Intro to LLD and OOP

1. Intro to LLD
2. Why is LLD important
   - → Job
   - → Interviews
3. LLD Module Curriculum

Start learning ⇒ 4. Intro to OOP

Intro to LLD

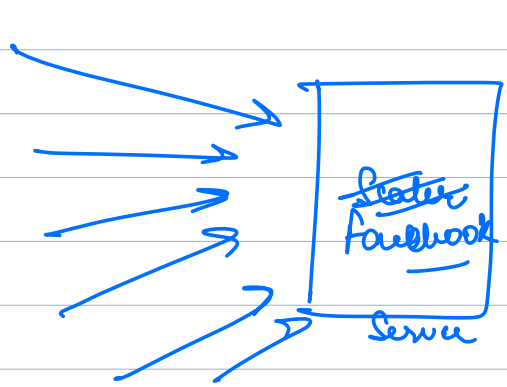
L → ~~low~~ High

L → Level

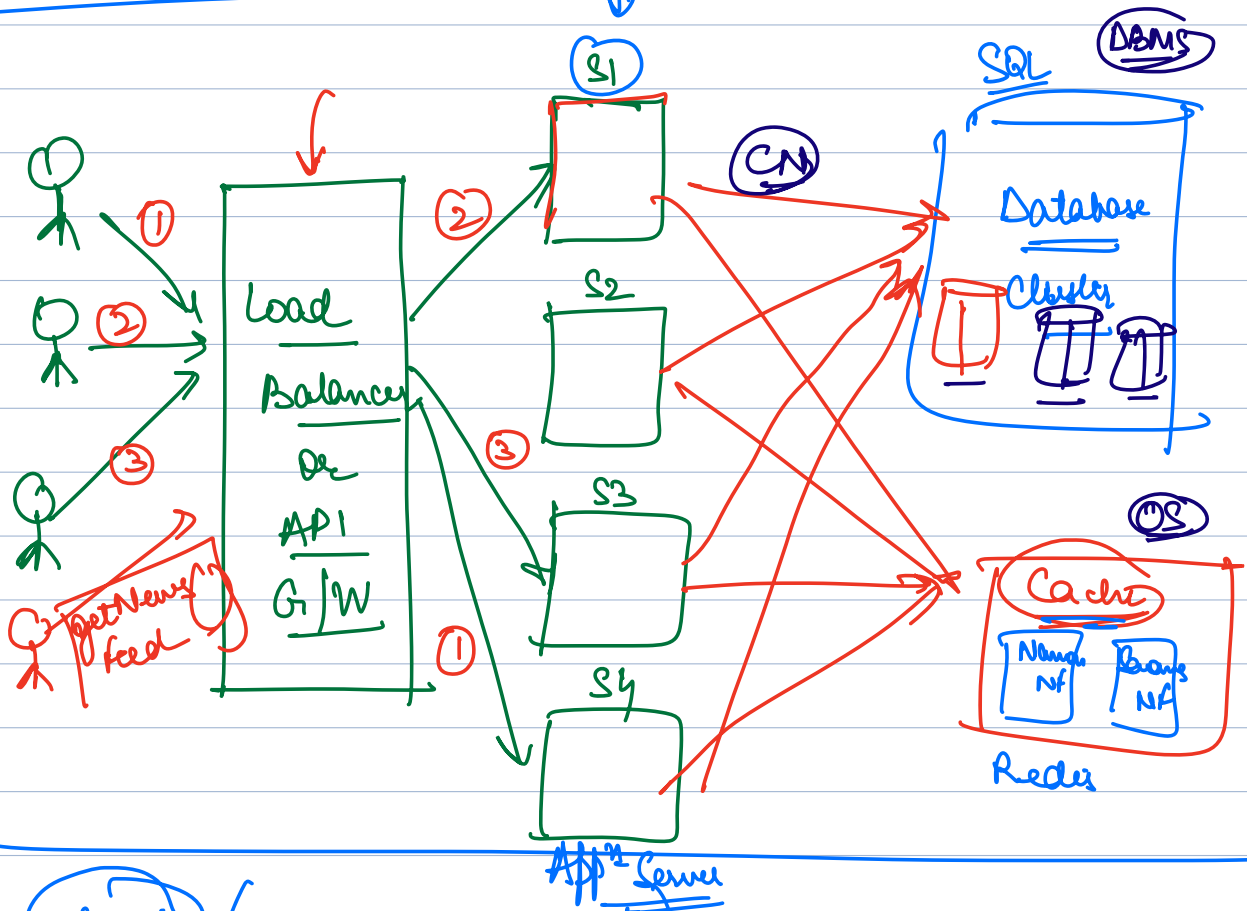
D → Design

High Level

- → Overview
- → Not going into detail
- → Bird's Eye View
- → Just providing structure
- → Basic Idea



Overview of how a software system is going to work



login (username, password)

High level Design : Overview of diff infra layers that work together to serve a S/W system

Low Level Design

→ Details

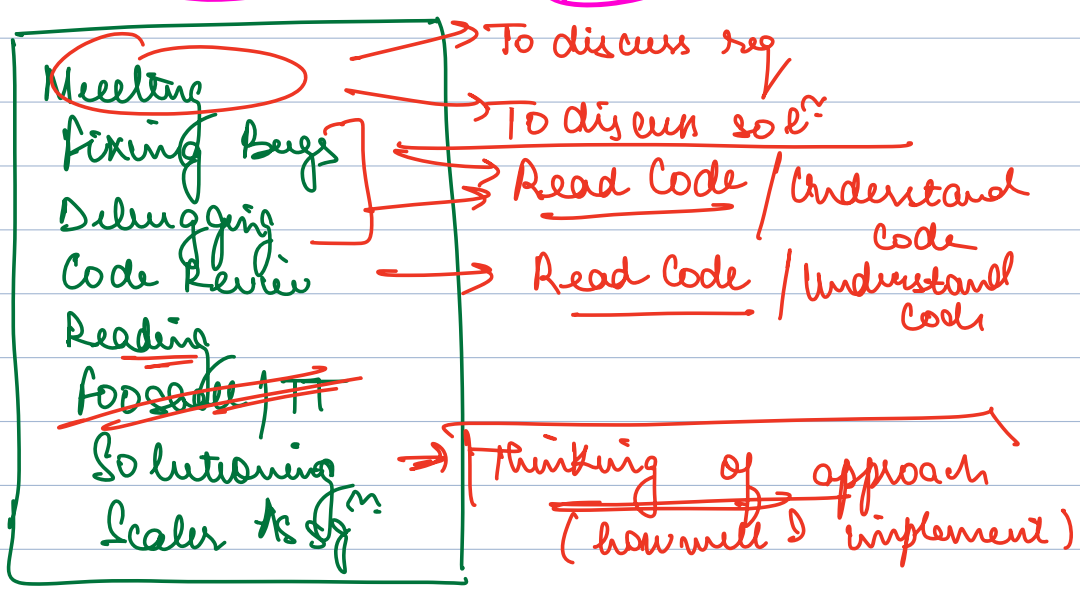
How a computer works / what it acts like depends on S/W that is running

- ⇒ going into details of H/W
- ⇒ how exactly should a S/W be structured
- ⇒ code should be written

Sees ⇒ 40 hrs

≈ 12% of their working day actually writing code.

1 LPA ≈ 10 LPM



≈ 70% of your day reading code or other

pre code → gathering req, planning impl

post code → code review, debugging, testing

⇒ Knowing about OO concepts makes ~70% of your day better

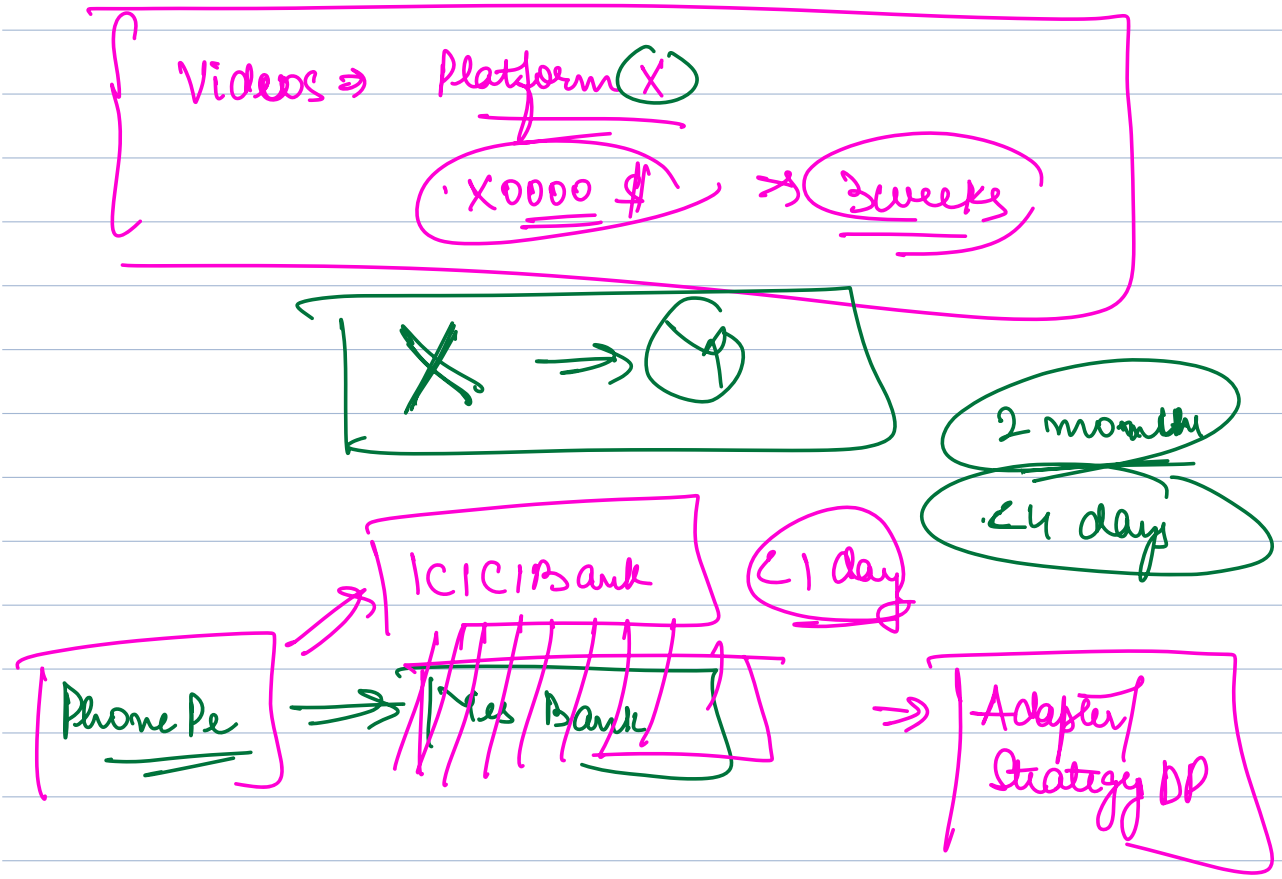
1. Gather Req
2. Readable / Understandable
3. Extensible
4. Maintainable

Maintainable ⇒ Keeping the set of features that are implemented in a working state (NO NEW FEATURES)

Reasons

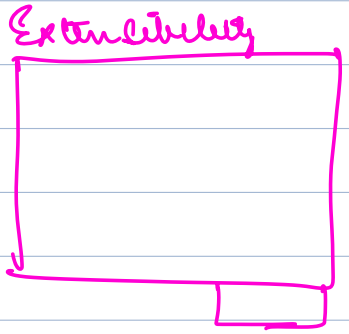
- Bugs
- 3rd Party Dependencies → log4J
- Infra Upgrades / Platform Upgrades

Windows XP



Extensibility : How easy is it to support new requirements in your system.

Regression ⇒ On one update something that was working previously stops.

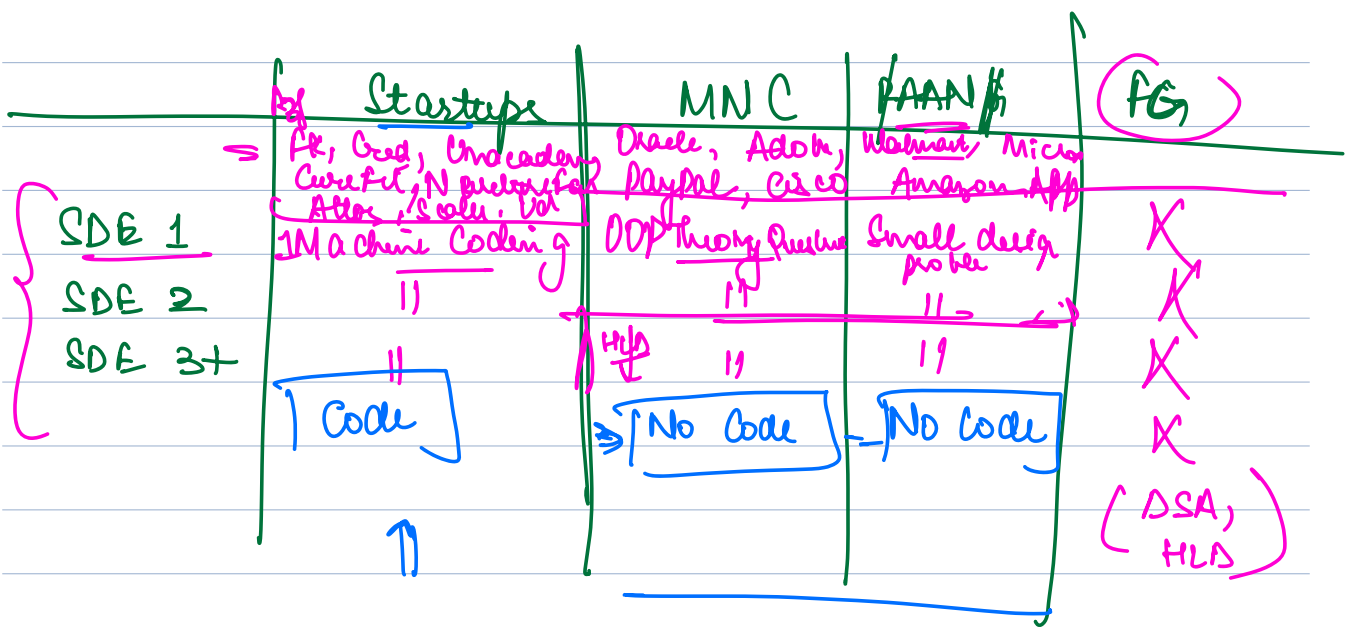


Why is LLD imp

→ Every startup is going to have atleast 1 LLD Round

→ Even many big MNCs ask LLD in their interviews.

0 - 2 yoe	→ SDE 1
2 - 5 yoe	→ SDE 2
5+ yoe	→ SDE 3+



LLD Curriculum

Java → Both code + Theory

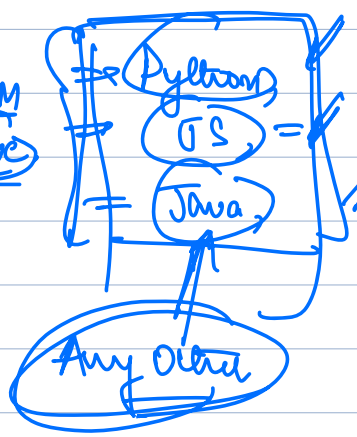
1. OOP → 4 classes ✓
2. Design Principles (SOLID) → 2 ✓
3. Design Patterns → 5 ✓
4. UML Diagrams → 1 ✓
5. How to approach LLD Interviews → 1 ✓
6. Case Studies

Language Agnostic.

Implement APIs, Test Cases, Command line → SpringBoot, ORM

→ TicTacToe
→ Parking Lot
→ Book My Show
→ Splitwise

→ ORM
→ MVC



Why Java

1. Interviews
2. Verbosity

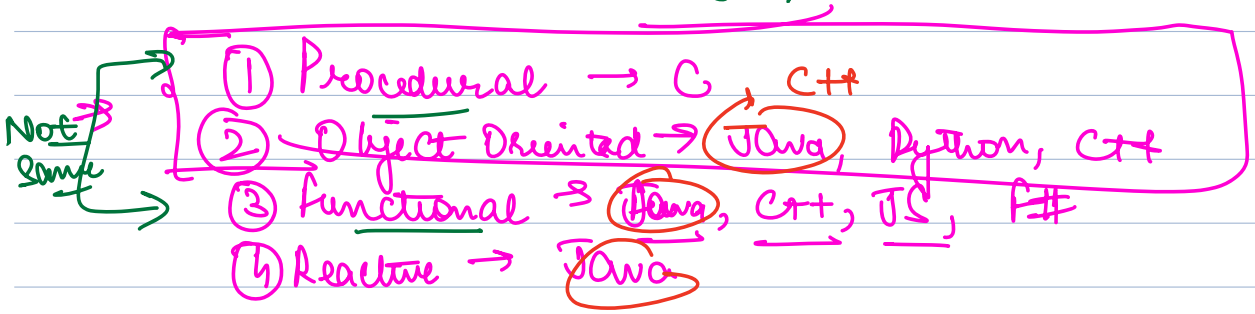
create variable
if ( )
for ( )

Break till 10:25PM

Intro to OOP

Programming Paradigm (Way to organize code)

→ Each programming language support multiple prog paradigms



PROCEDURAL PROGRAMMING

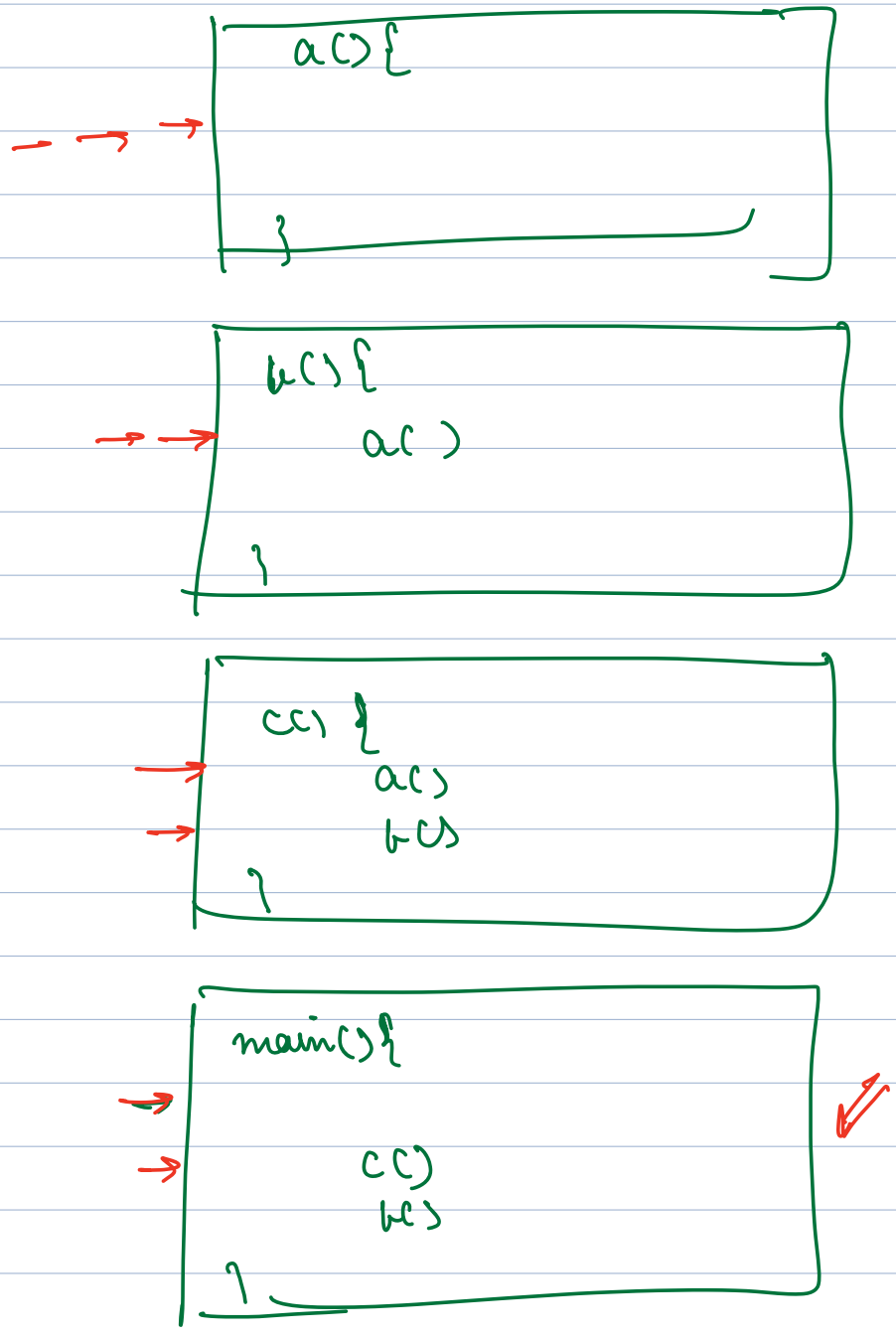
⇒ Procedure → Set of instructions

↳ Old name for function/methods

→ We organize the code into a bunch of procedure

→ each procedure may internally call other procedures

→ execution of a prog starts from a special procedure. (typically called main)


a(){
}
b(){
a()
}
c(){
a()
b()
}
main(){
c()
b()
}

Problem with Procedural Prog

We are studying DS

Naman is teaching DS

We are understanding DS

I am learning about procedural

Subject + Verb

Someone Something

```
print Student (String name, int age, String gender) {

    Sout (name)
    Sout (age)
    Sout (gender)

}
```

Non OOP Language (C++/C)

```
Struct Student {
    String name
    int age
    String gender
}
```

// coll^n of attributes

① ⇒ Coll^n of attributes

② like classes but w/o any methods

Void

```
print Student ( Student stud ) {
    cout ( stud. name )
    cout ( stud. age )
    cout ( stud. gender )
}
```

Subject — Student stud — Someone

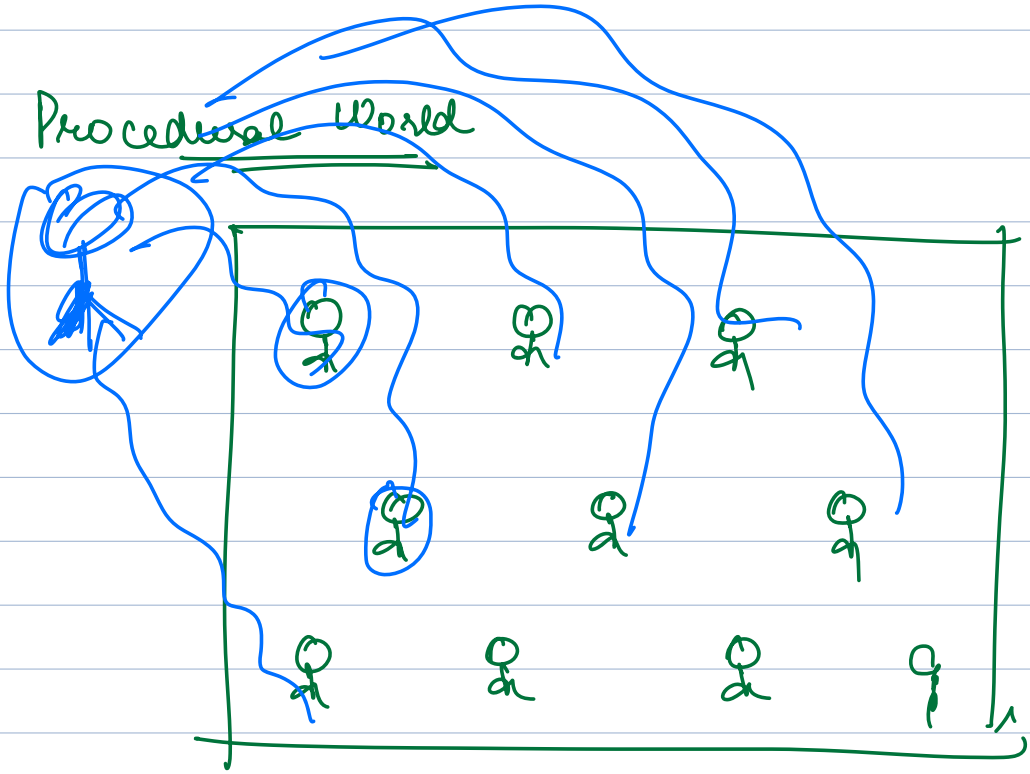
Something

⇒ procedure can access any data of the Struct

⇒ it can change any data

IRL ⇒ Someone. Something() student. print()

Procedural ⇒ Something (Someone) print (student)



→ No action attached to the data
→ Entities are just a collⁿ of data
→ Dep procedures take entities and do some comn
⇒ In a big enough system, procedural prog makes code understanding difficult

⇒ Entities are just data. They have no free will

(AL) entities are core to the concept

Procedural ⇒ procedures are core entities

DOP ⇒ Overhead of creating object

↓

Additional req
- → CPU
- → Memory
- → Perf

OOP

⇓

⇒ Programming paradigm centered around entities

→ every entity has data + actions that they can perform

⇒ actions and data are grouped together.

⇒ more natural to how humans make sense of real world.

1 Principle of OOP ⇒ values, guidelines

⇒ abstraction

3 Pillars of OOP =

⇒ help implements abstraction

→ Inheritance → Polymorphism

→ Encapsulation

Next Class

1. Abstraction
2. Encapsulation
3. Classes
4. Objects
5. Constructors
6. Deep and Shallow Copy

Frontend : TS + React

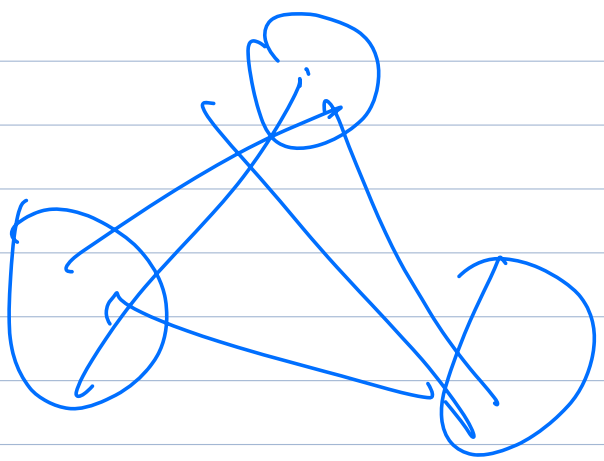
Backend : Java + Spring

⇒ Python + Django

Udemy: Chad Darby

Cons of Procedural

1. Diff to make sense of a complex system
2. Diff to debug
3. Spaghetti code



Revise Transactions

Funcⁿ Prog ⇒ Functions are treated like objects

doSomething(void xyz) {

}