Agenda

1) Recap → SRP
2) L → Liskov's Substitution Principle
   I → Interface segregation principle
   D → Dependency inversion

Nikhil Jain — 10+ years exp → Backend Java

↳ Principal Engg at Atlassian
↳ Confluence

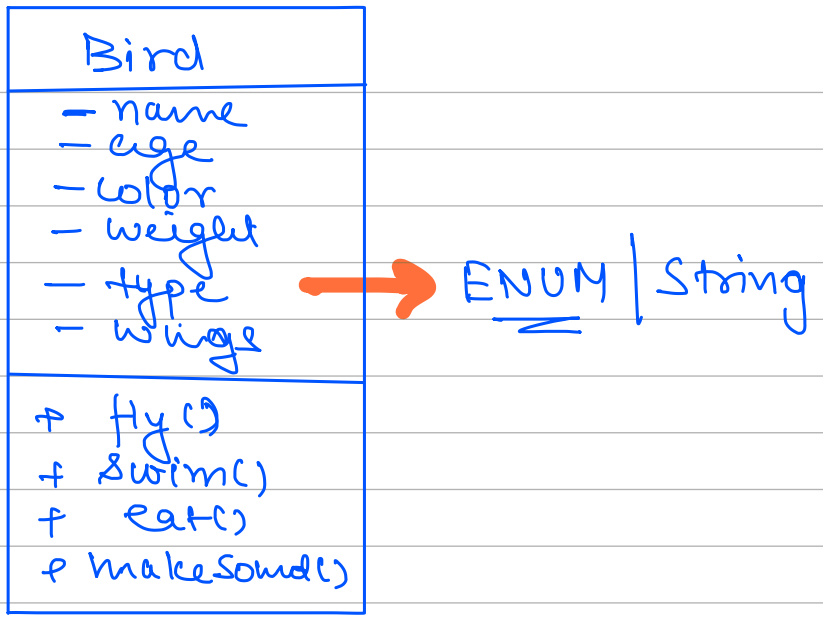
↳ Sr SDE at Amazon
↳ Staff/Principal Engg at Walmart
↳ Sr SDE at Microsoft Bing C#
↳ SDE at Amazon Image
↳ SDE at finsatva/Misys OLD

MNIT Jaipur 2014 CSE

Design a Bird (Pen)

Assume that we have a software system that stores different type of birds, with their corresponding Attrs & Behaviors

(V1)



Bird b1 = new Bird()
b1.name = —
b1.age = —
b1.weight = —
b1.type = "Piegon"

Bird b2 = new Bird()
b2.name = —
b2.age = —
b2.weight = —
b2.type = "Sparrow"

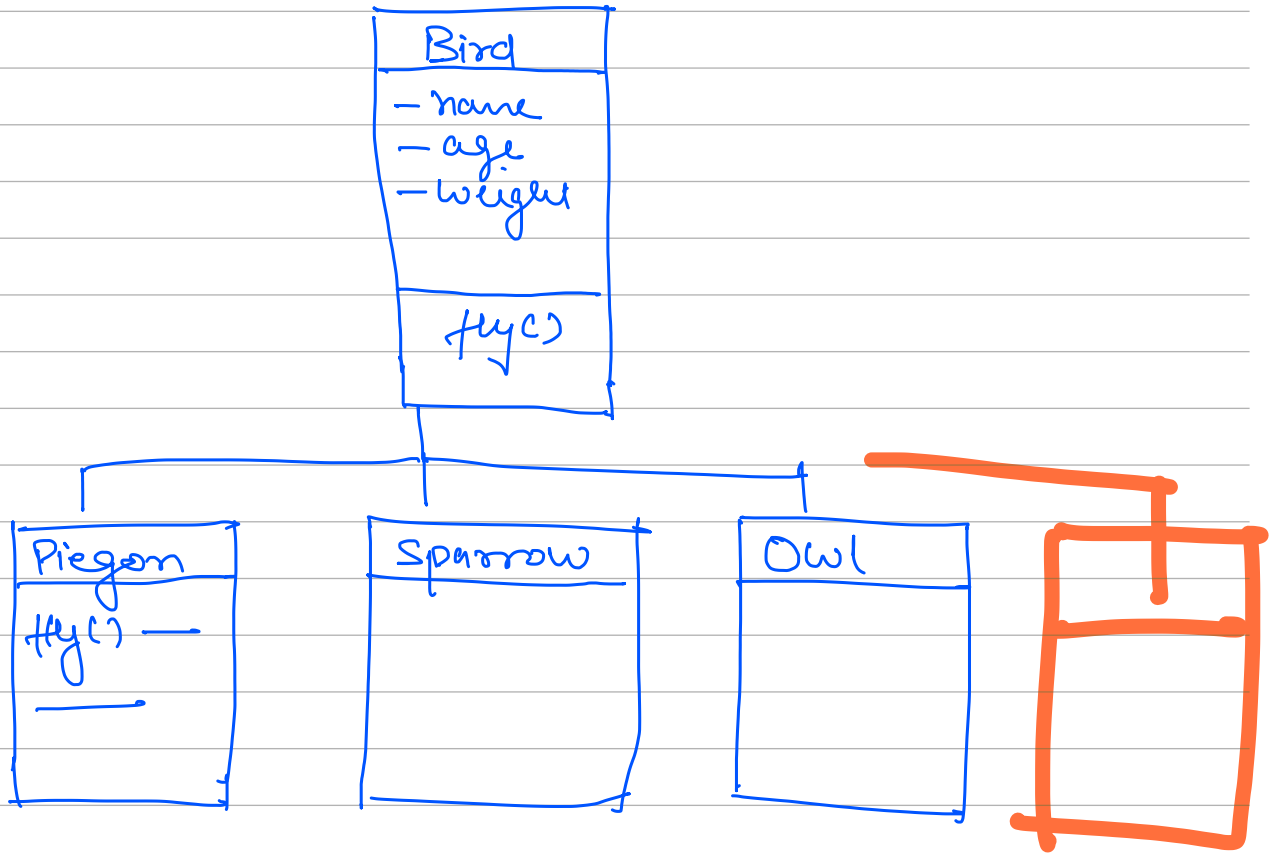
```
fly() {
    if(type == "Piegon")
    {
        fly altitude = 100ft
    }
    else if(type == "Sparrow") {
        fly altitude = 20ft
    }
    else if(type == "eagle") {
        fly altitude = 1000ft
    }
    else {
        // do nothing
    }
}
```

SRP voilates
OCP voilates

→ Bird Abstraction → fly()

→ Piegon Concrete fly

Abstract Class



V2

eg

Print(T ⟶)

Long

↳ Type is Not supported

Int | String

abstract Print( )

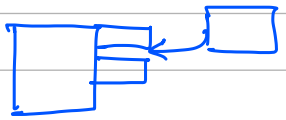
Print(Int)

Print(String)

Print(Long)

Type Check → Int / String / Long

Confluence — Plugin



Penguin

↳ fly() {

}

List<Bird> ——

for (

try { Bird.fly() }

catch(Exception)

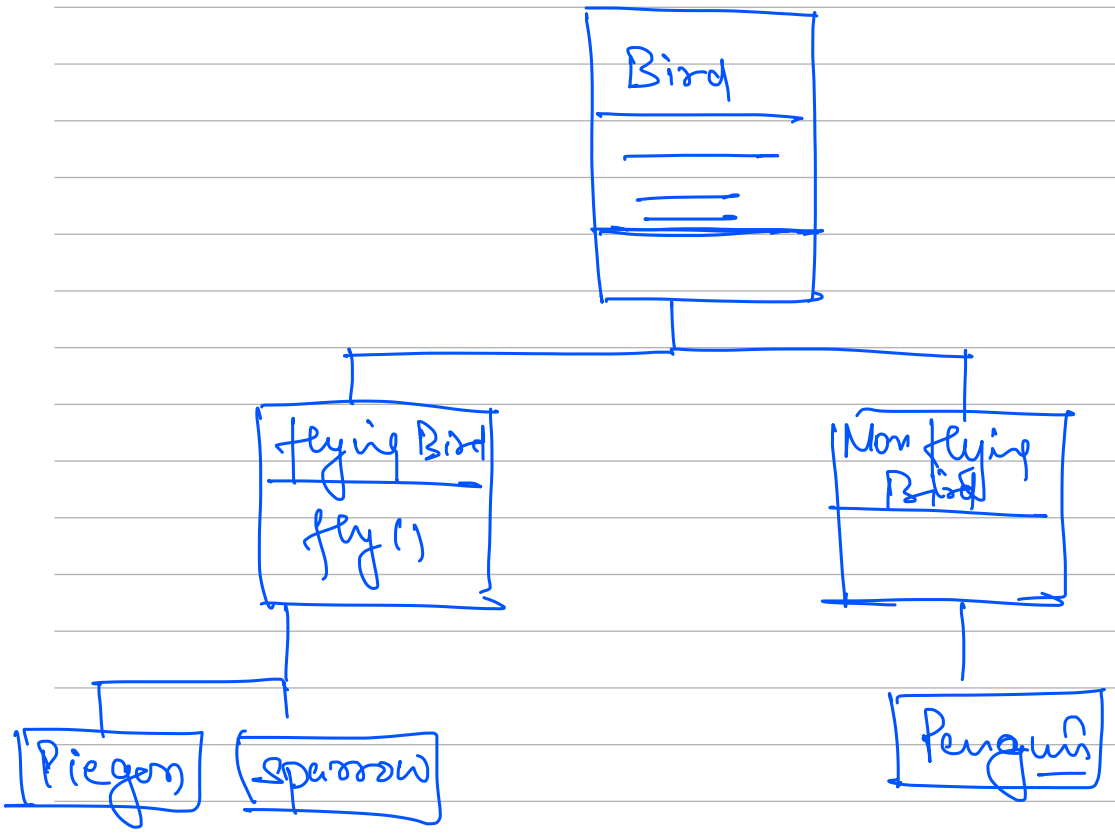
{ continue }

} } Runtime

If an entity does not support a behaviour then that behaviour in the entity should not exist

2 type of Birds

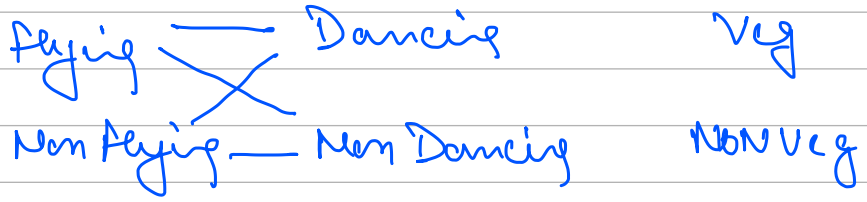
→ flying Birds
→ Non flying Birds




```
Bird b = new Penguin()
b.fly();
```

List<flyingBirds> ——————
List<NonflyingBirds>

Some Birds can dance & some can't



flying Dancing Bird Non flying Dancing

2^N Classes N is Number of features

Class explosion

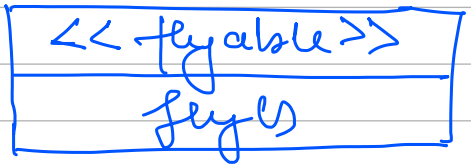
Problem Statement

Some birds demonstrated some behaviors and other birds demonstrated other behaviors.

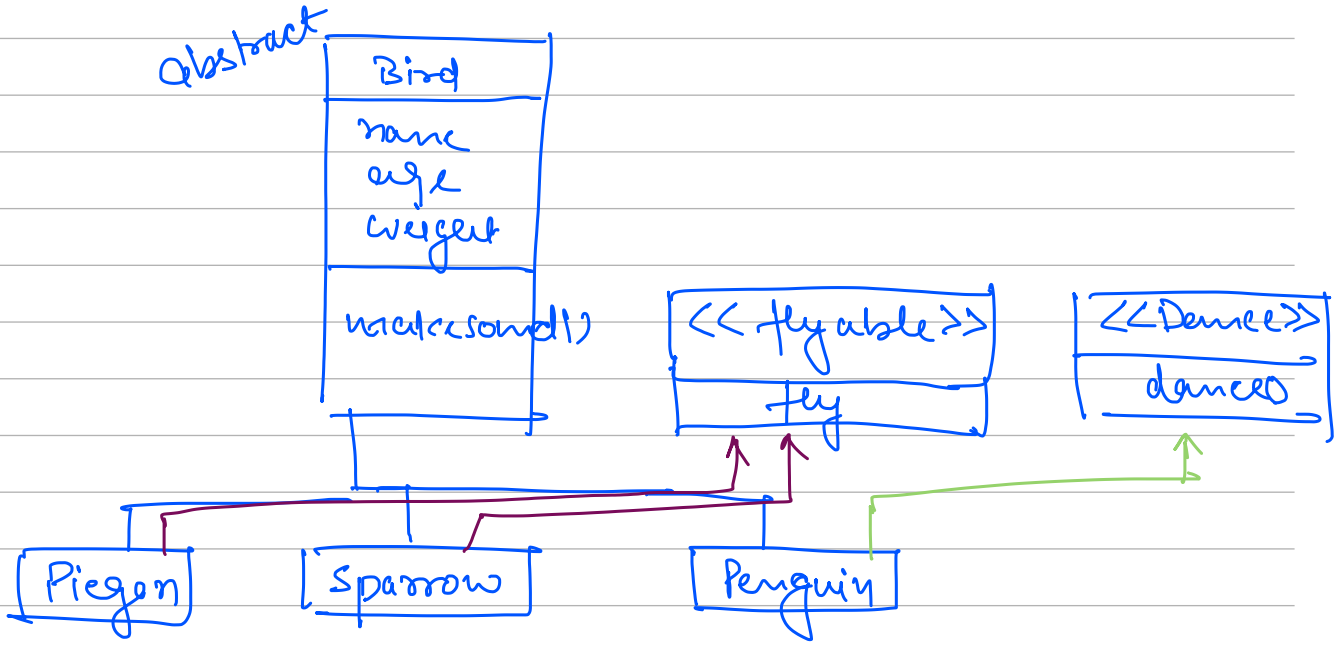
Only birds that support a behavior should have that Method

We should be able create list of birds based on behaviors

Interfaces: Used to represent behaviors



V3



Class sparrow extends Bird implement flyable {

}

List<Birds>
List<flyable>
List<Dancee>

Penguin → Non-flying + Dancing Bird] Abstract

Pegon → flying + Dancing] Abstract

flying
Non flying

Dancing
Non Dancing

FlyingDancingBird NonFlyingDancingBird

fly()
Dance()

Dance

FlyingNonDancingBird NonFlyingNonDancingBird

fly()

2 Behaviors $= 2^{\wedge 2} = 4$

n Behaviors $= 2^{\wedge n}$ = Class Explosion

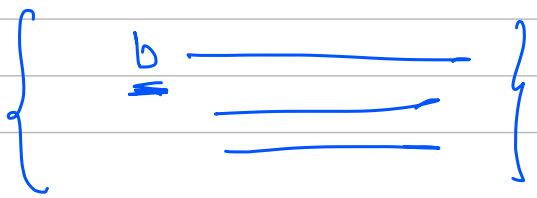
<< flyable >> (walk) << Dancer >>

✓	✓	✓	fly + Dancing + walk
✗	✓	✓	Dancing + walk
✓	✗	✗	fly
✗	✗	✗	None

Liskov's Principle

Object of any child class should be as-is substitutable in variable of parent class.

Bird b = new Pigeon()
new Penguin()
new Crow()
new Sparrow()



Phonepe

YES Bank

Phonepe

int getBal (int Acc)

ICICI Bank

long checkBal (String AccNo)

24hours

Parent BankApi → int getBal(String Acc)

Phonepe

YESBank

ICICIBank

BankApi

BA = new YESBank()

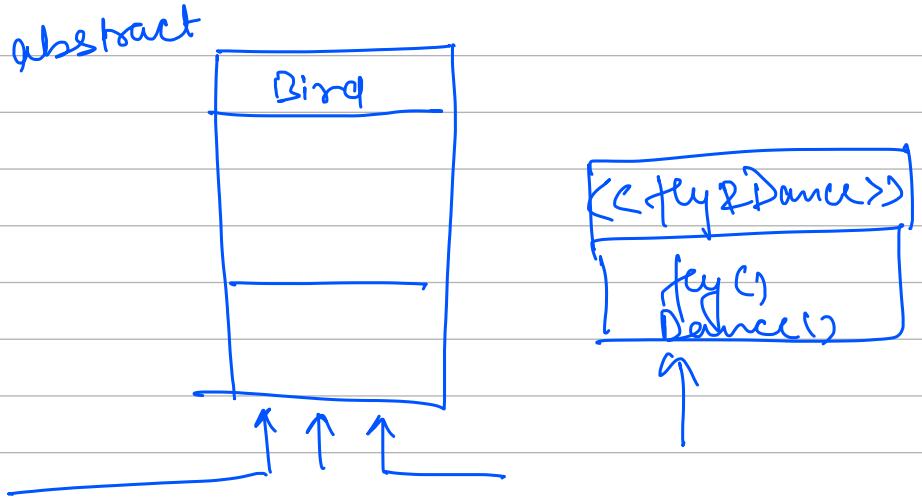
ICICIBank()

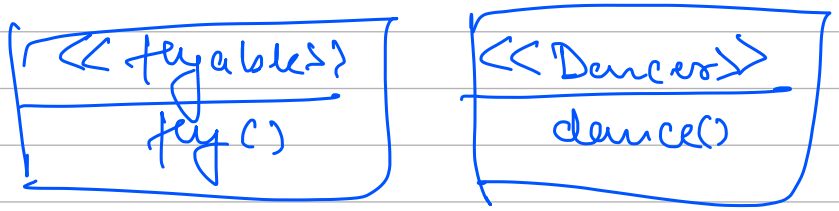
flyable f = new Parrot()
new Sparrow()
new Penguin() X

PM → Req 1 ⟶ 4 weeks Req 2 ⟶

Req 1 : Bird which can fly it can dance also and vice versa

fly()
Dance() } together





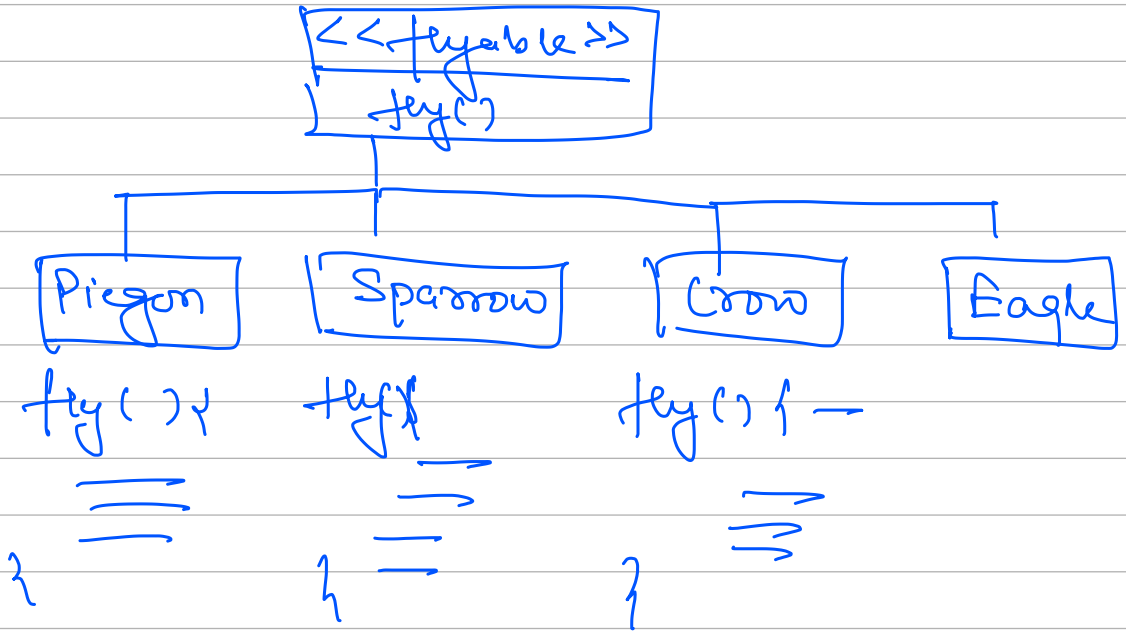
Interface Segregation Principle

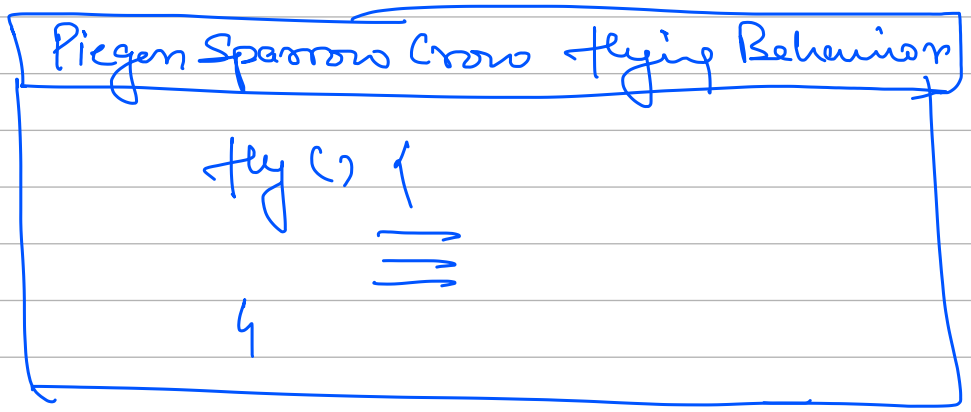
Interface should be as light as possible

Interface, Ideally should contain only 1 Method

→ Functional Interfaces

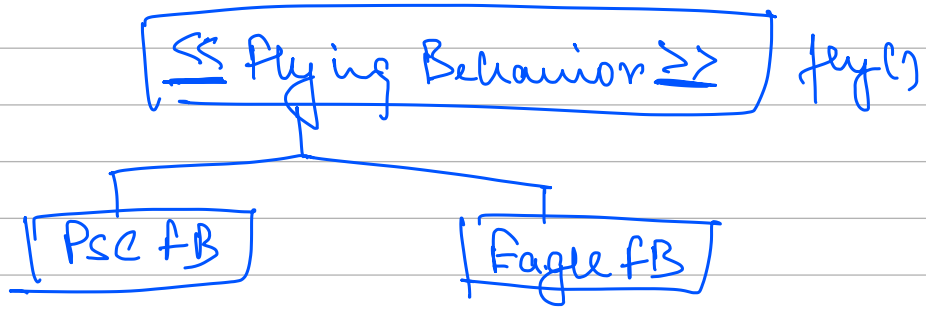
Dependency Inversion






```
Pigeon {
    PSCFB PB = new PSCFB();
                   EagleFB();

    fly() {
        PB.fly();
    }
}
```

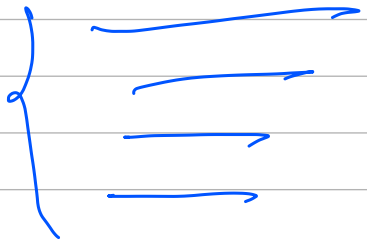
Dependency Inversion says that NO Two Concrete Classes should directly depend on each other they should depend on each other via an interface

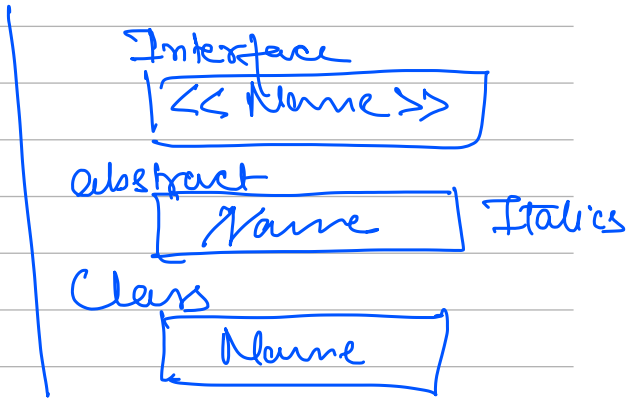


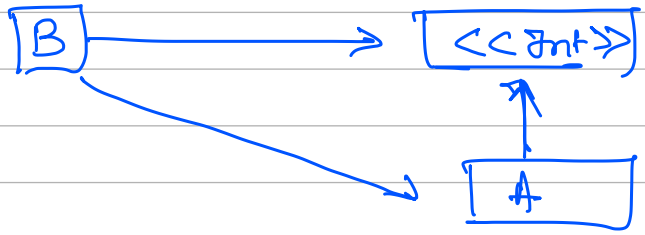
Pigeon {

FB fb = new PscFB()

EagleFB()







Dependency Injection

NOT SOLID

```
Piegon {
    FB fb = new PSCFB();
            EagleFB();
}
```

Robot
↳ Battery
↓
Battery.Power()

Everea
Duracell

No Need to create object of dependency within the Class. Let Client/Consumer Create dependency and provide to Class

```
Piegon {
    FB fb;
    Piegon ( FB fb) {
        this.fb = fb;
    }
}
```

new PSCFB()
new EagleFB()

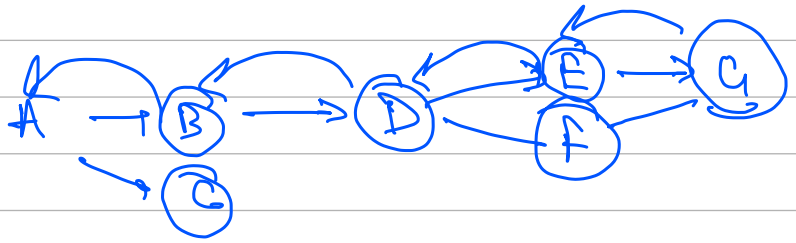
Piegon P1 = new Piegon( )

Robot R = new Robot (Duracell)
(Eveready)

Testing → Easier

```
A {
    B b
}

B {
    C c
}
```

```
C c = new C();      (3)
B b = new B(c);     (2)
A a = new A(a);     (1)
```



Dependency → Spring
Django

S → SRP
O → OC
L → Liskov's
I → IS
D → DI

Repo → DB

```
A {
    B b
}
```

Client ← Main()

```
B b = new B()
A a = new A(b);
```

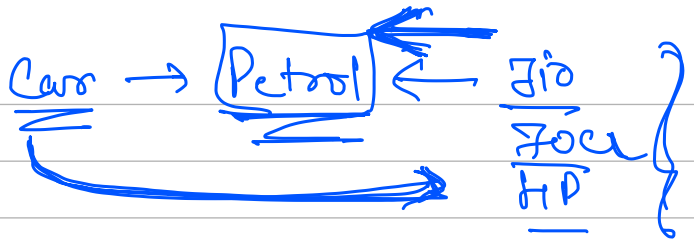
```
Sparrow extends Bird implements Flyable, Dancer
{

}
```

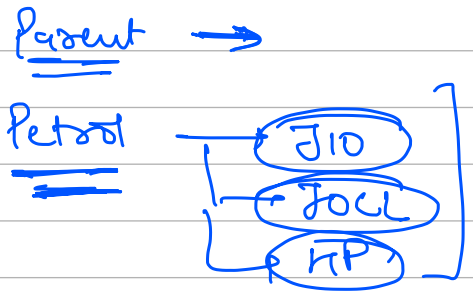
```
Sparrow S = new Sparrow()
Bird b = S;
Flyable f = S;
Dancer d = S;
```

b.fly(); ✗ Compile Exception

```
S.fly();
S.dance();
```

FlyingDancing → Flyable, Dancer


Car
Petrol
Jio
Jocl
HP


Parent
Petrol
Jio
Jocl
HP