Agenda: S O L I D

(1) Single Responsibility Principle
(2) Open - Closed principle.

Class starts at 9:05

S ⟹ Single Responsibility Principle
O ⟹ Open closed "
L ⟹ Liskov Substitution
I ⟹ Interface Segregation
D ⟹ Dependency Inversion

⟹ principle ⟹ set of rules that every S/W Engineer must follow while creating/design a S/W system ⟹ classes, methods, packages

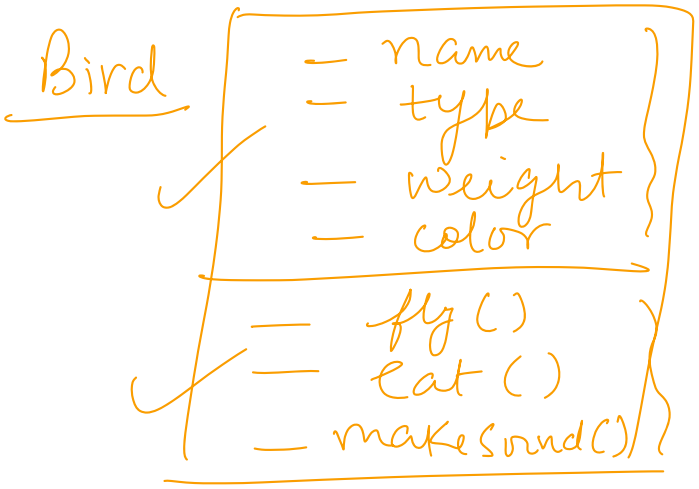
⟹ Extensible, Maintenable, Reusable, Easily Testable, Readable, etc.

⟹ Assume S/W system where we have to store info about a Bird

Bird ⟹ attributes → name
type
color

⟹ methods →

fly()
eat()
makeSound()




```
Bird b1 = new Bird();
b1.name = "abc";
b1.type = "Sparrow";
```

```
Bird b2 = new Bird();
b2.name = "xyz";
b2.type = "Eagle";
```

b1.makeSound() ; b2.makeSound()

```
void makeSound() {
    if type == "sparrow"
```

```
    else if type == "eagle"
```

else if type == "peacock"

}

Problems with multiple if-else

(1) Readability ⇒ if there are more than 10-15 lines of if-else, would be very diff to read.

(2) Test cases ⇒ Multiple test cases would be requried if there are a lot of if-else

(3) Reusability ⇒ probably write his own function with the code of { else if type == "eagle" }

⇒ this would lead to code duplication which would result

in violation of DRY
dont repeat yourself

⇒ Above design of Bird class is not a good design and in fact it is a violation of SRP Single Responsibility principle.

① SRP ⇒ code unit ⇒ { method, class, package }

→ Every code unit should have exactly one responsibility.

⇏ There should be only reason to change the code of that code unit. ⇒ whenever you are updating that particular responsibility.

⇒ Responsibility ⇒

ⓐ method ⇒ print() ⇒ printing something to the terminal

ⓑ Bird Class ⇒ attributes that are

④ going to be generic/common package controllers/utils ⇒ contains all the controllers./utility classes

Bird class is holding the attrs. and also containing the details about how a sparrow will make sound, how an eagle will make sound, diff. birds would eat. And therefore it is violating the SRP.

NOTES : ① LLD is very subjective, there is no absolute correct or incorrect.

② Always be open to counter arguments.

③ Dont do over-engineering.
↳ class fly ✗
class Eat ✗

⇒ How to identify an SRP ?

① Method with multiple if-else ⟹ most likely the method is violating the SRP but there could be cases where this is not true.

Business logic

```
boolean isLeapYear(int year){
    if( year % 400 == 0 )
        return true
    if (      )
}
```

eg:

```
void doSmething ( ) {
    String input = ____ ;
if (input == 1) {
    ____ going to start
    ____ finding chrome
    ____ opening chrome.
}
if (input == 2) {
    ____
    ____ playing music
}
```

First Iteration

```
        }
        if (input == 3) {
            drawing a picture
        }
    }
}
```

Soln:

```
void doSomething() {
    if (input == 1) {
        openBrowser()
    }
    if (input == 2) {
        playMusic()
    }
}
```

Second Iteration

What would be the only reason to change

(1) First Iteration ⇒ if there is new user input and also how a browser would open, how music would be played.

(2) Second Iteration ⇒ if there is a new user input or anything

related to user inputs.

② Monster Methods ⇒ when the method is doing much more than its name is suggesting.

```
void saveToDatabase(User user) {
    ① String dbQuery = "Insert into table users..."
    ② DatabaseConnection db = new DatabaseCo
                              -nnection();
    ③ db.execute();
}
```

Problems ① Can there be other code that wants to know the query to insert a user ⇒ yes

② Can there be other code which wants to know how to get a DB connection. ⇒ yes.

```
void saveToDatabase(User user) {
    ① String dbQuery = getUserInsertQuery();
    ② DatabaseConnection db = getDBConnection();

    db.execute();
}
```

⇒ Biggest advantage of SRP ⇒ Reusability

③ Common / Utils package ⇒

⇒ Whenever an engineer is not able to decide on the correct place for a method, the engineer puts that code inside the utils folder / class, which basically turns into a dumping ground.

com.scaler.utils ⇒ date utils, String utils all in one class — bad design

com.scaler.utils ⇒ { DateUtils class, StringUtils class — better design

com.Scaler.User ⟹ { User.class, UserUtils.class } even better design

Summary of SRP ⟹ (1) Every code unit should have exactly one responsibility.
(2) There should be only 1 reason to change the code of that code unit.

(3) How to identify the SRP ⟹
(a) Multiple if-else
(b) monster methods
(c) Utils package.

Biggest advantage of SRP ⟹ Reusability.

Break till 10:34

⟹ There's another principle which it is violating ⟹ SO

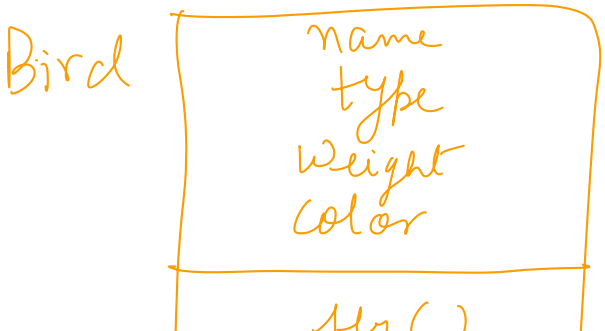
(2) Open closed principle ⟹ Codebase

should be open for extension and closed for modification.

⇒ While adding new features, I should not be modifying the existing codebase.

⇒ If we have to add new features and we are not modifying the existing codebase, then we can add new code units ⇒ class, methods, package.

More practical defⁿ ⇒ Whenever you are adding a new feature, dont modify the existing code, and if you have to modify keep it as minimal as possible.



makeSound() {
if (type == "sparrow")

makeSound()
eat()

if (type == "eagle")

if (type == "peacock")

Violation of OCP as we have modified the existing code unit.

Let us see the reasons for OCP ⇒

1. Test Cases ⇒ I'll have to modify the test cases as well.
2. Regression ⇒ Whenever you are adding a new code, if it causes the existing code to break, then the regression has failed.

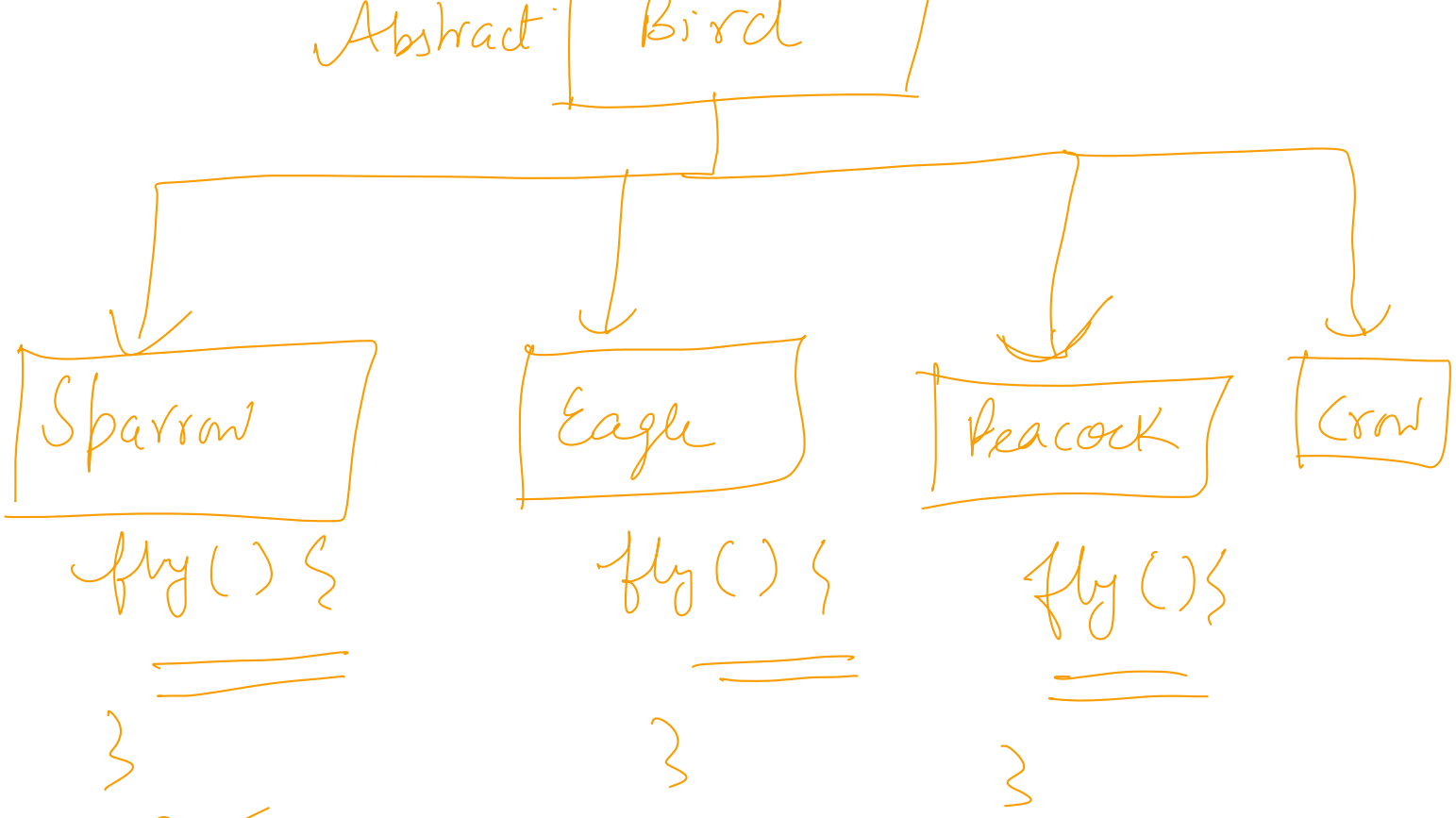
V2 of Bird

abstract

Abstract

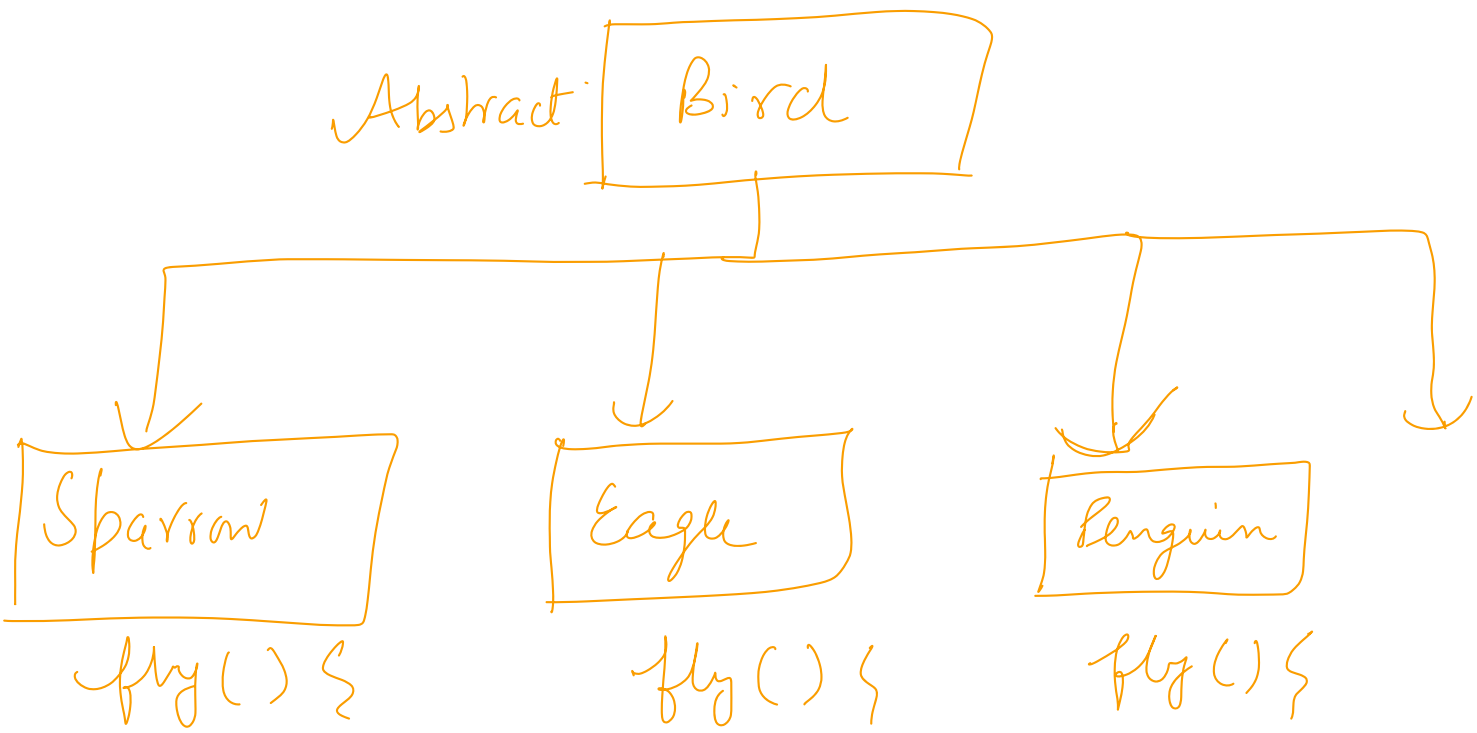
name
type
color
weight

fly();
makeSound();
eat();



OCP is being followed

Bird class has lesser reasons to change and its responsibility has been limited ⇒ SRP is being followed.



} }

⇒ Penguin cant fly
① Dummy implementation of fly
② Throw an exception.

Client {

Bird b = getBird( ); ⇒ penguin
b.fly( ) ⇒ client will be surprised over here.

}

⇒ Never ever surprise the client

eg: UberEats ⇒ used to use Paytm API for payments.

```
UberEats {
    Paytm api;
    checkOut() {
        if (api.paymentStatus != failure) {
            order = successful;
```

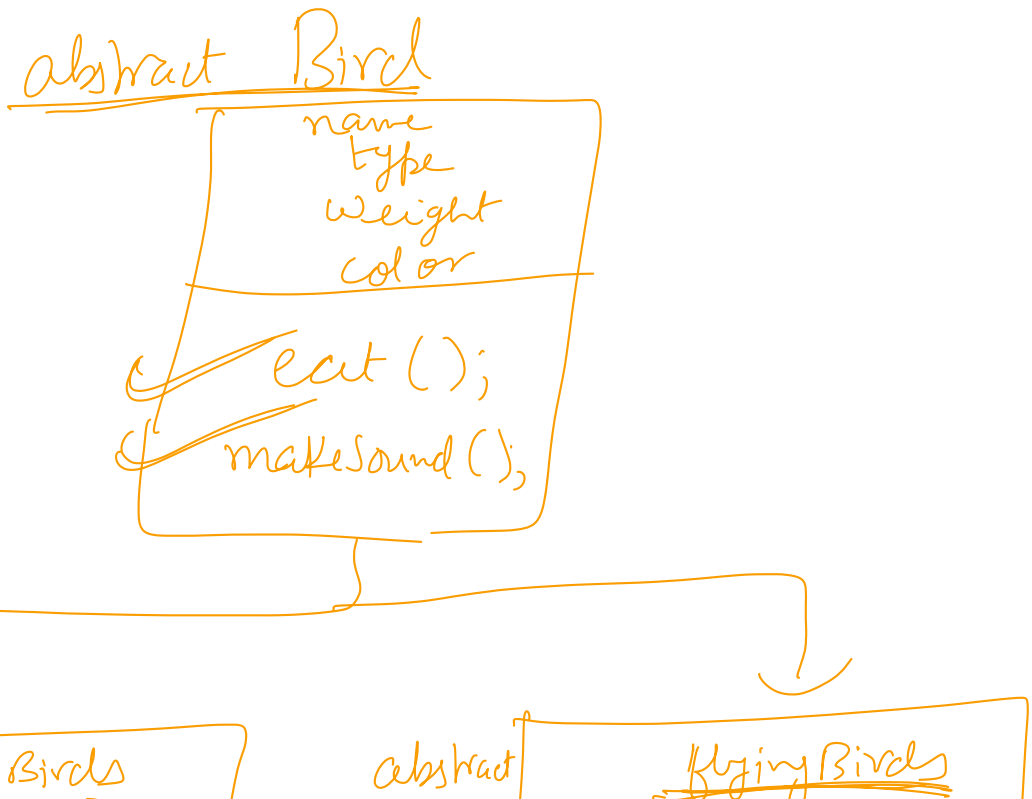
processOrder();

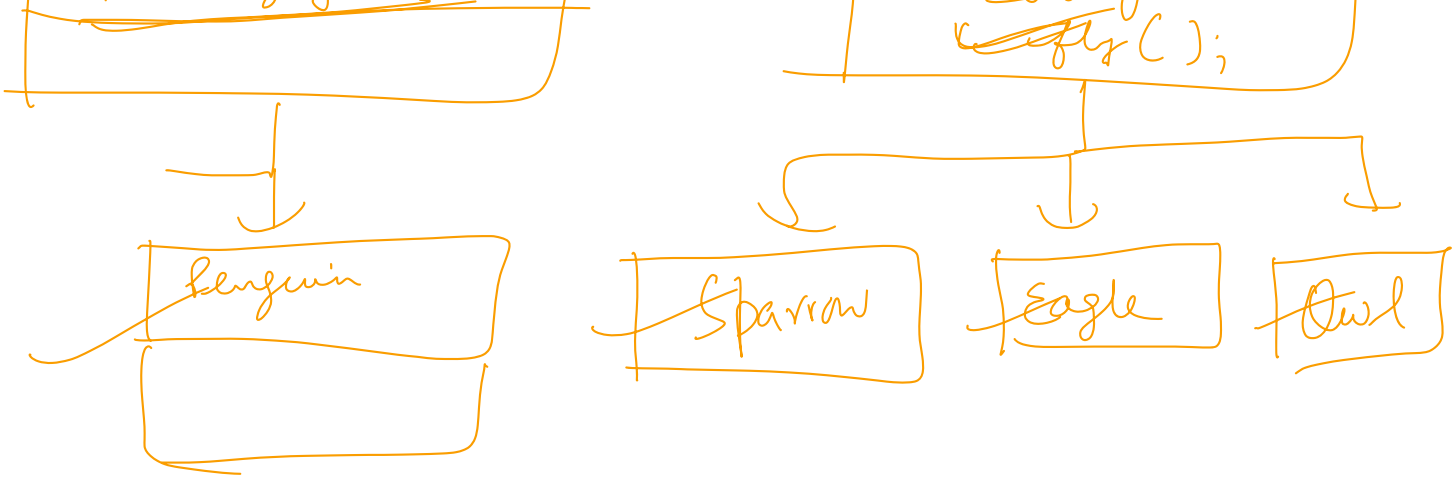
}

⟹ paytm status = Success or failure

⟹ paytm status = Success or failure or Bank failure

(Ideal soln. for such cases (like penguin)
⟹ dont have such methods which your entity cannot fulfill. rather than having a dummy implementation or throwing an exception.

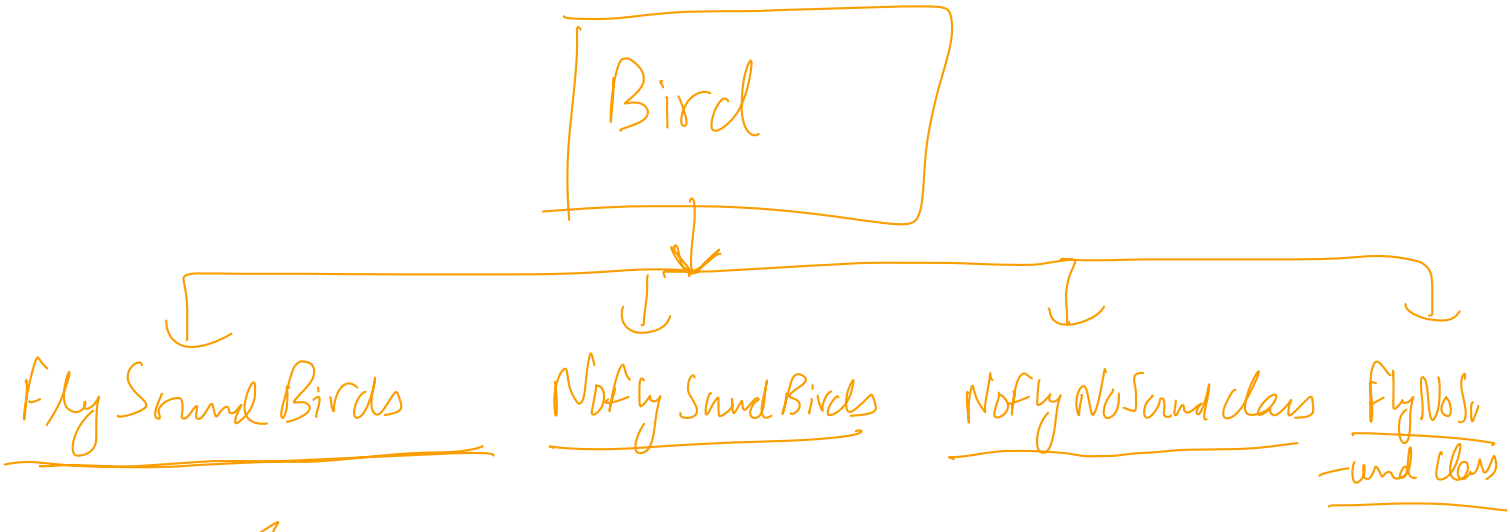




List <FlyingBirds> ⟶

List <NonFlyingBirds> ⟶

n=2	makeSound()	fly()
⟶	✓	✗
⟶	X	✓
⟶	✓	✓
⟶	X	X



⟹ n behaviours ⟹

⟹ 2^n ⟹

↳ ① Class Explosion ⇒ would result in a lot of classes

② List<FlyingBirds> ⇒ this would not be possible then.

↳ Solve this Bird problem
LID of SOLID

TODO ⇒ Check assignments