVAL

3.> Actor:

Actor: who would use a particular usecase

(verb)

Actor should be - NOUN



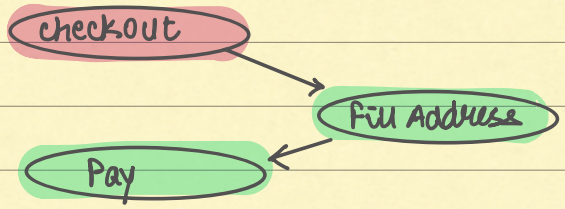
Represented by Arrow - relation b/w Actor & usecase

** 4.> Includes

Eg: Checkout In Flipkart

Q:> Other things required for checkout ?

(Prereq.)

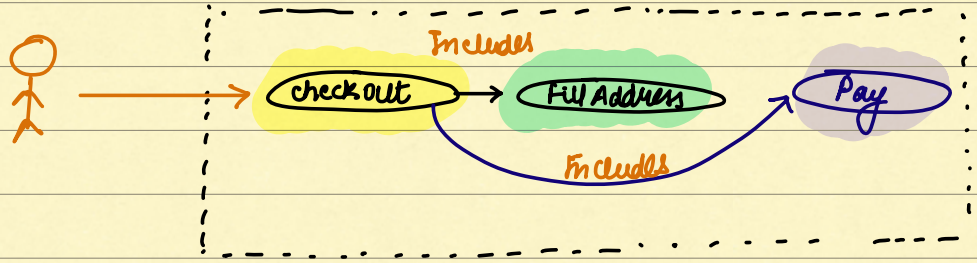


Think of every feature as func:

```
checkout () {
    FillAddress ()
    Pay ()
}
```

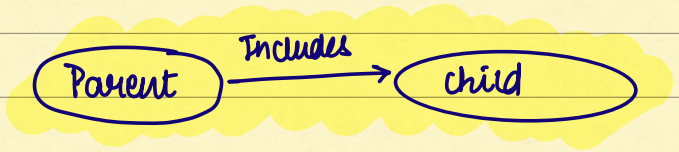
```
//
//
```

Note: Can say: Parent includes child usecase



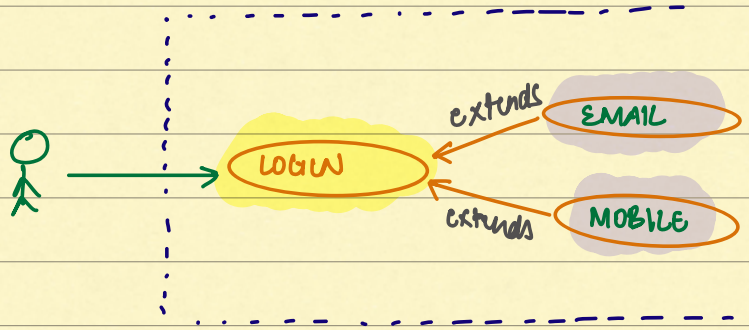
direction of ARROW: Always Parent → child

∴ so as to Implement checkout; you need to Fill Address.



5.) Extends

Whenever 1 feature has many variants



∴ Multiple ways to LOGIN < Email / Phone

direction → Always from specific feature → Generalized

ASSIGNMENT:

Draw usecase diagram for scaler

a.> Atleast 5 usecases

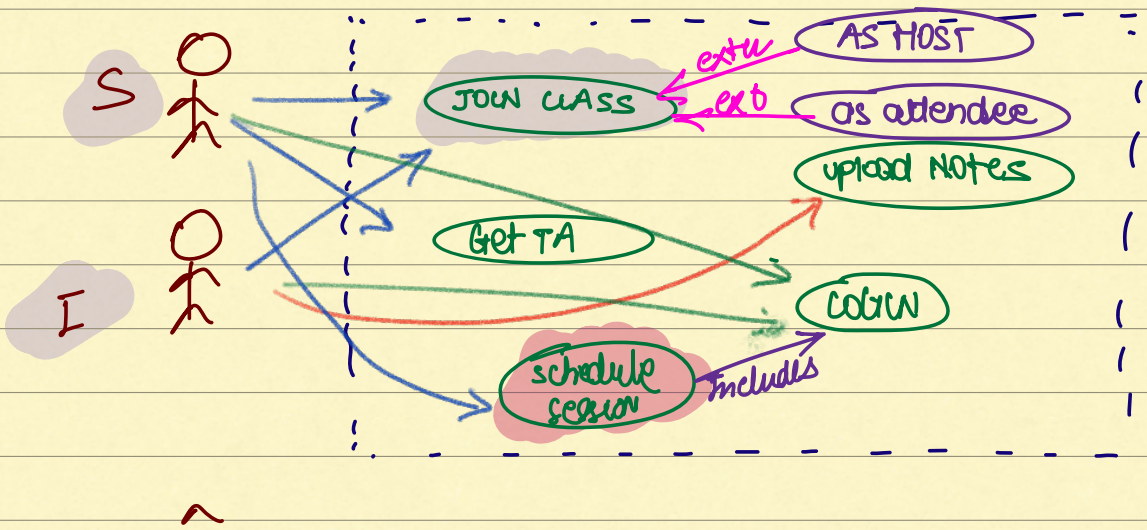
b.> Atleast 2 Actors

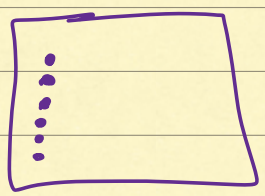
c.> Atleast (1) usecase should involve another usecase

d.> Atleast (1) usecase should Ext. another usecase

10 MARK

① JOIN class
② upload Notes
③ Get TA help
④ LOGIN
⑤ schedule session





*] CLASS DIAGRAM:

① Represents diff entities in s/w system

↓

classes / Interfaces.

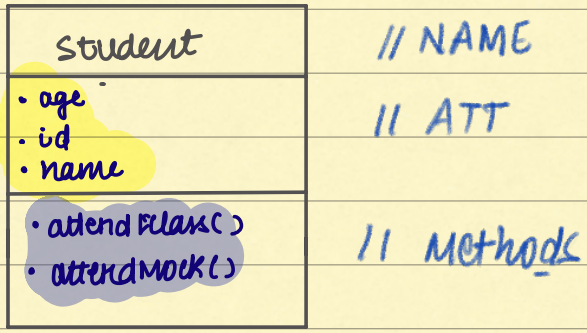
② Also; represents relationship b/w those Entities

eg: ① Which class Implements Cl)
② Who extends class
③ Having Attributes.

*] REPRESENTATION:

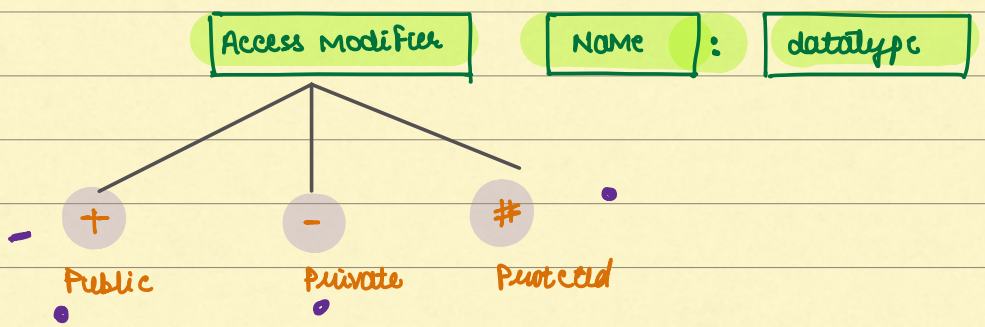
1> class [Rectangle]

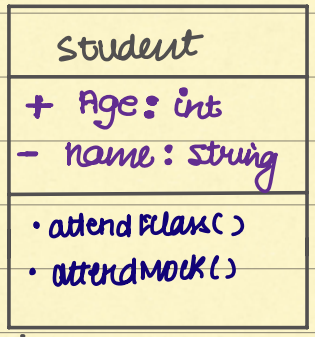
Contains (3) things
·> name
·> Attributes
·> methods



Representing Methods / Attributes :

1-> Attribute



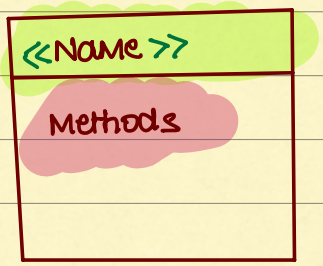


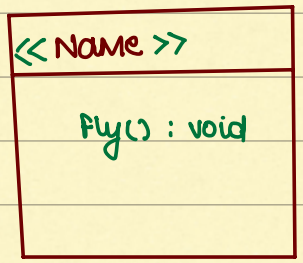
2.> Methods

modifier signature : datatype

Eg: + changeBatch(str, str) : void

2.> Interfaces [NO Attributes]

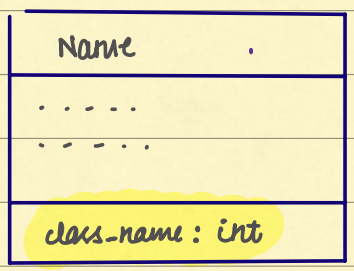




- Name is enclosed by ≪ ≫
- All Methods are public - by default

3. STATIC ATTRIBUTES:

- create another block / section

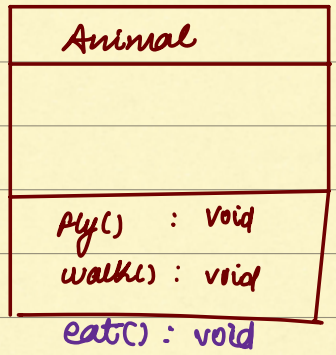


// Static

4. ABSTRACT CLASS:

exactly like Normal class - (1) difference.

•> Represented as italics



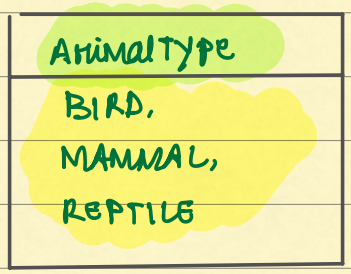
Animal

✓ itali

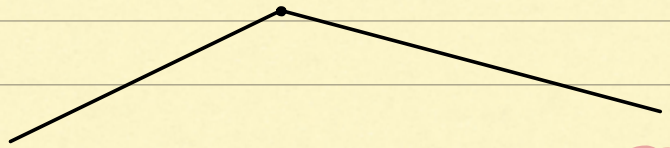
<< >>

5.> ENUMS

Represented by Name & , separated values



*] REPRESENTING RELATN B/W CLASSES:



Inheritance

Parent ⇝ child
interface ⇝ Implemn.

basically:
extends / Implements

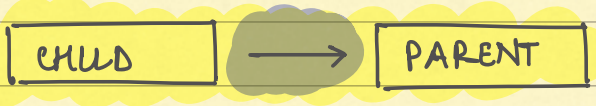
Association

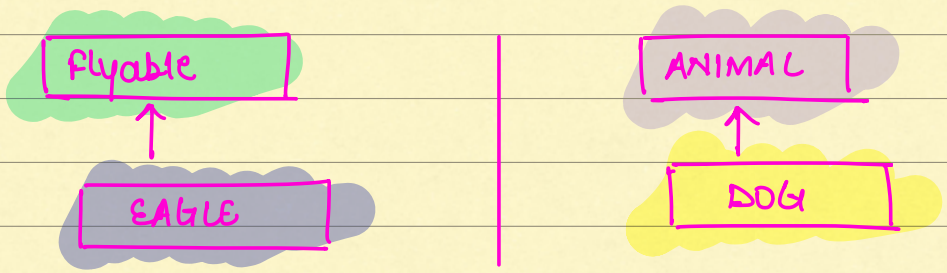
(Has-a relationship)

Having attribute of other class.

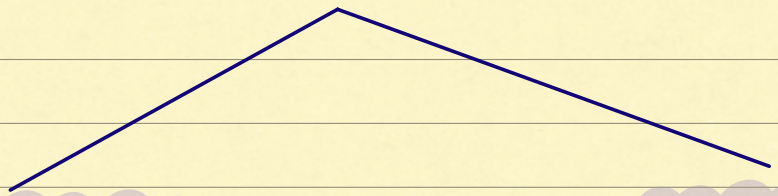
*] INHERITANCE:

Always from child → Parent





★] ASSOCIATION :



COMPOSITION

AGGREGATION

strong association

AGGREGATION

collecting

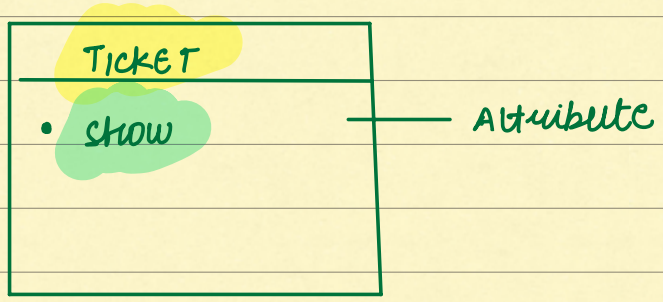
i.e. collecting ② things / classes that existed Independently.

(Relation b/w ② classes - both Independent.

↓

Object of both can exist Independently.

Eg: BMS



show can exist independently without ticket ?

i.e. (A) aggregates (B) even if Obj of (A) is deleted; (B) remain in system

2.) COMPOSITION

[create something]

Relation when (A) composes (B) - if obj(B) cannot exist w/o obj(A)

eg: House & door /
class & assignment

Aggregation - collection
Composition - creation

In composition :

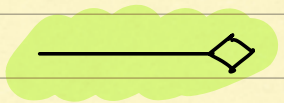
if 2nd class doesn't have independent existance

↳ Both classes are strongly associated

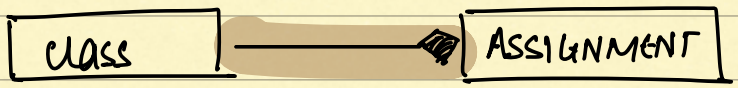
Hence; called STRONG ASSOCIATION

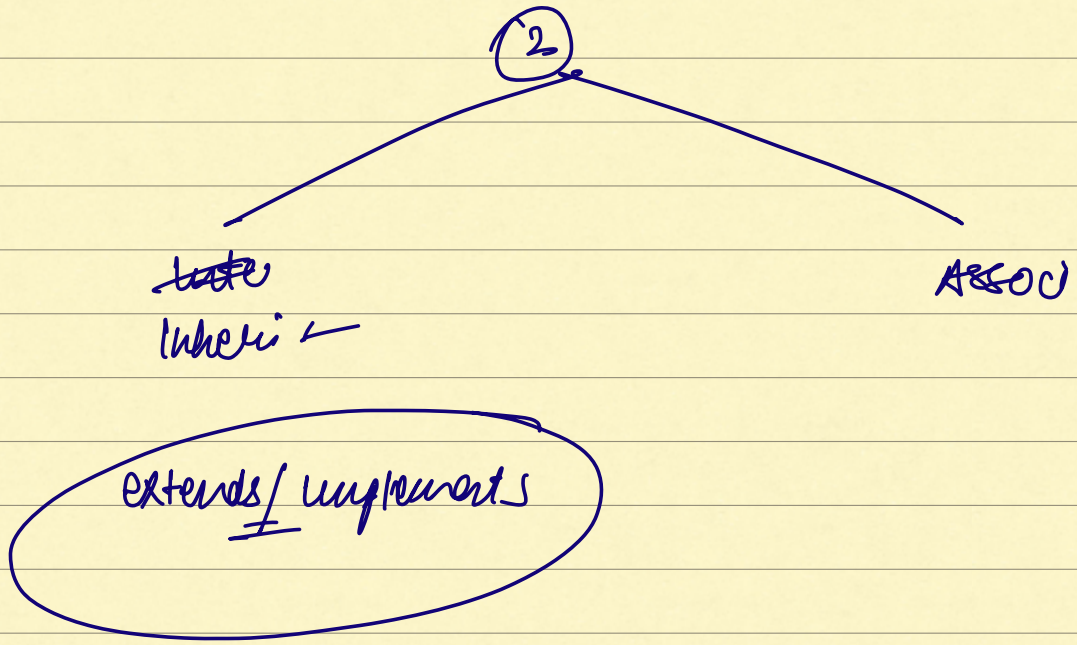
REPRESENTATION:

1.> weak association

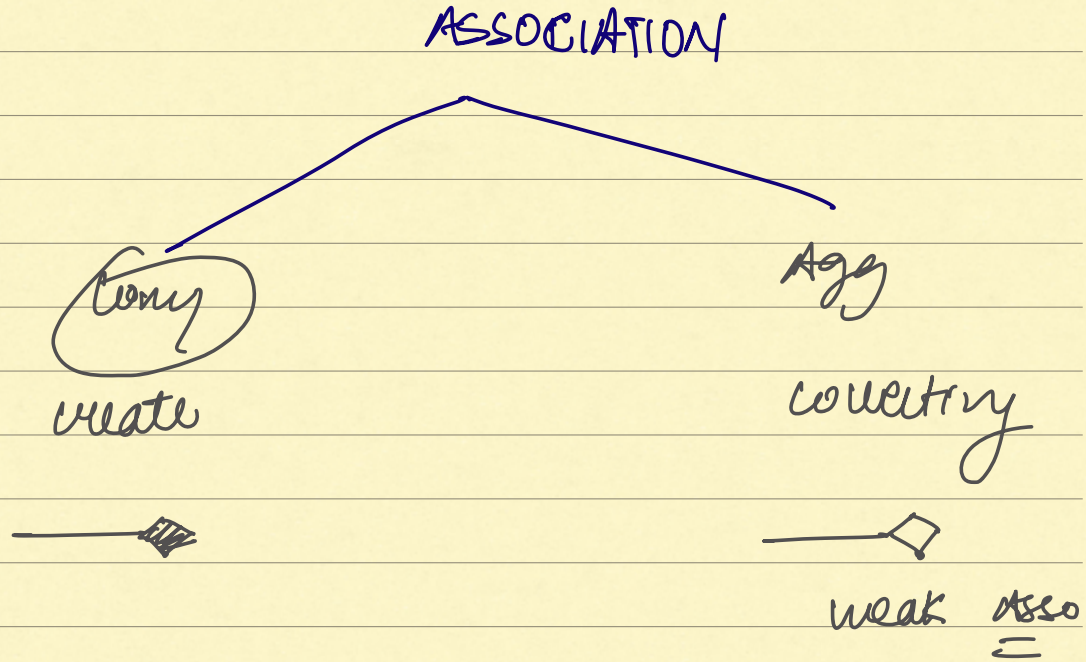


2.> strong Association




2
~~Gate~~
Inheri ←
ASSOCI
extends / implements

C → P


ASSOCIATION
Comp
create
Agg
containing
weak Asso

BMS