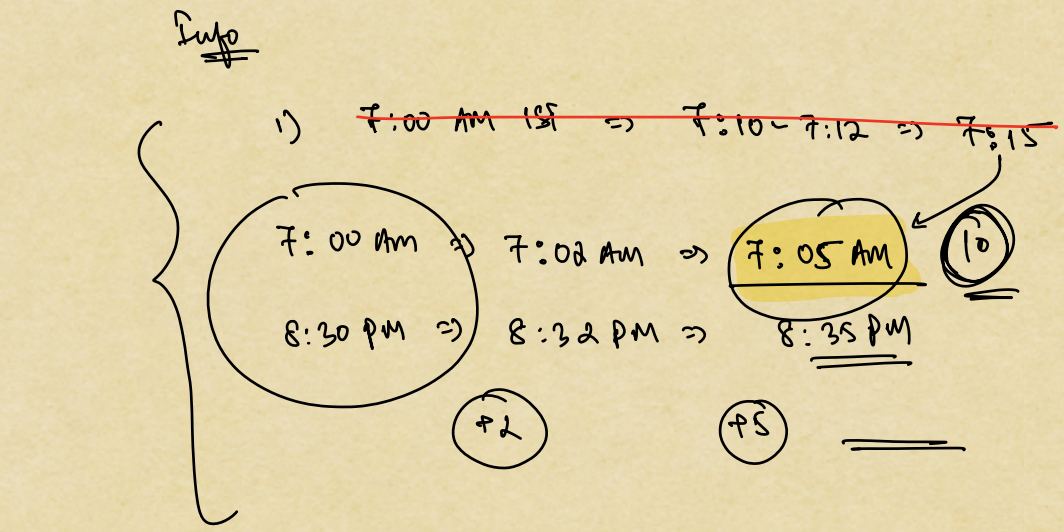
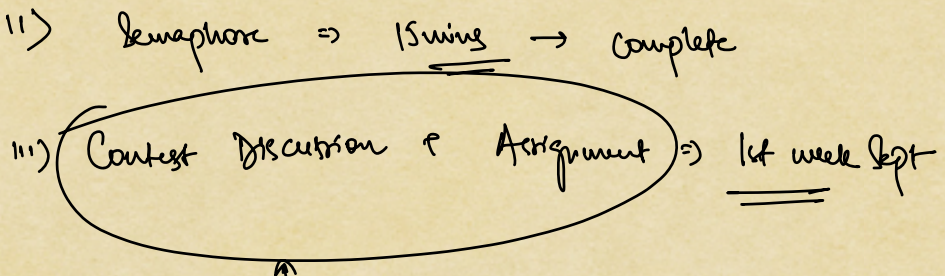
Info

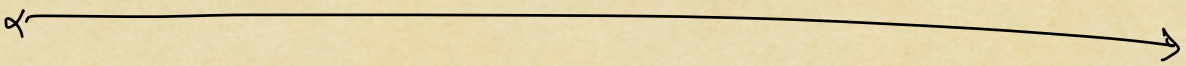




LLD 1

>60 pass

<= 60 not pass

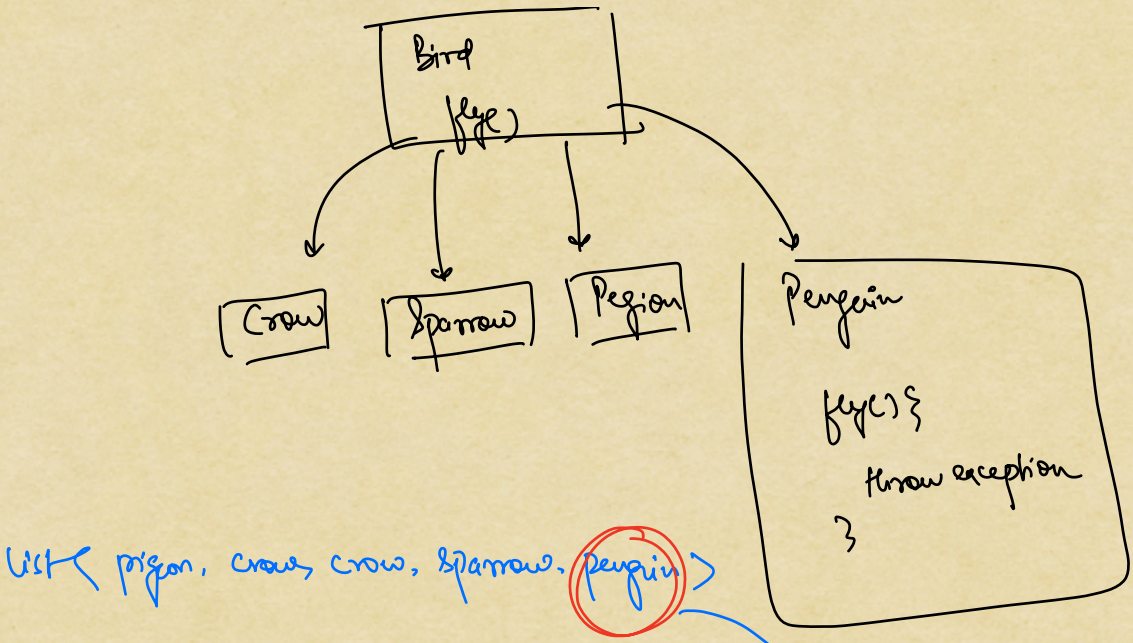


LSP => Liskov Substitution Principle

without any extra effort

Defn => Object of any child class should be as-is substitutable in a variable of the parent type without requiring any extra effort or code changes.

↑ upcasting



List< pigeon, crow, crow, sparrow, Penguin >

```
void makeAllBirdsFly(List<Bird> birds) {
    try{
        for(Bird b : birds){
            b.fly();
        }
    }catch(Exception e){
    }
}
```

→ only req. because of Penguin class

↓

any extra code / special treatment to accomodate any child class obj in parent ref. variable is violation of LSP

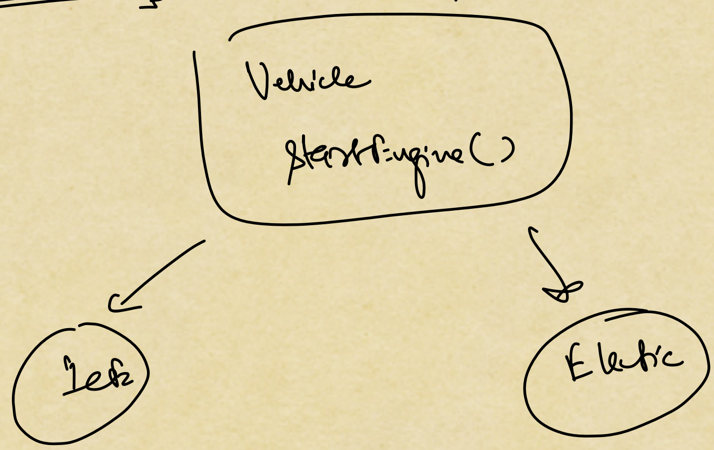
Defn: If a parent class has N child classes then object of any of N child classes should be easily upcasted to parent ref. variable without any extra effort.

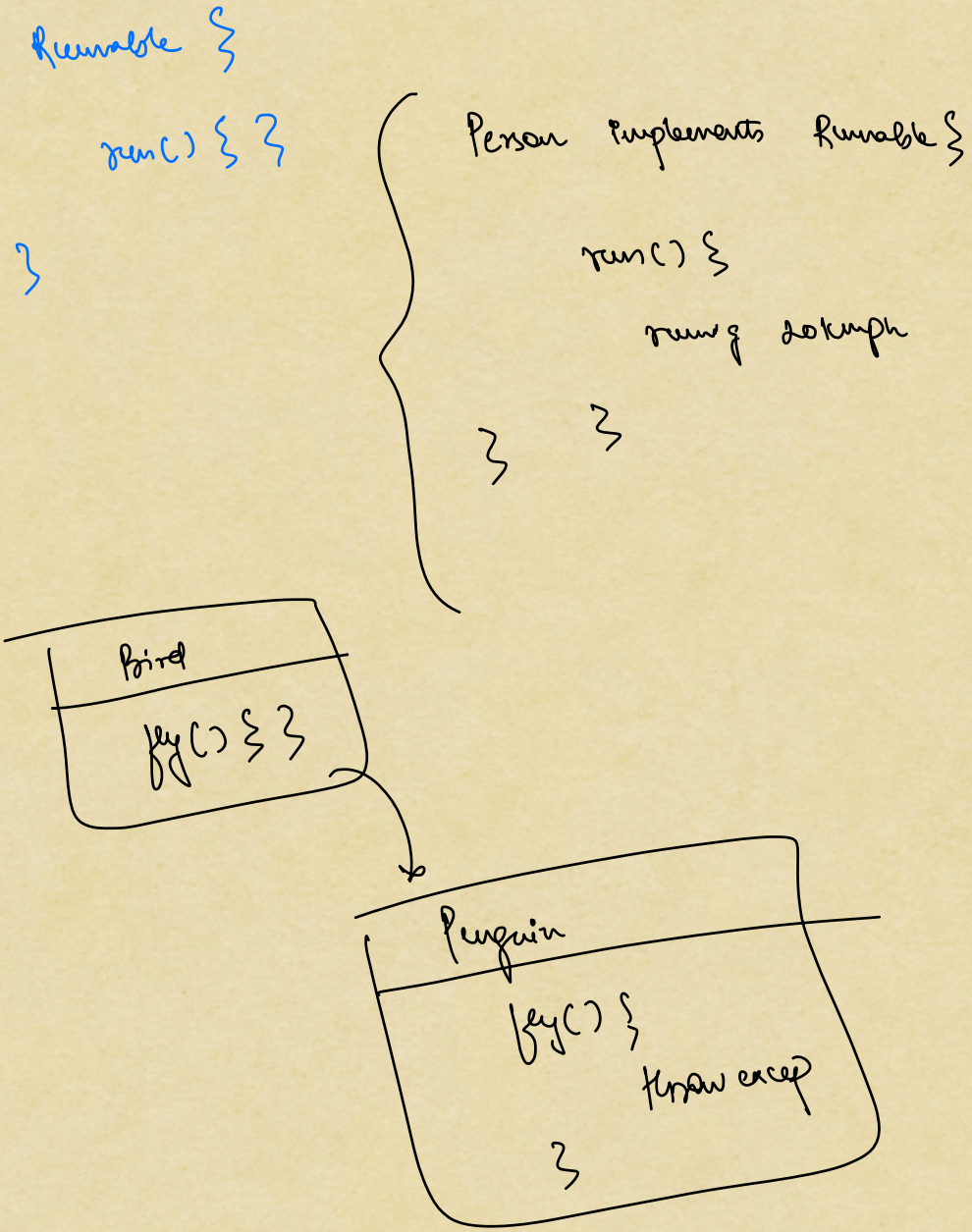
```
public void method(Parent p) {   ← child obj

}
```

Possible scenario for LSP violation:

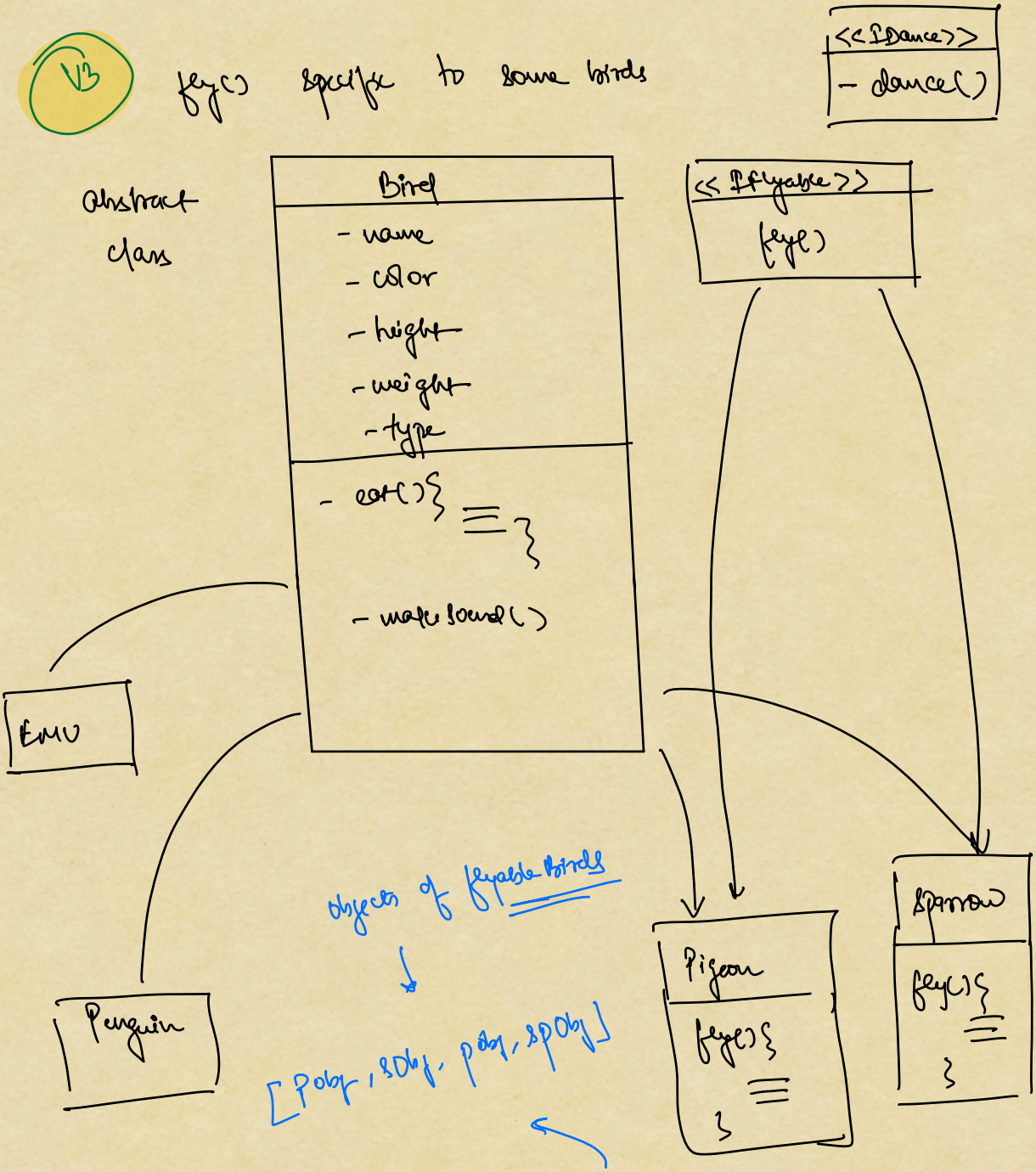
- **Overriding** :- and change the objective/goal of method.




Runnable {
run() { }
}
Person implements Runnable {
run() {
run @ 20kmph
}
}
Bird
fly() { }
Penguin
fly() {
throw excep
}

V3

fly() specific to some birds




```
void makeAllBirdFly(List<Iflyable> birds) {
    for(Iflyable b : birds) {
        b.fly();
    }
}
```

Class Pigeon extends Bird implements IFlyable {

}

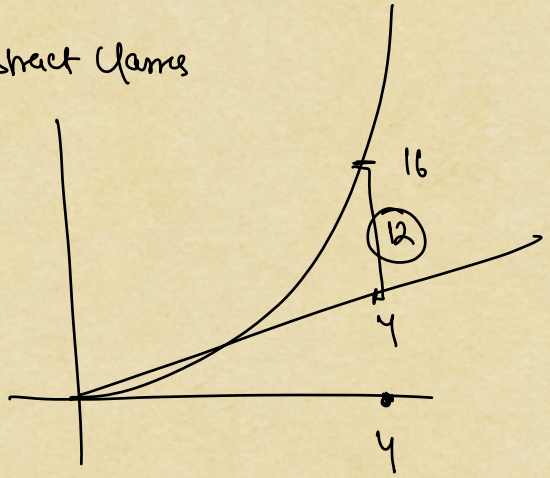
⇒ for any new bird

Class BirdName extends Bird implements IFlyable, IName, — {

}

N features ⇒ 2^N abstract classes

N features ⇒ N interfaces



* every interface is related to 1 action.

ISP :- Interface Segregation Principle

→ Interfaces should be as light as possible

→ As less methods as possible

→ Ideally there should be only 1 method in interface

↓

functional interface

↓

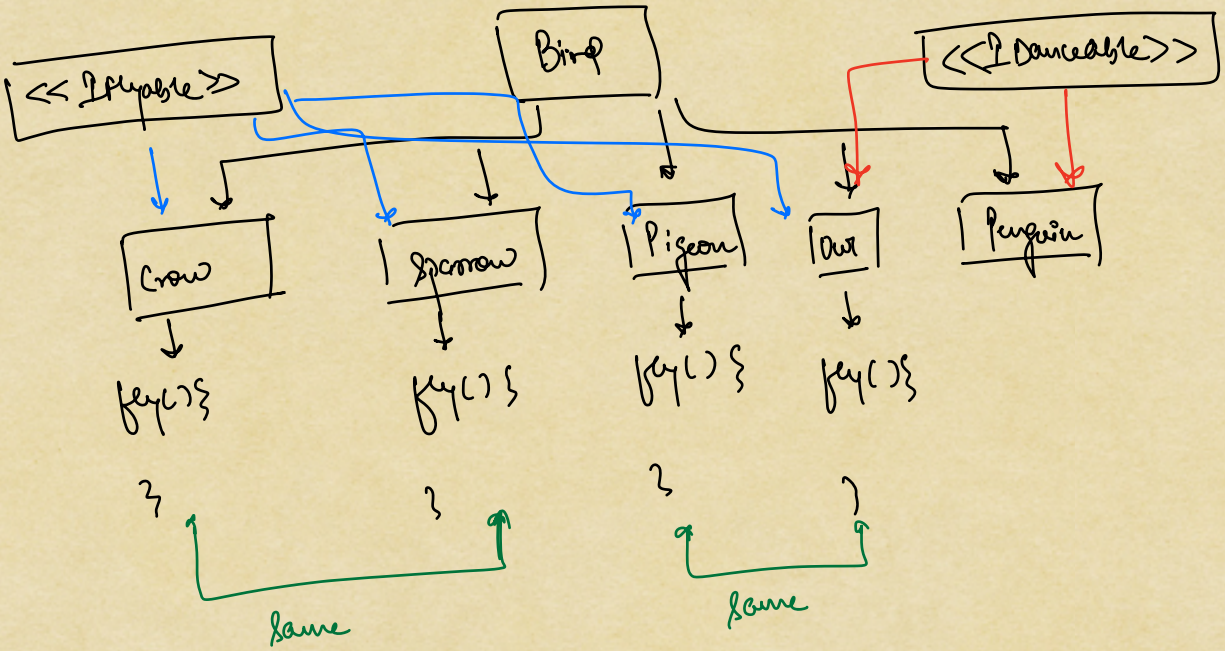
it contains only 1 abstract method

helps to implement ⇐ lambda expressions

⇒ Combine methods in an interface which logically belong together

⇒ { SRP for Interfaces } ≡ { ISP }

⇒ DIP ⇒ Dependency Inversion Principle



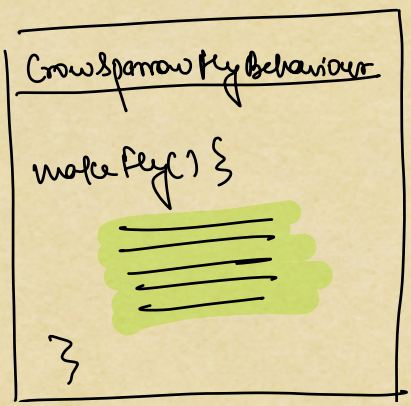
- When implementing a lot of birds, there is a high possibility of duplicate fly behavior hence Code duplication

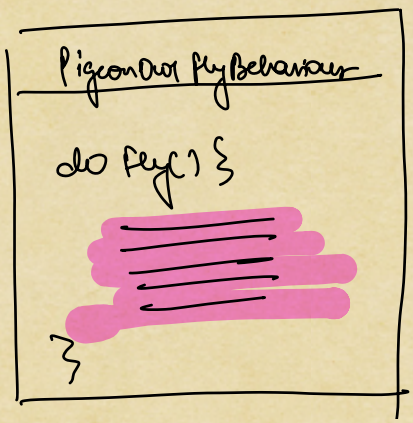
Assume

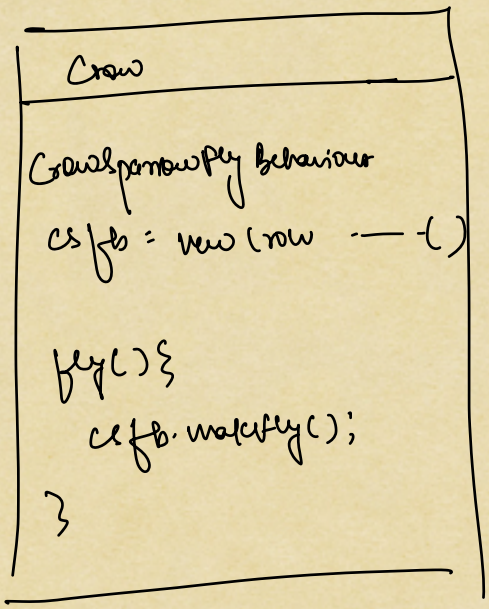
Crow & Sparrow have same fly behaviour

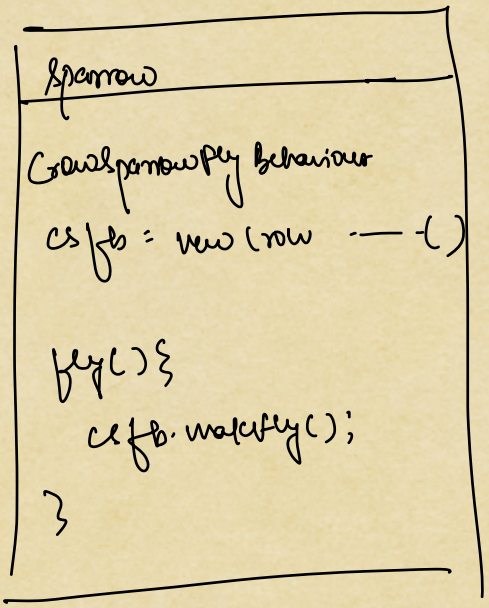
Pigeon & Owl " " " "

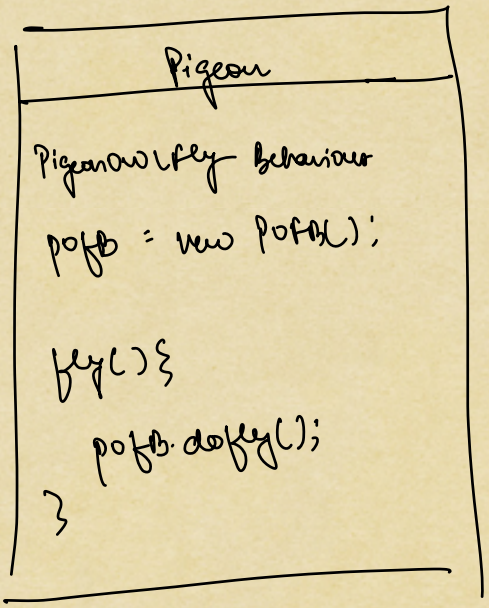
Sol^n

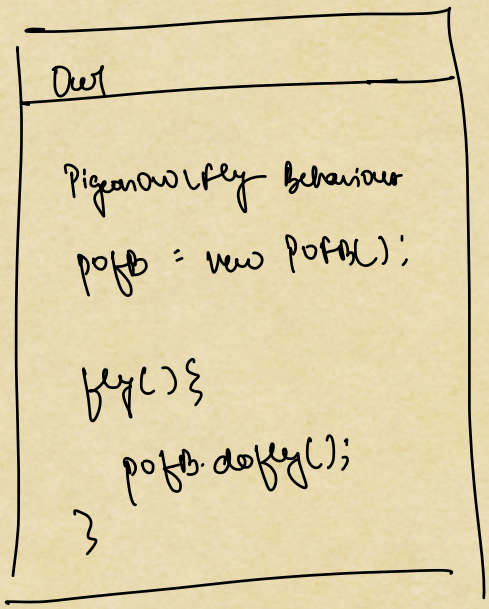


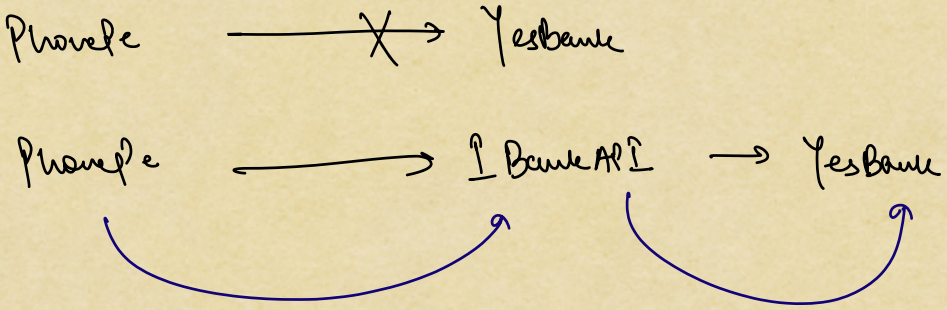



Crow
CrowSparrowPigeonBehaviour
csfb = new CrowSparrowPigeonBehaviour();
fly(){
csfb.makeFly();
}

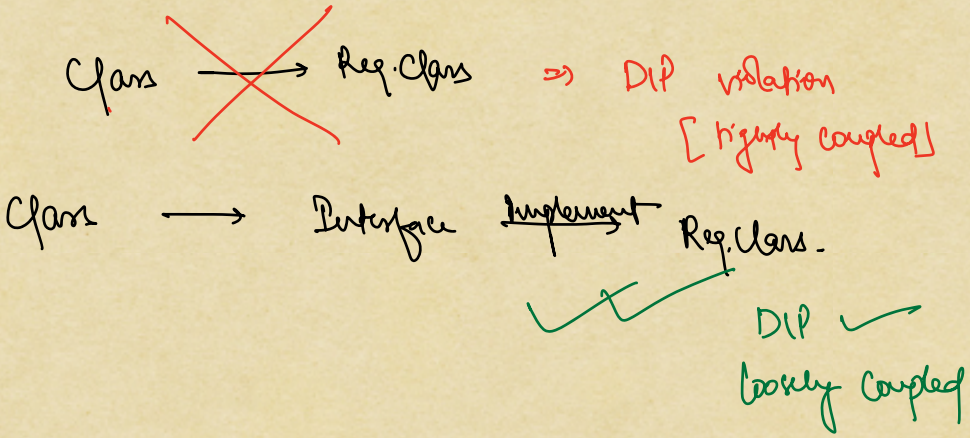

Sparrow
CrowSparrowPigeonBehaviour
csfb = new CrowSparrowPigeonBehaviour();
fly(){
csfb.makeFly();
}

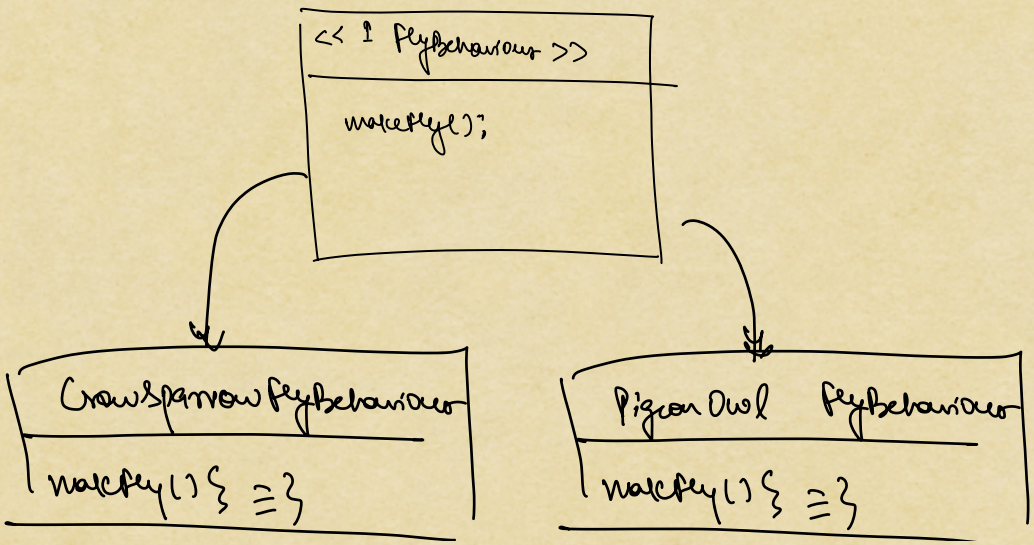

Pigeon
PigeonNoFlyBehaviour
pofb = new PoFB();
fly(){
pofb.doFly();
}

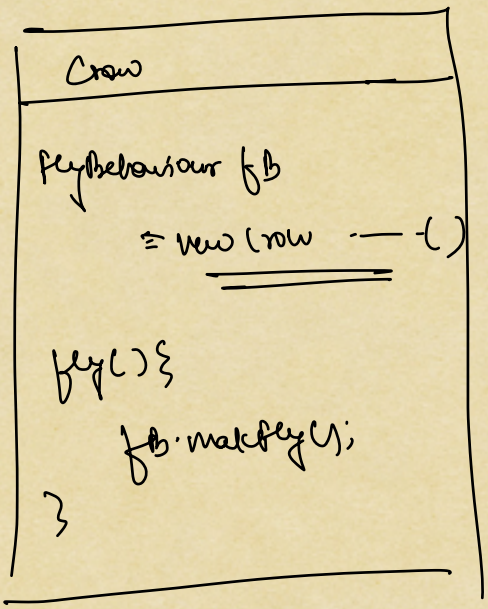

Owl
PigeonNoFlyBehaviour
pofb = new PoFB();
fly(){
pofb.doFly();
}



* DIP => 2 concrete classes should never depend on each directly, they should depend on each indirectly [Inversion to dependency]







only this dependency change will be needed

I1 {
run();
}

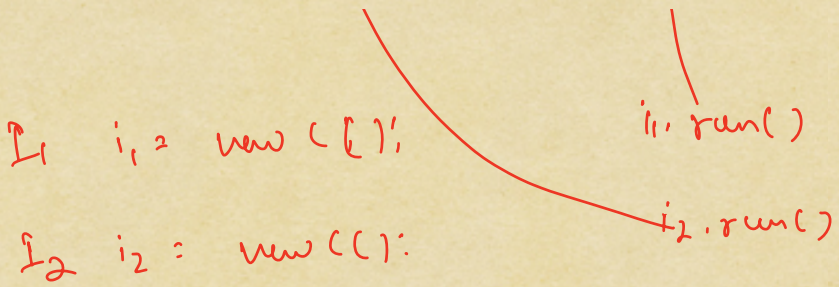
I2 {
run();
}

class C implement I1, I2 {

run() {

10lumps

?




```
Class Crow extends Bird implements IFlyable {
    FlyBehaviour fb;

    public Crow( FlyBehaviour fb) {
        this.fb = fb;
    }

    void fly() {
        fb.makefly();
    }
}

interface FlyBehaviour {
    void makefly();
}
```

```
class FastLowFlyBehaviour implement FlyBehaviour {

    void makeFly() {
        ========
        ========
        ========
        ========
    }
}

class Main {

    psvm() {

        Crow c = new Crow(new FastLowFlyBehav());

        c.fly();
    }
}
```

HW Write the final version Bird in code