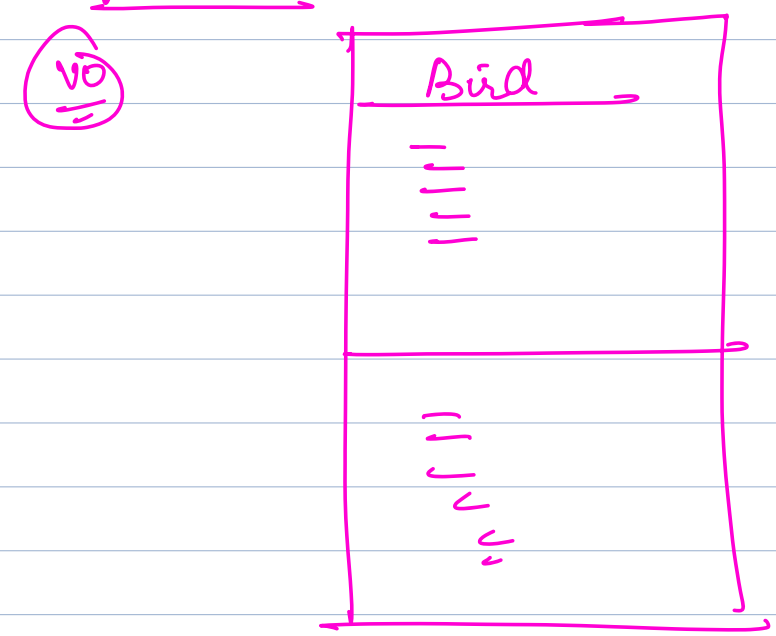
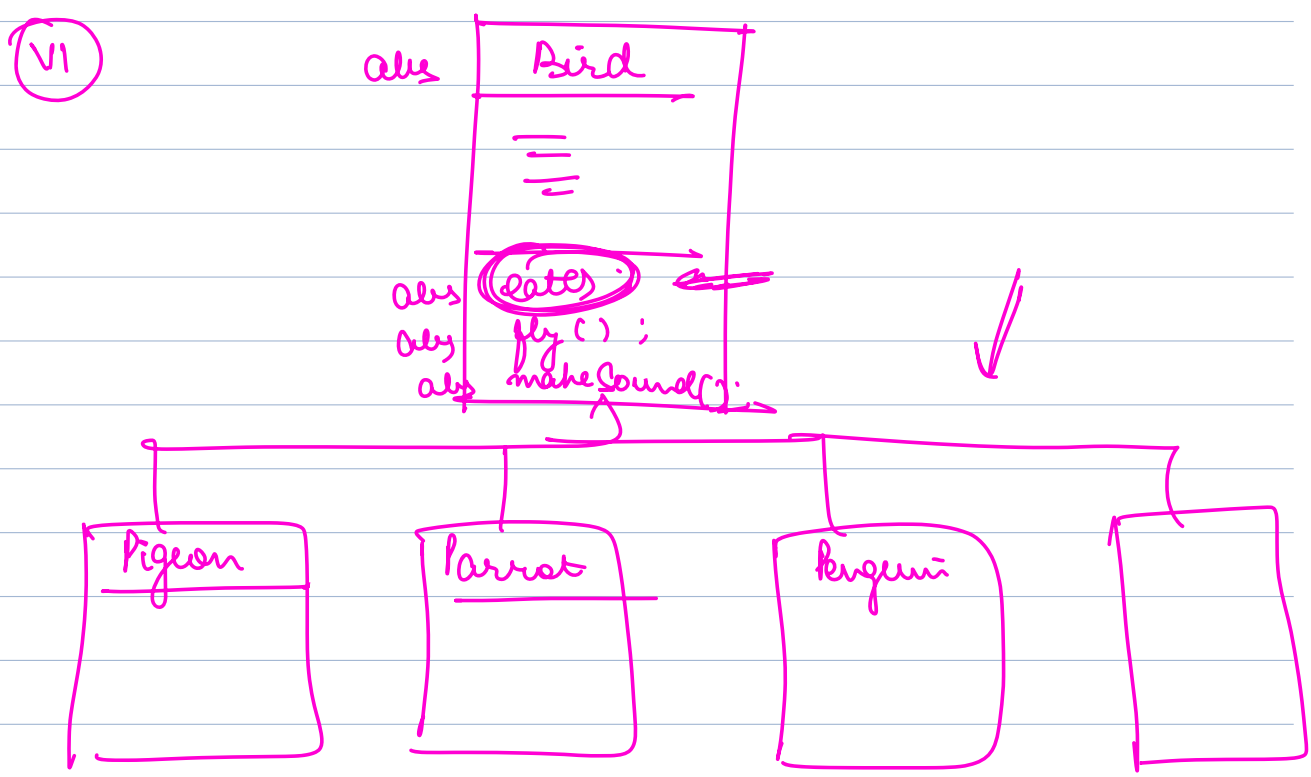
L
1
D

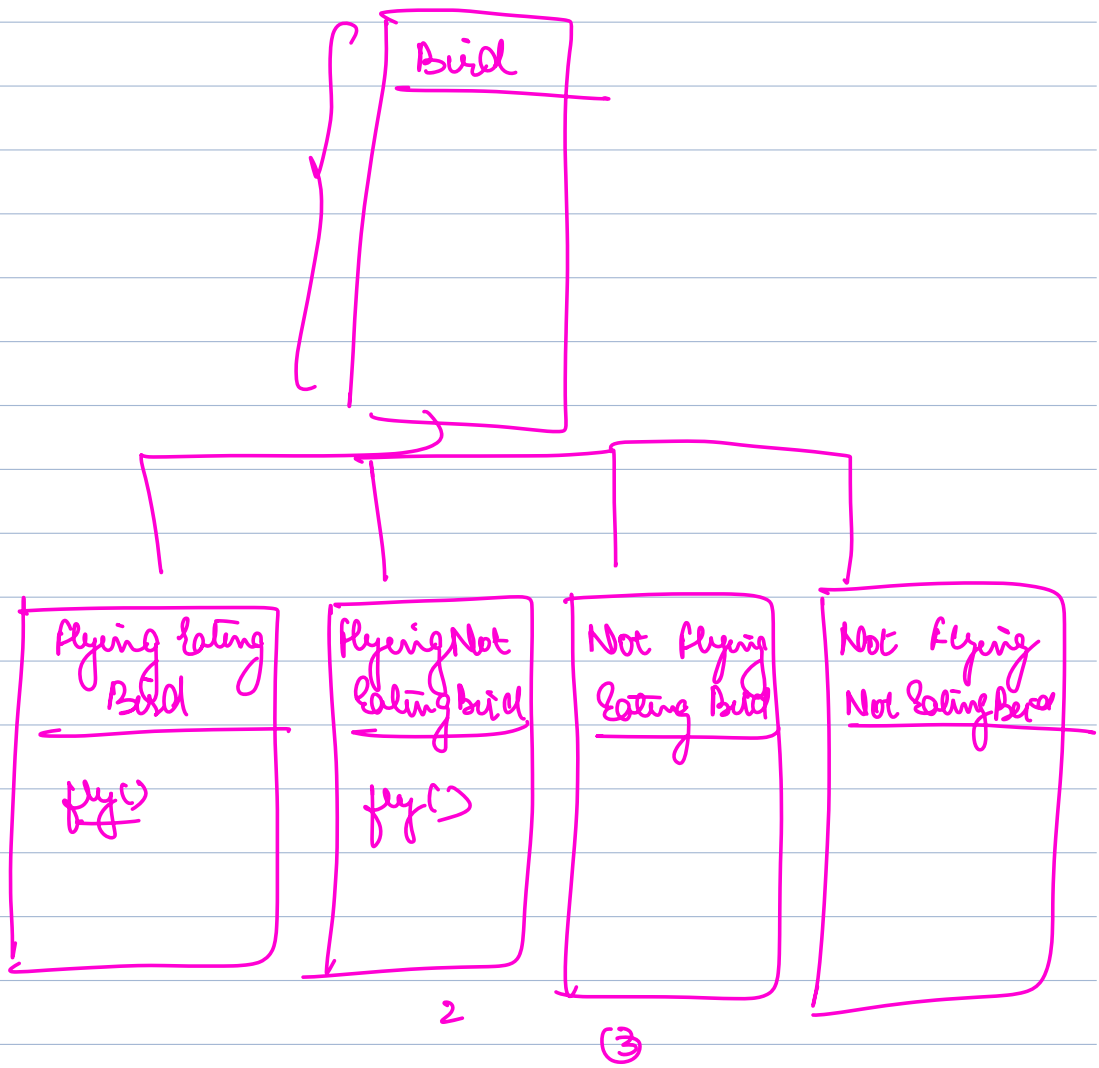
Design a Bird



⇒ SRP X
⇒ OCP X



V2

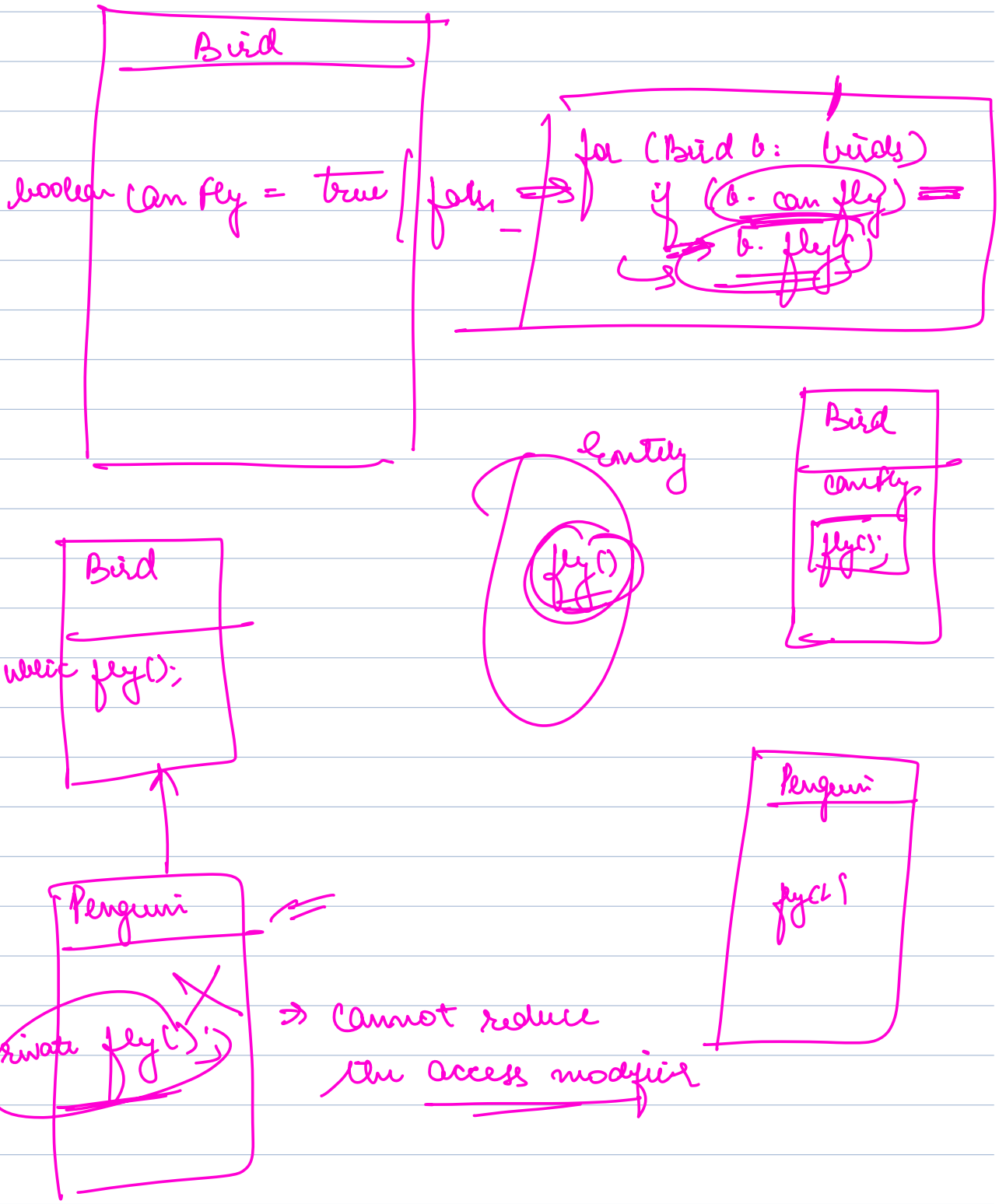


① 2^n classes ⇒ Class Explosion

Birds who can fly

List <Bird> a = +——————————+

for (bird : a)
    bird.fly()

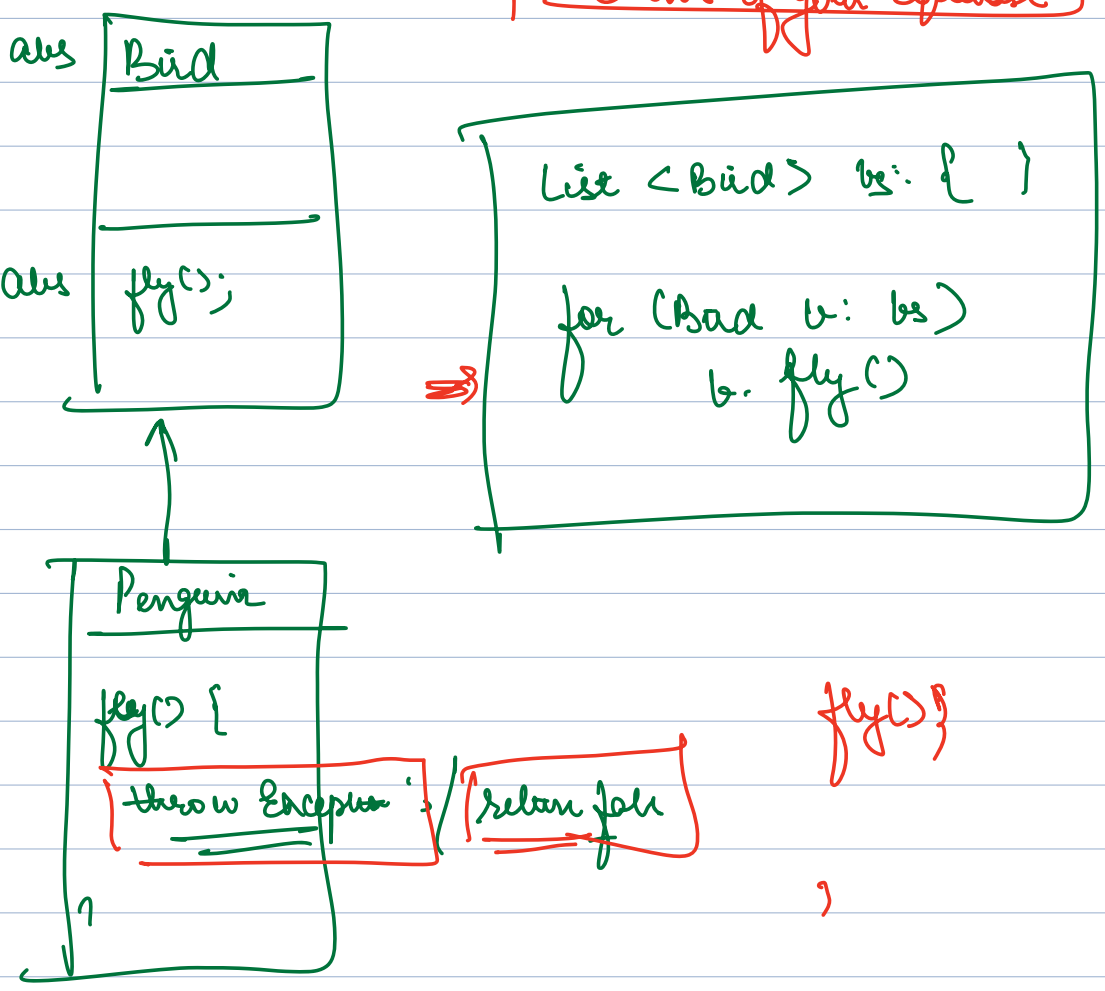

Bird
boolean canFly = true
fly()
for (Bird b : birds)
if (b.canFly)
b.fly();
Bird
canFly
flyC);
Sentry
fly()
Bird
public fly();
Penguin
fly();
Penguin
private fly();
=> Cannot reduce the access modifier

List <Bird> l = { }

for (Bird b : l)
    b.fly()

→ fly() method not_there

Reduces surprises for clients of your codebase



L ⇒ Liskov's Substitution Principle

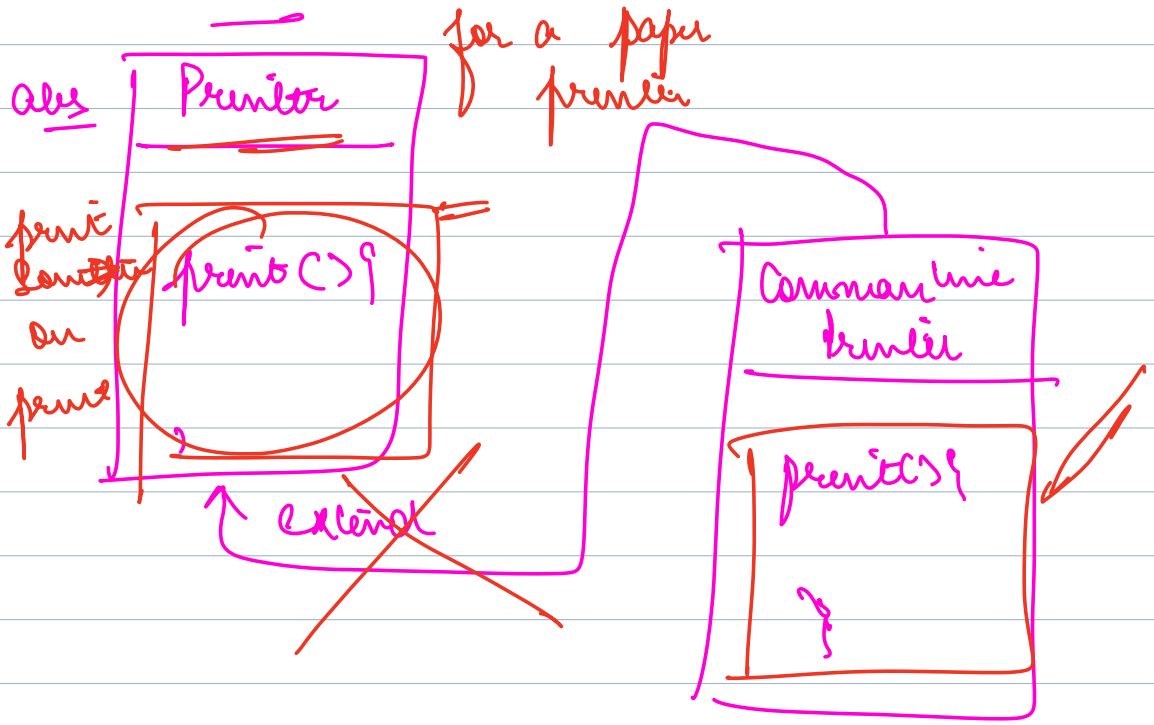
⇒ Objects of a child class should be As-is substitutable in the variables of the parent class.

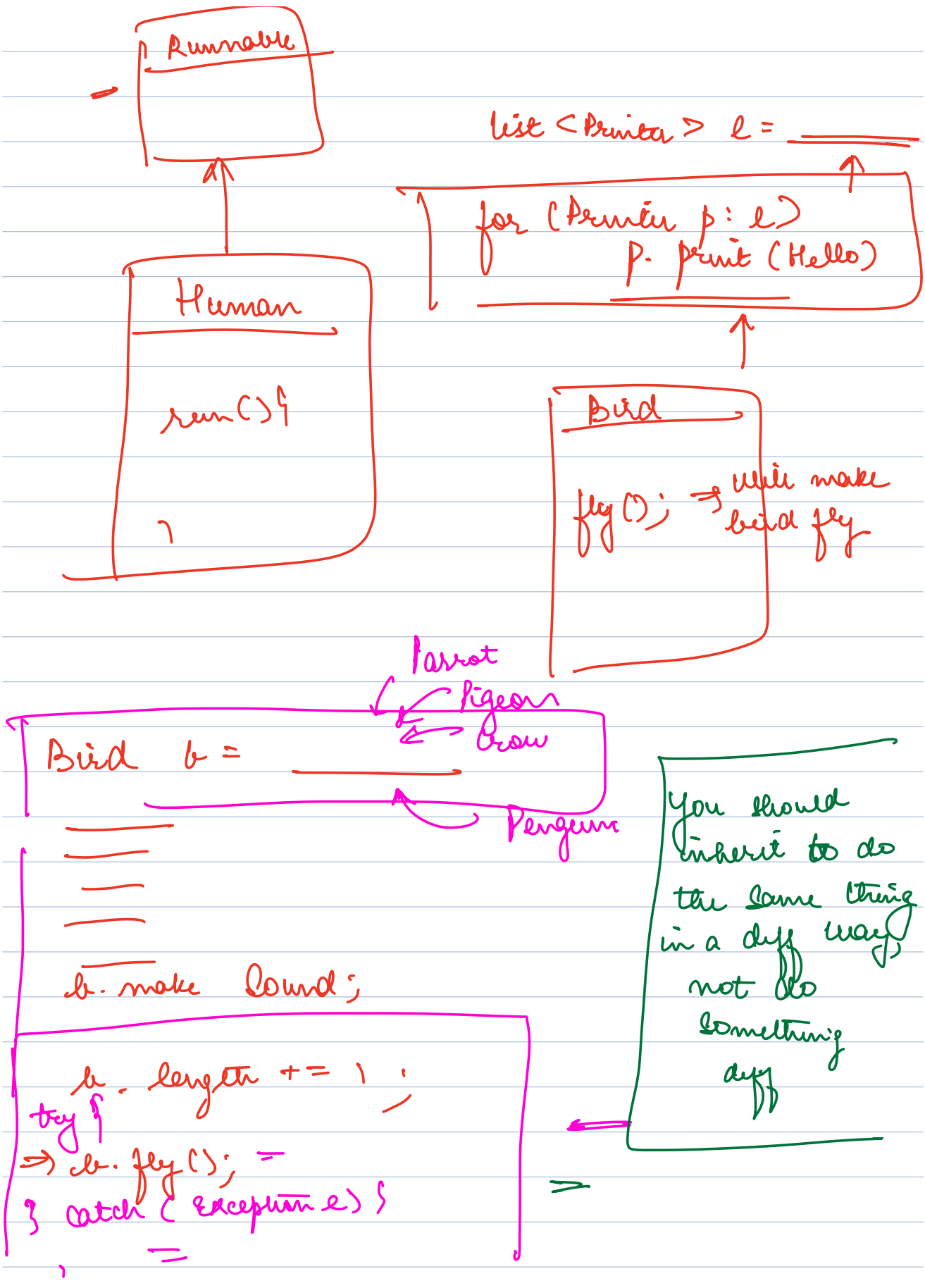
⇒ No change should be required in the codebase to accomodate the child class

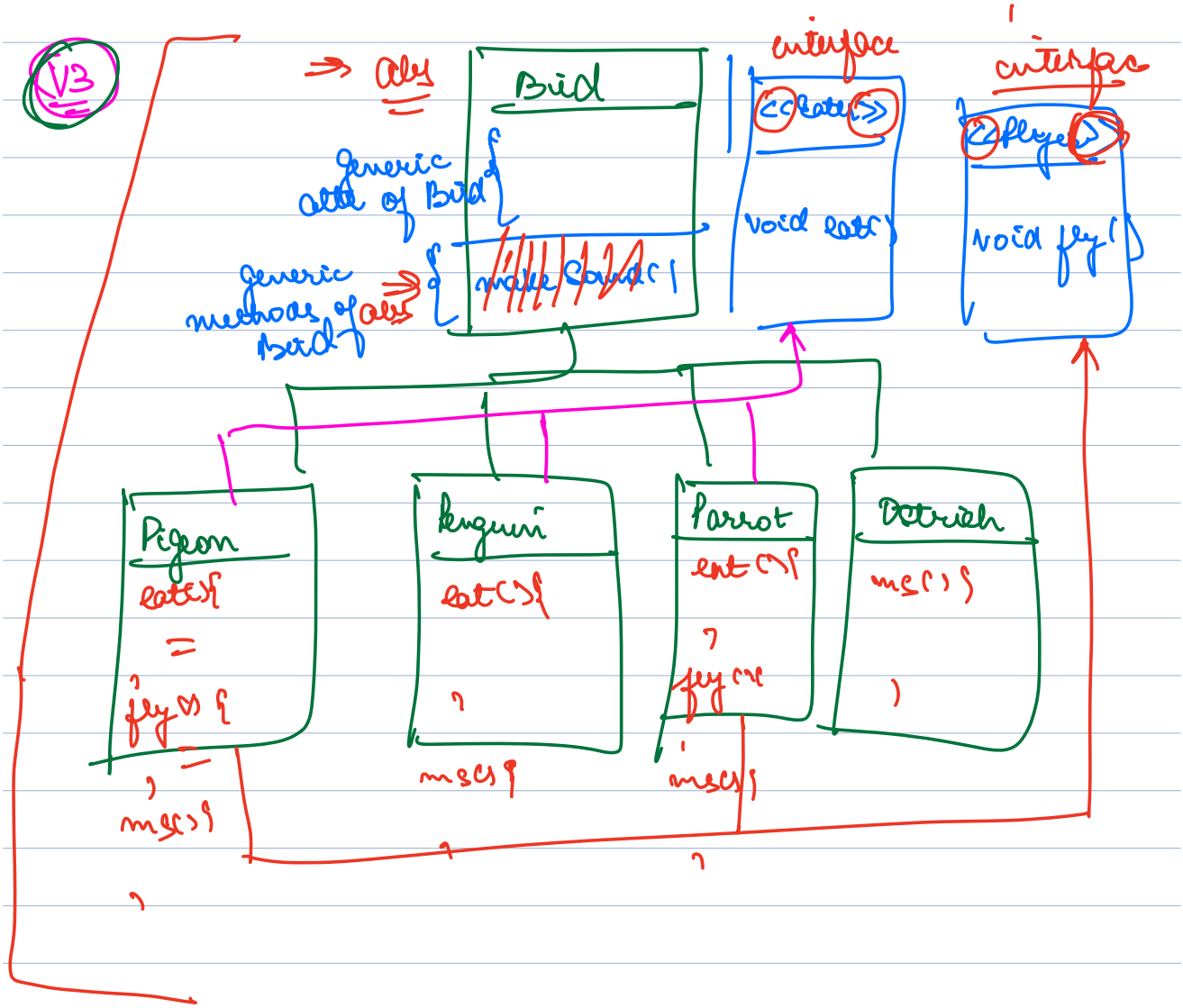
⇒ No child class should require special treatment

⇒ Child classes should do exactly what their parent class expects them to do.

⇒ Don't give a special meaning to parent's methods.




Runnable
Human
run(){
}
List<Printer> l =
for (Printer p : l)
p.print(Hello)
Bird
fly(); ⇒ will make bird fly
Parrot
Pigeon
Crow
Penguin
Bird b =
b.make Sound;
b.length += 1;
try {
⇒ b.fly();
} catch (exception e) {
You should inherit to do the same thing in a diff way, not do something diff



⇒ Not all birds can fly.

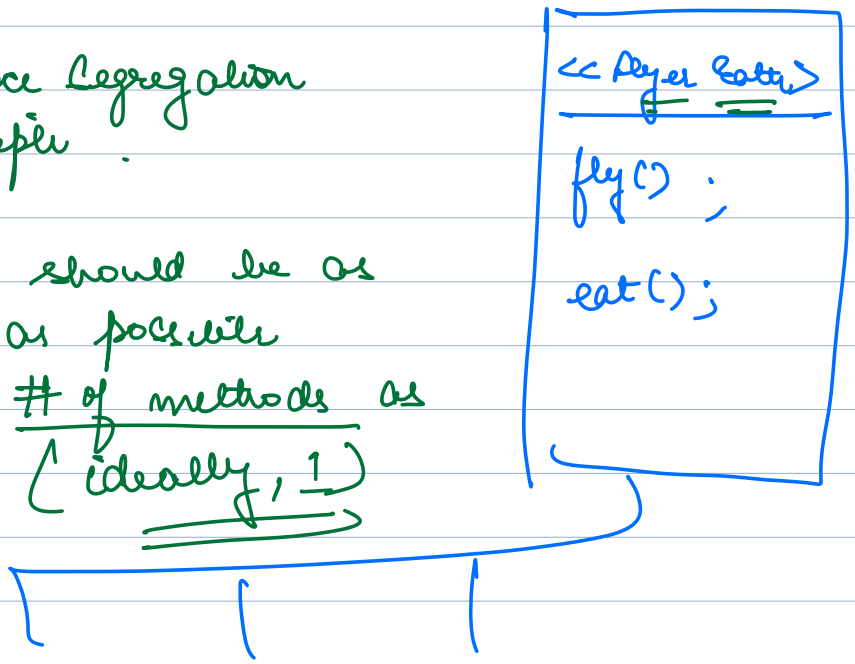
All Birds who can fly can eat and vice versa.

⇒ Both fly and eat go together

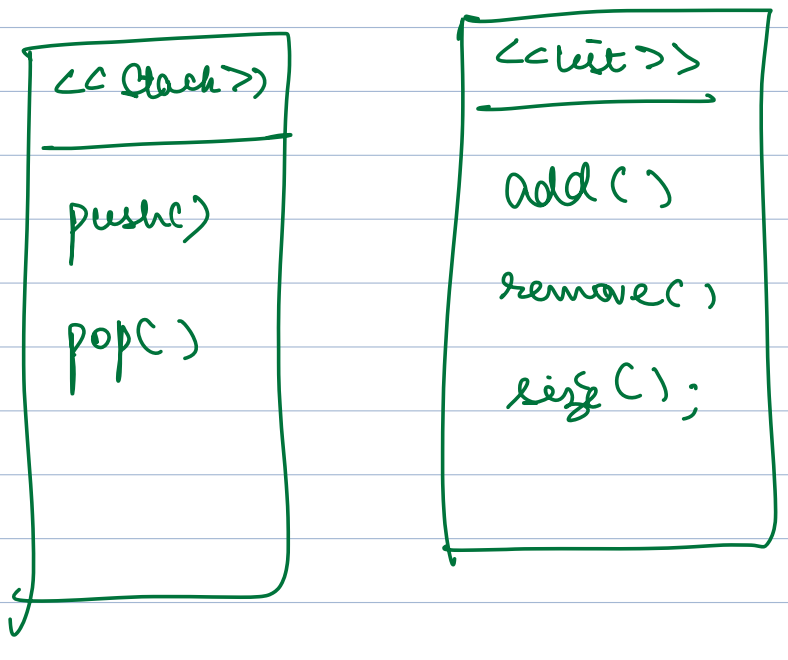
(I) ⇒ Interface Segregation Principle.

⇒ interfaces should be as light as possible

⇒ as less # of methods as possible (ideally, 1)



⇒ interfaces should have those methods that logically belong together / are related to each other



(99%) ⇒ 1 method

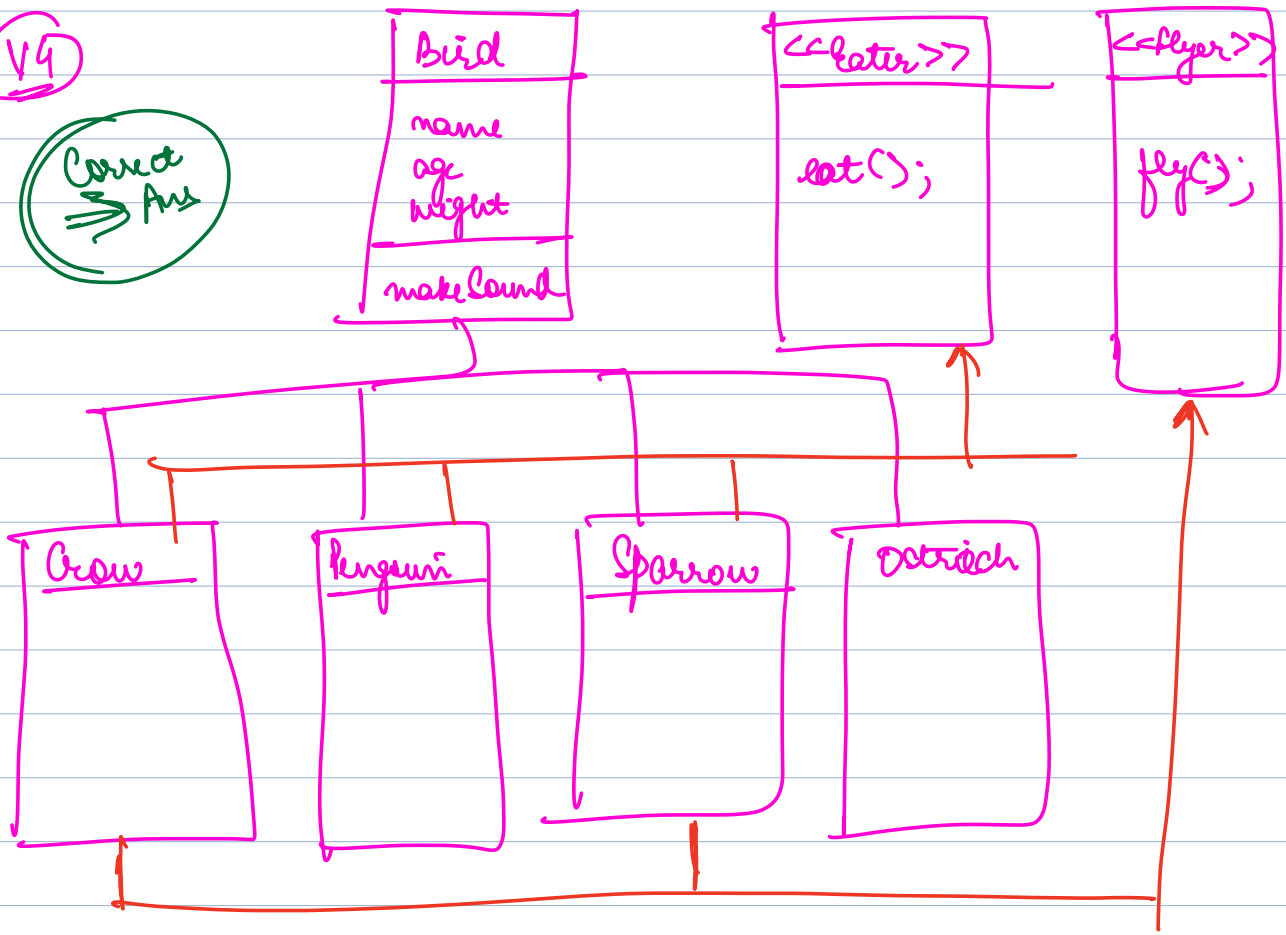
⇒ SRP for interfaces

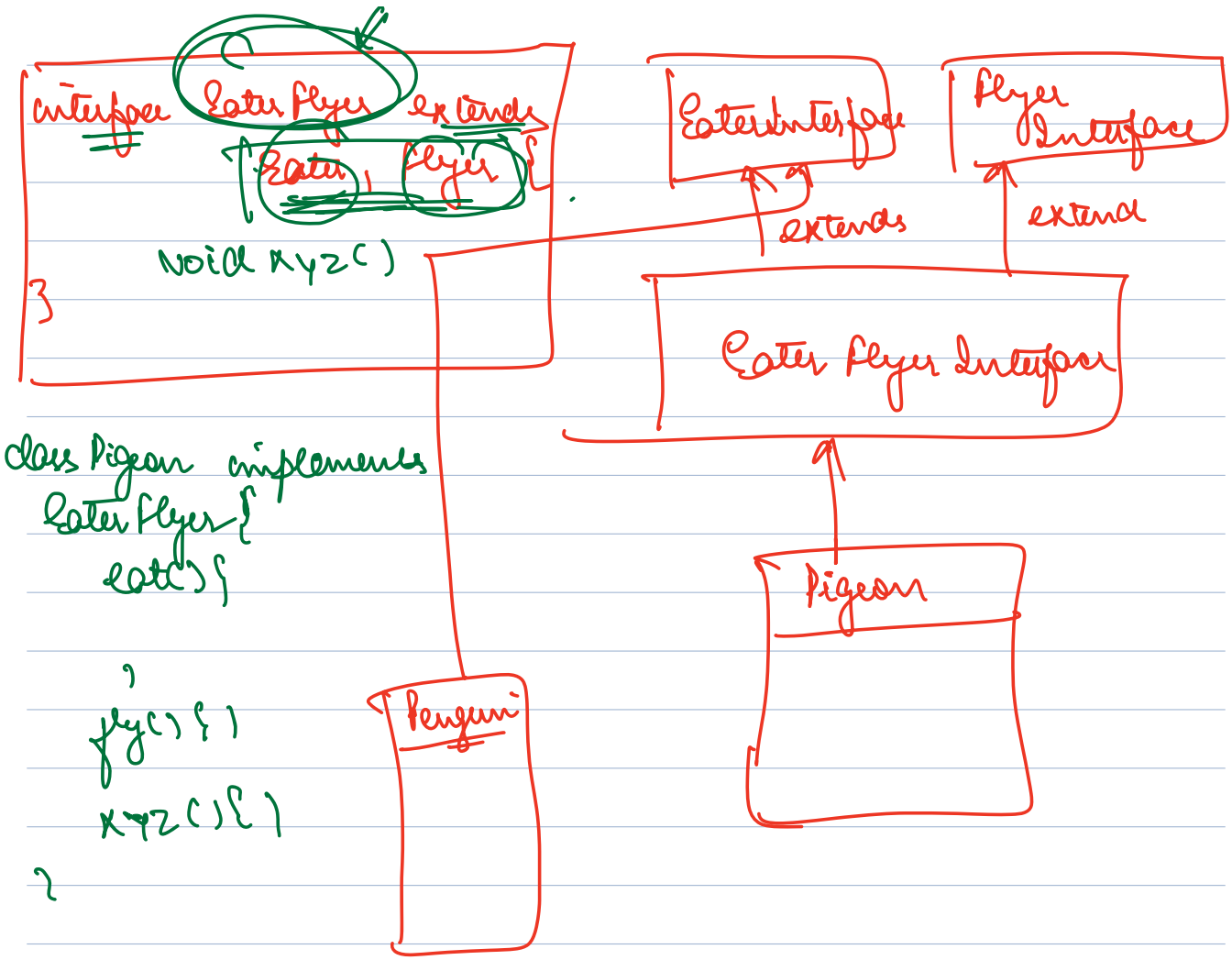
Functional Interfaces : interfaces with only one method.

Lambda Expressions ⇒ interfaces with 1 method.

V4

Correct Ans ⇒



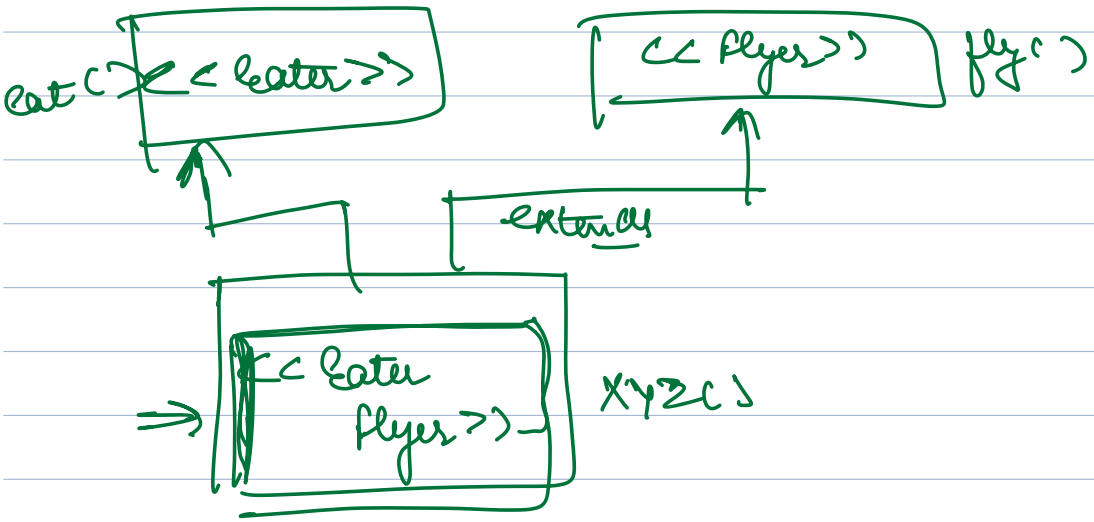



```
class Pigeon implements
EaterFlyer{
  eat(){

  }
  fly(){}
  xyz(){}
}
```

SOLID

```
List<Flyer> l = {new Crow(), new Sparrow()};
```



Class A implements eaterFlyer {

eat() {

}

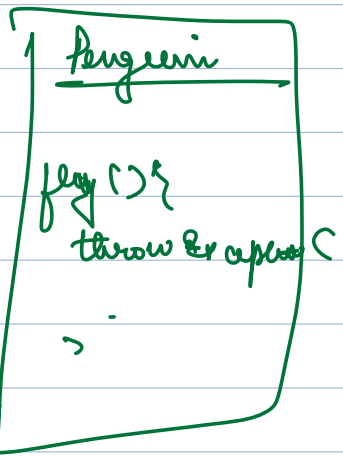
fly() {

}

XYZ() {

}

}



try() {

} catch {

10 ⇒ 10 interfaces

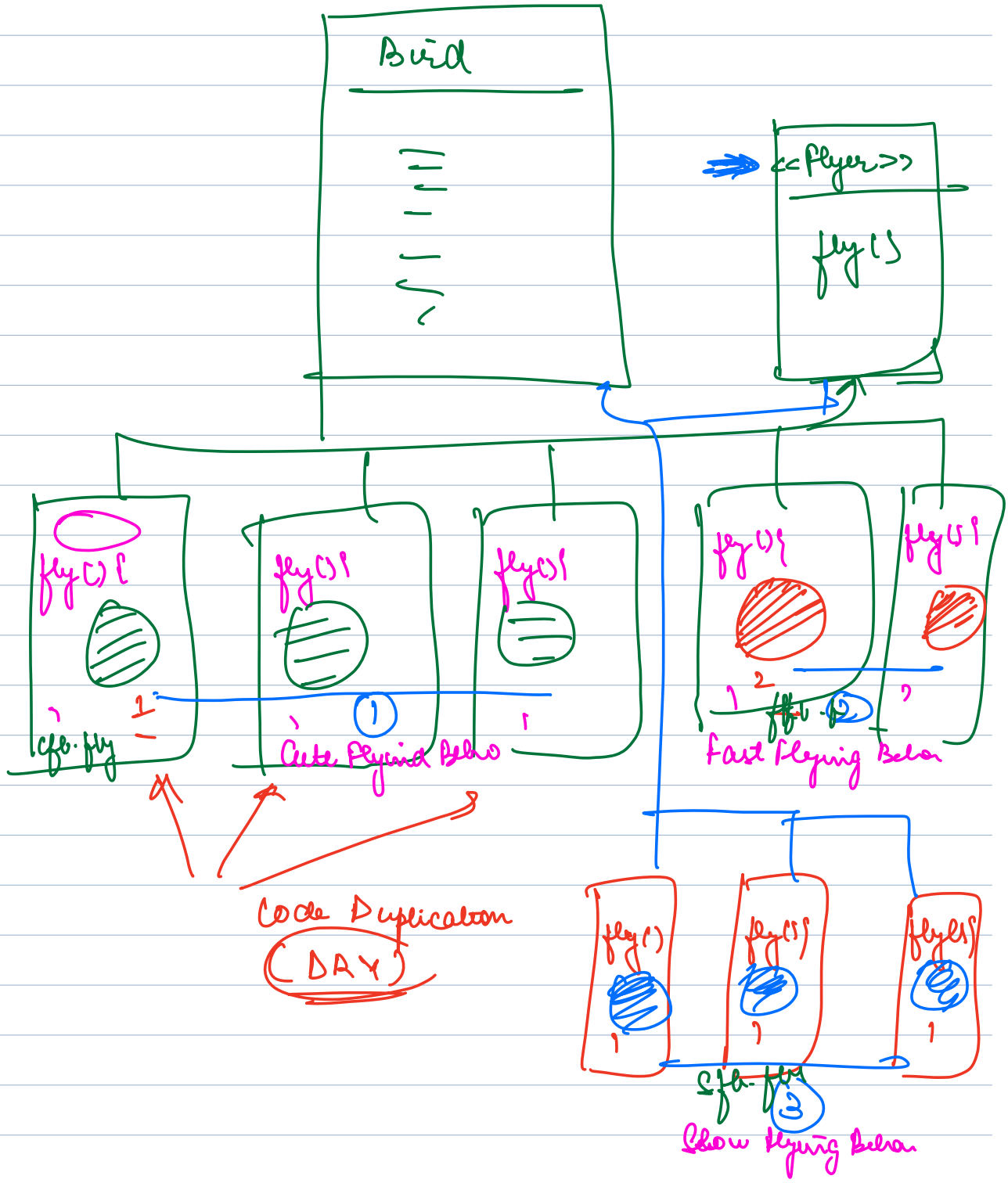
void ~~play()~~

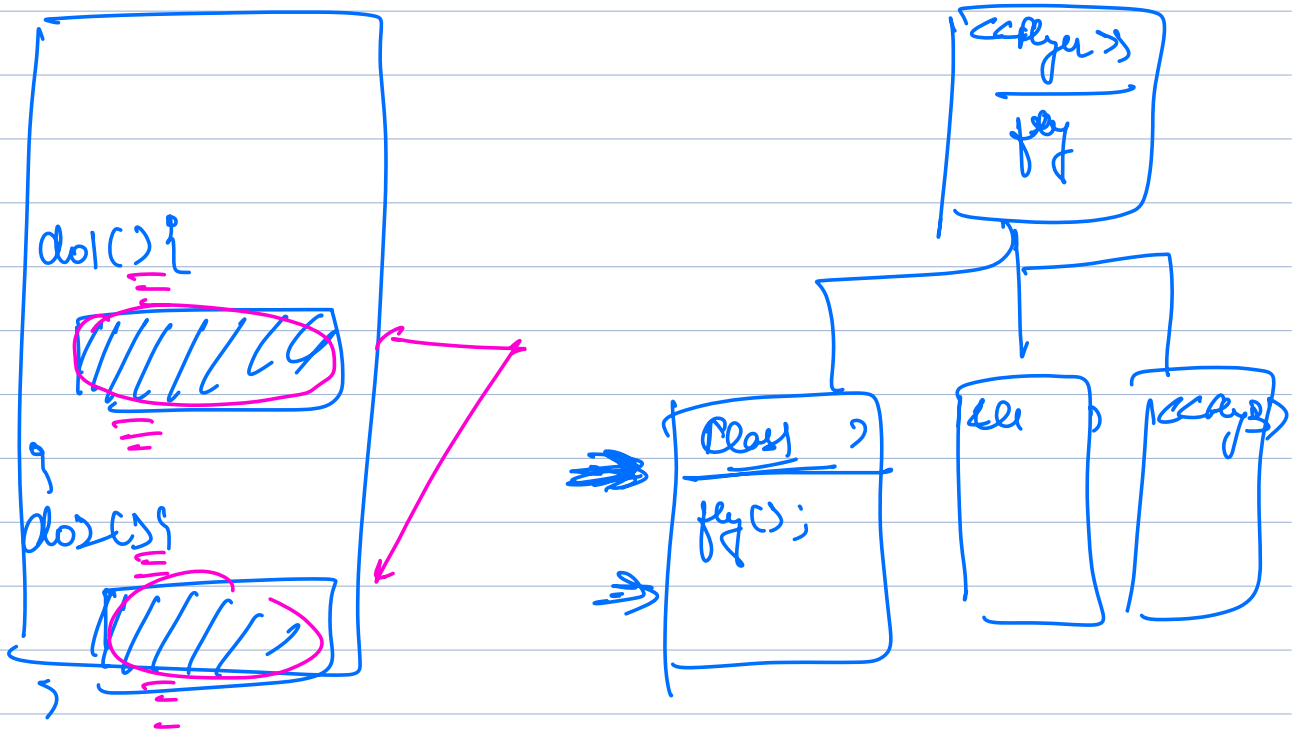
List <Player> fleis {

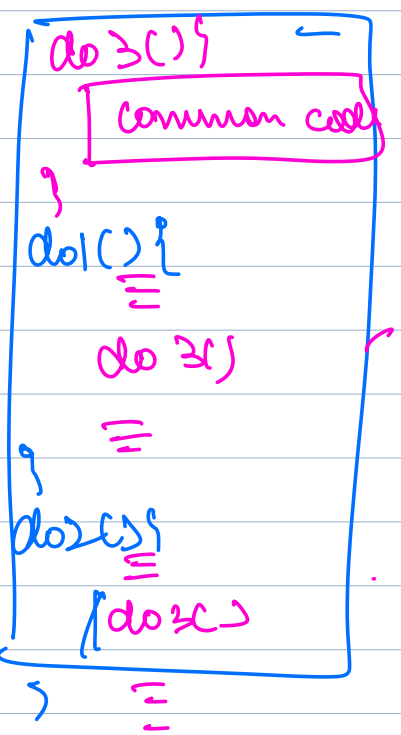
put objects of classes that implement Player

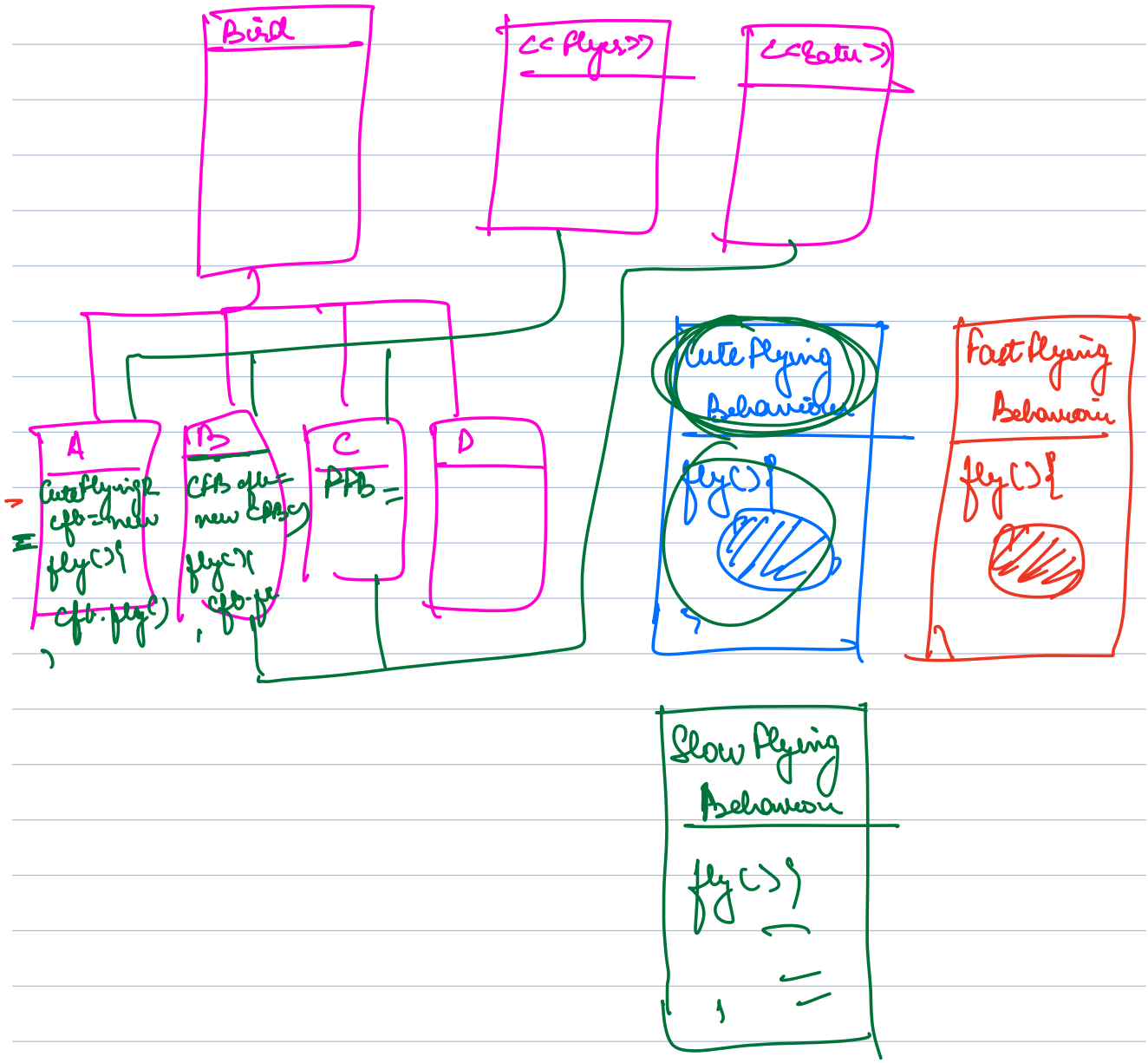
}

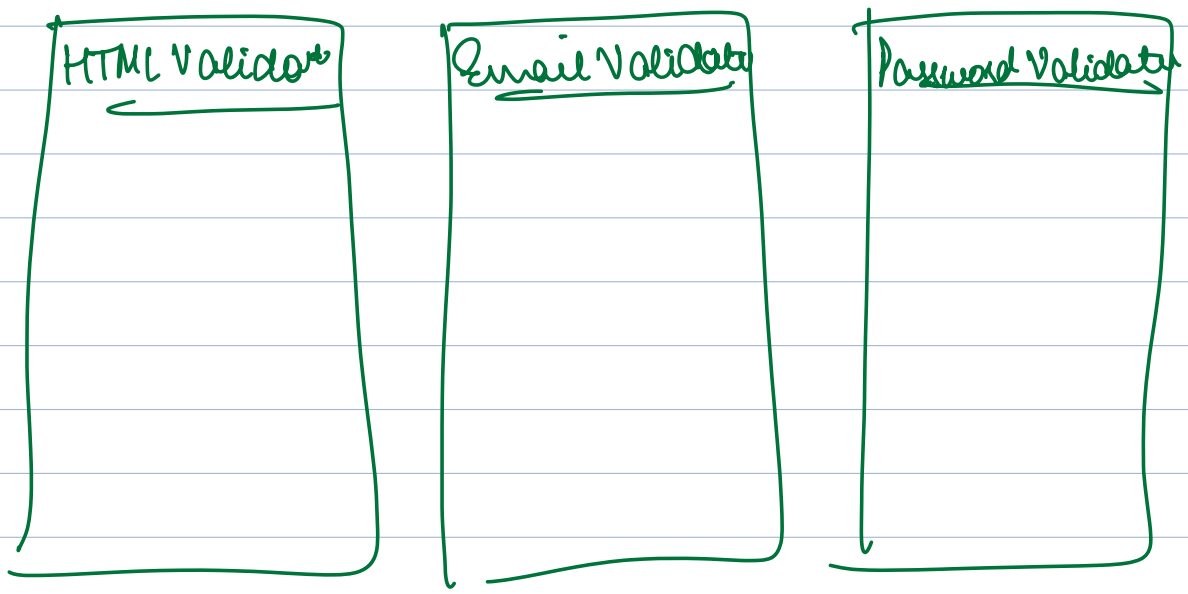
- Dep Inversion
- — Injection
- Code Tryout


Bird
<<flyer>>
fly()
Cute Flying Behav
Fast Flying Behav
Slow Flying Behav
Code Duplication
(DRY)


do1(){
do2(){
class
fly();

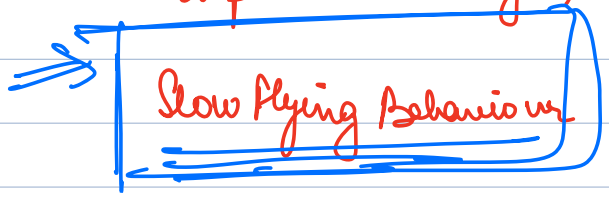

do3(){
common code
do1(){
do 3()
do2(){
do2()


Bird
<<Flyer>>
<<Eatu>>
A
B
C
D
Fast Flying Behaviour
fly()
Slow Flying Behaviour
fly()



⇒ when their logic may be needed from multiple places.

class Pigeon extends Bird implements Flyer, Eater {



sfb = new SlowFlyingBehaviour();

fly() {

sfb.fly();

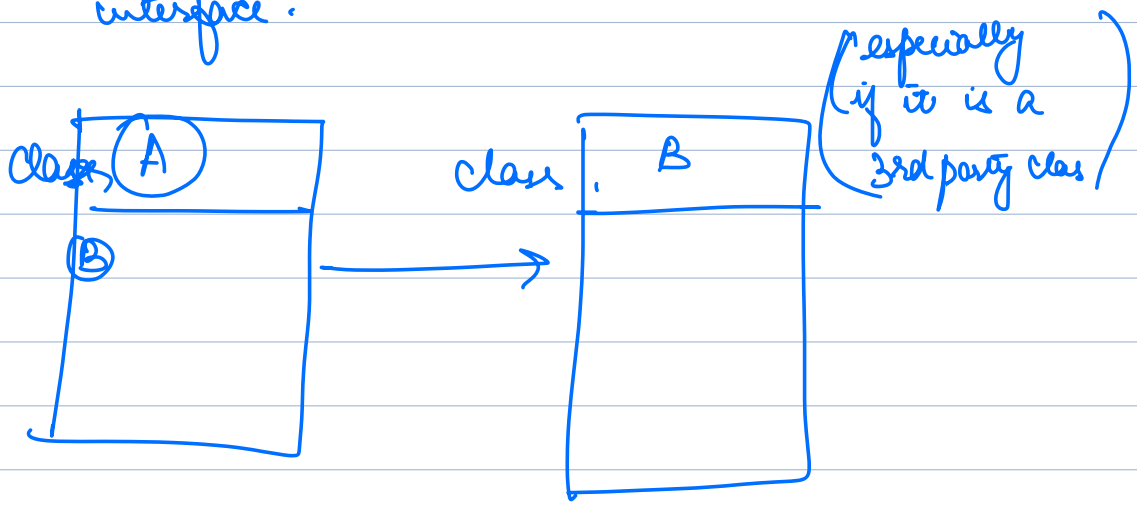
}

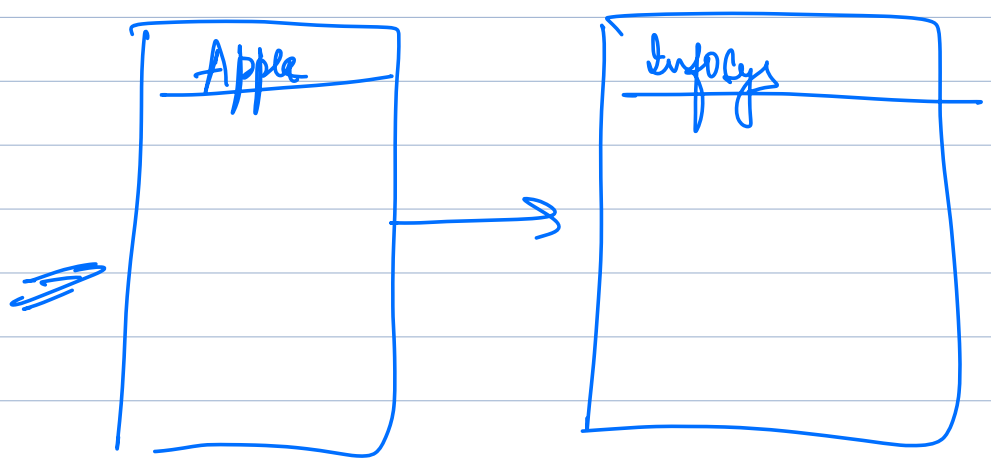
eat() { }

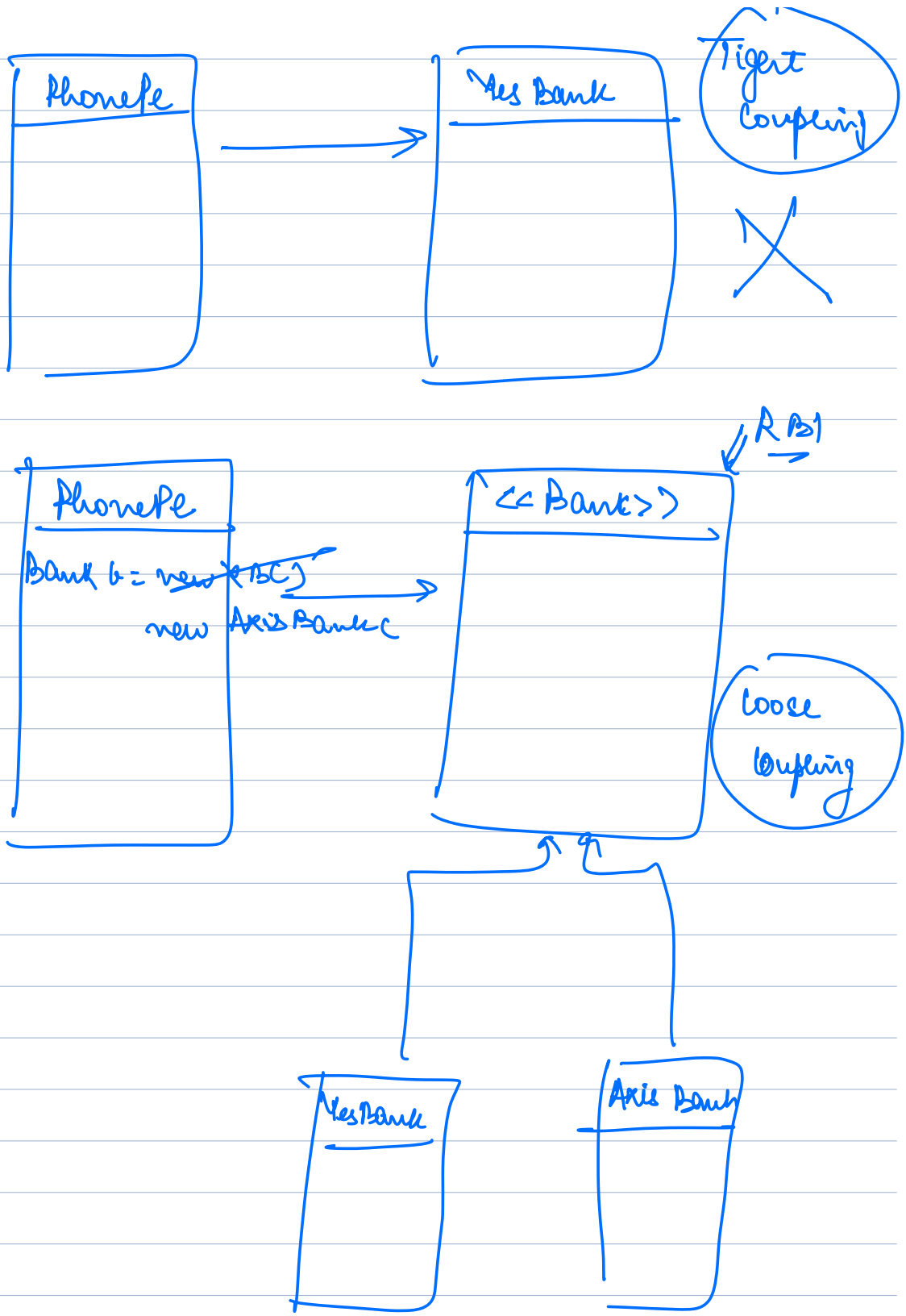
}

(D) ⇒ Dependency Inversion Principle

→ No 2 concrete classes should ideally depend on each other. They should depend on each other via an interface.






PhonePe
Yes Bank
Tight Coupling
PhonePe
Bank b = new ICICI();
new AxisBank();
RBI
<<Bank>>
Loose Coupling
YesBank
Axis Bank

List< > = new ~~ArrayList~~ LinkedList

Map< > = new ~~HashMap~~ TreeMap()

class Pigeon extends Bird
implements Flyer, Eater {

FlyingBehaviour fb = new SlowFlying Behaviour();

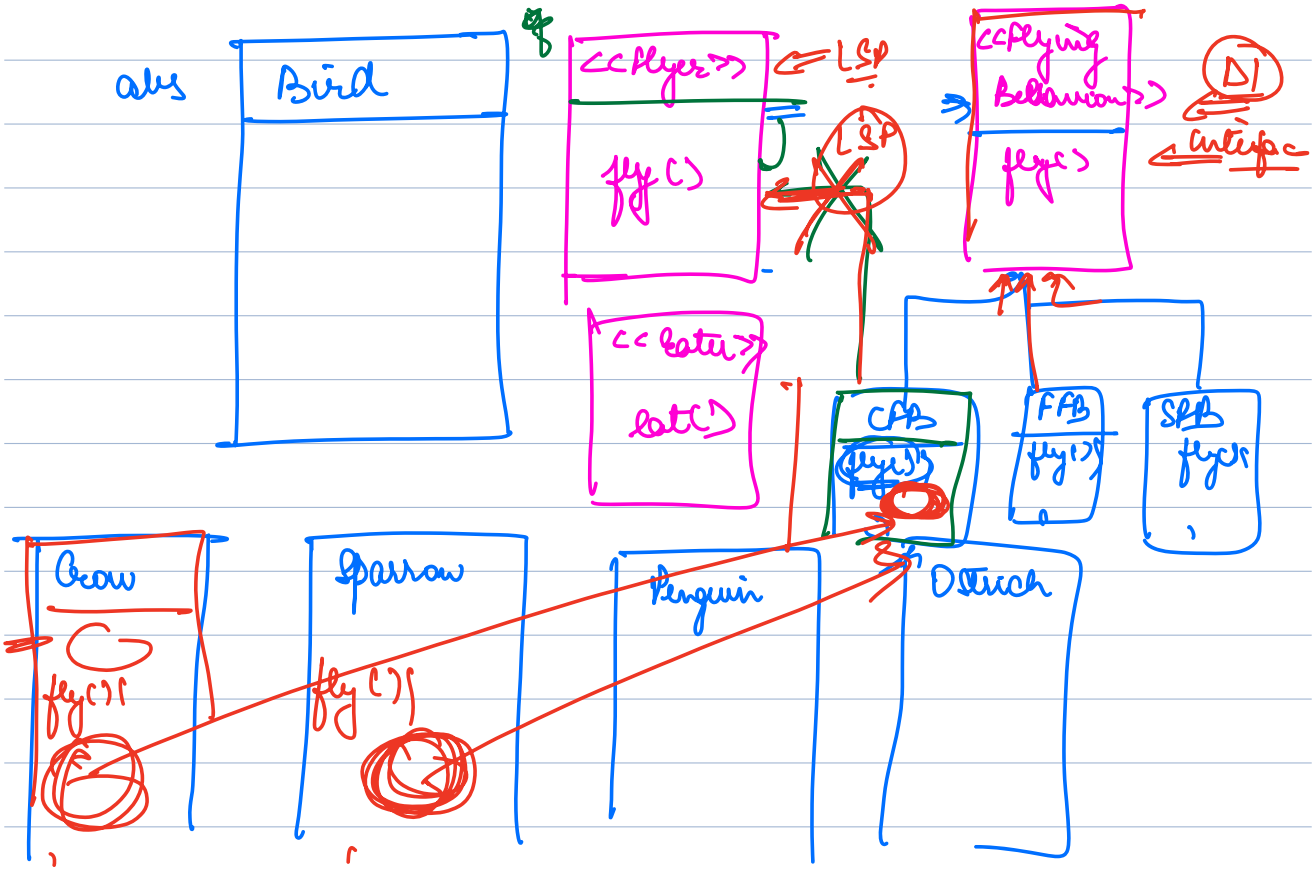
fly() {

fb.fly();

}

eat() { }

}



<<Flyer>> ⇒ a bird who can fly should implement this

List<Flyer> = { Birds }

new FB()

D ⇒ Dependency

FlyingBehaviour f = new SlowFlying

new ~~FlyingBeha~~

variable

Inheritance → IS-A

extending a class

implement an interface

HAS-A

having an attribute

```
Class Pigeon(Bird, FlyingBehaviour)

    def __init__(self, flyingBehaviour)
        self.fb = flyingBeha

    def fly()
        self.fb.fly()
```

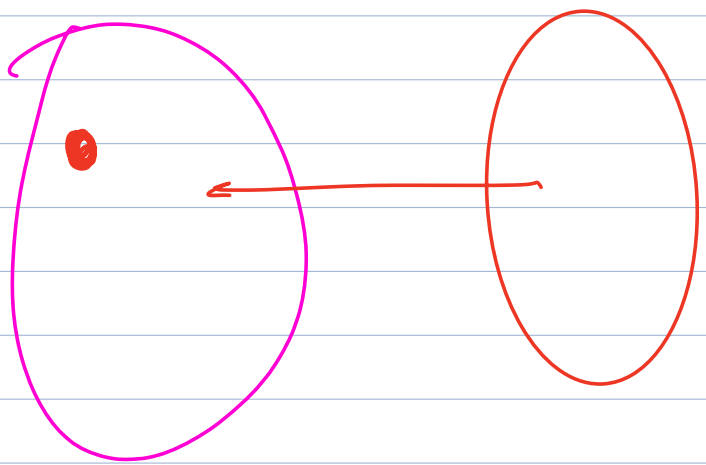
Dependency Injection

↳ Someone you are dependent on

↳ HAS-A (attribute)

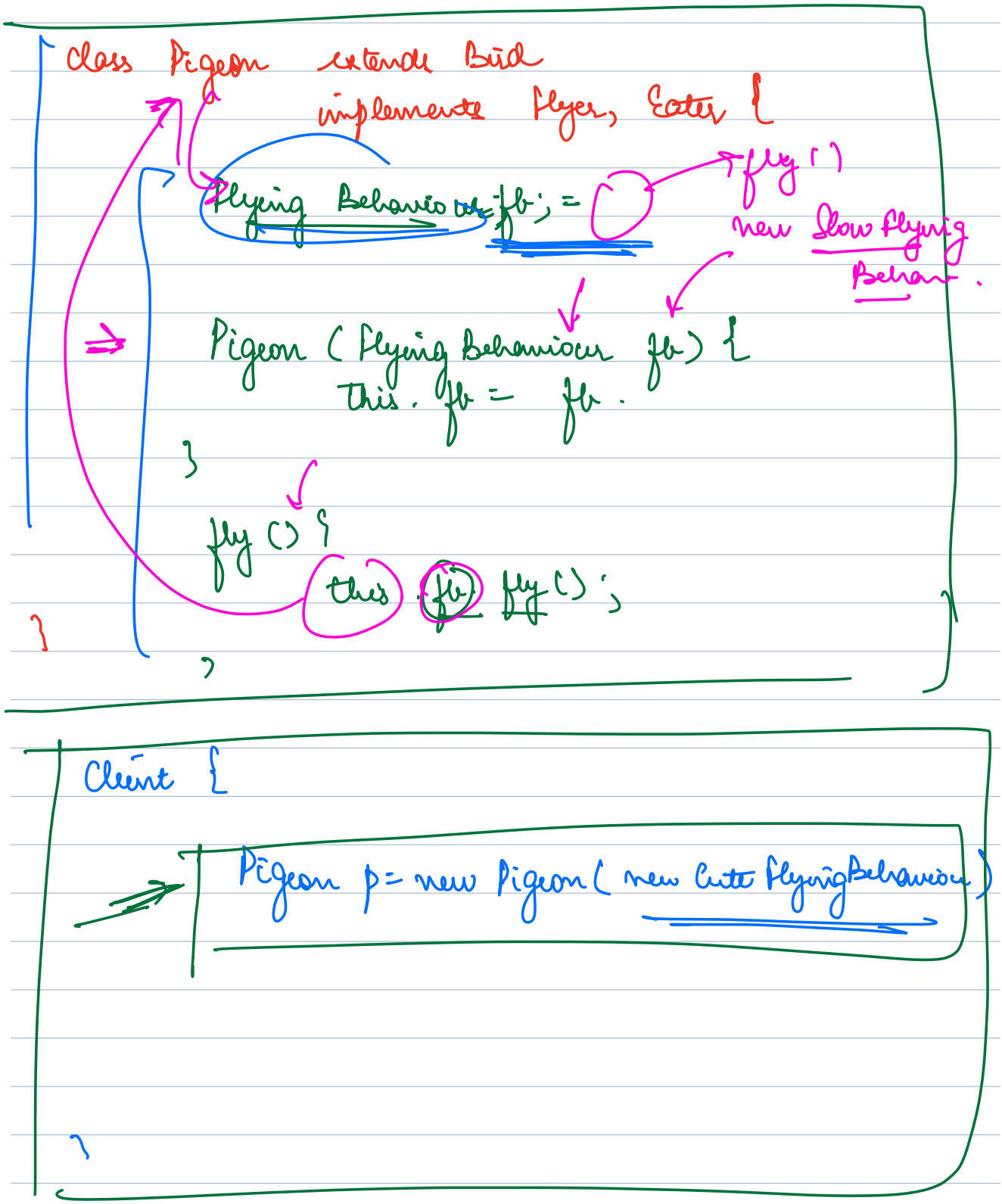
↳ FB is a dependency of pigeon class

In earlier example dependency was being created by the class itself



Dependency Injection: When the dependencies of a class are given to it by its client instead of class creating itself

⇒ often via constructor


class Pigeon extends Bird
implements Flyer, Eater {
FlyingBehaviour fb; =
fly()
new SlowFlyingBehaviour.
Pigeon (FlyingBehaviour fb) {
this.fb = fb.
}
fly() {
this.fb.fly();
}
Client {
Pigeon p = new Pigeon(new CuteFlyingBehaviour)
}

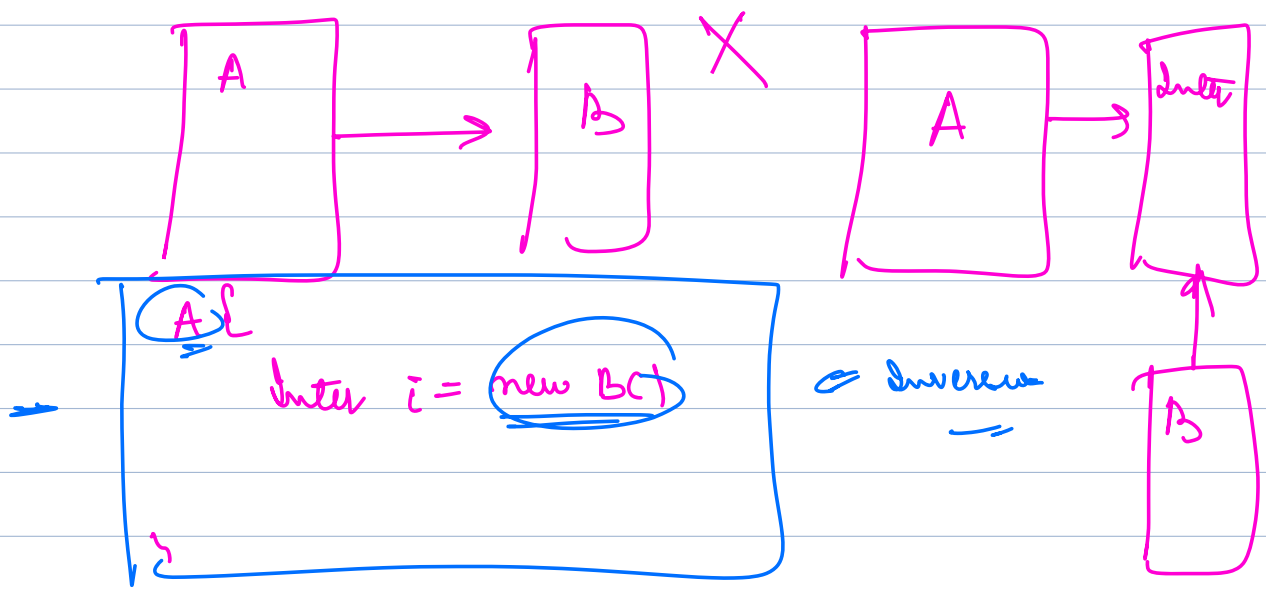
Bird → VO

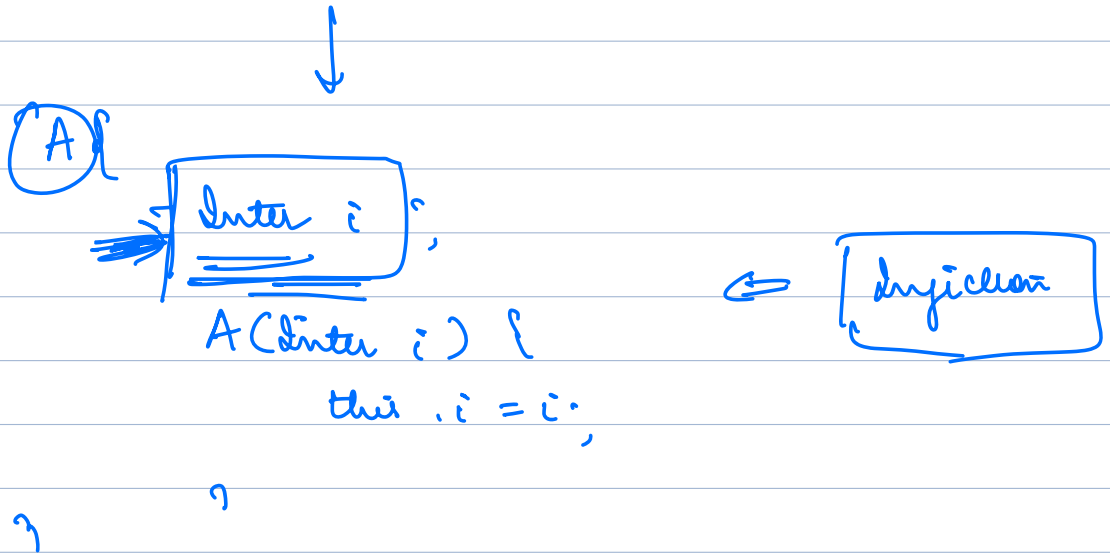
VS → TW

Write code of VS of Bird

OOP: Remaining Concepts →

All Doubts





H/W

1. Implement Bird V5
2. SRP H/W
3. 3 more q^n by tomorrow morning