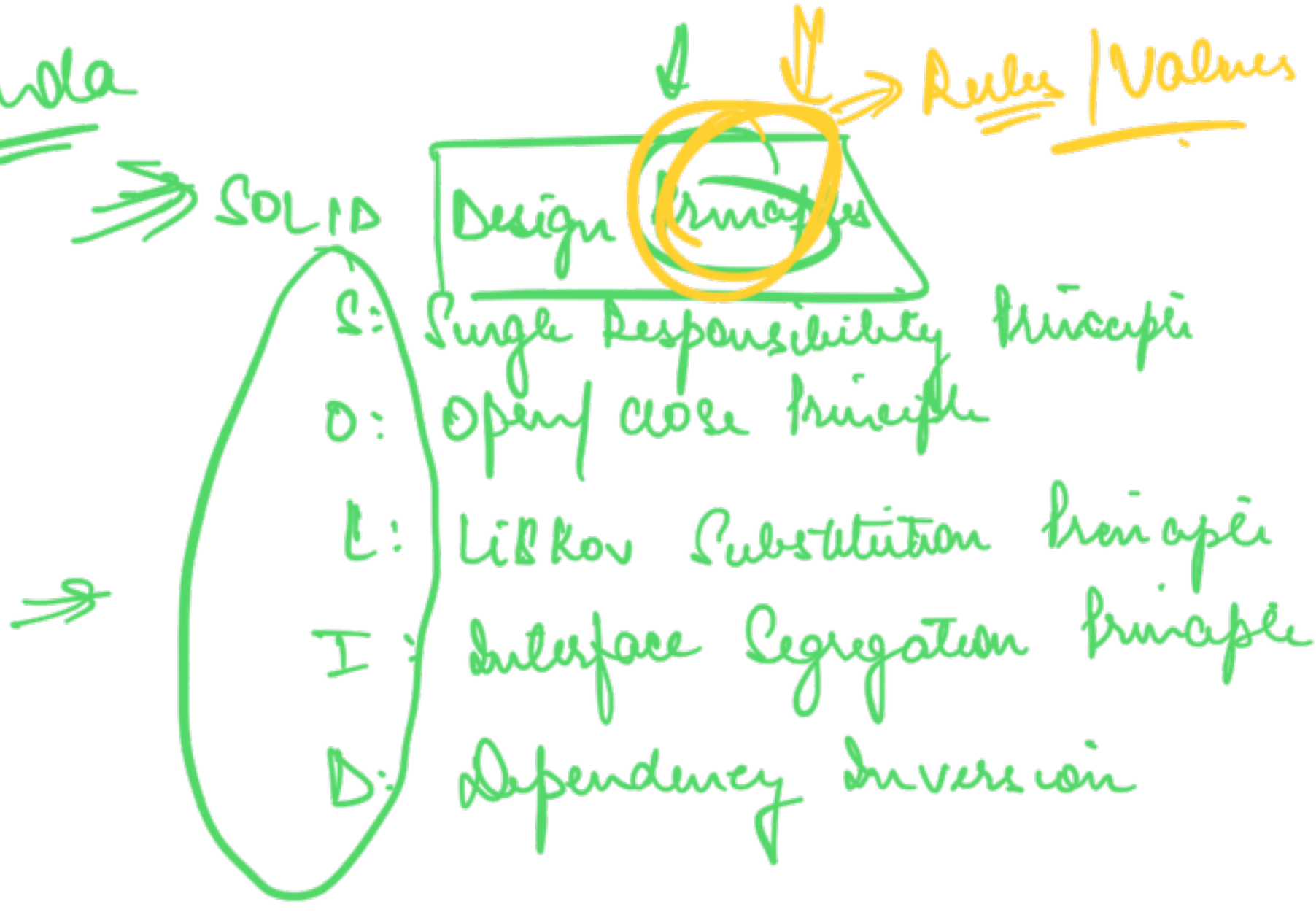
SOLID Design Principles

Agenda



Values

Be Nice

Be Respectful

DP: Moral values to be considered for a good software

S → ≠ → S

O → ≤ O/C

L

I

D

Final Code

① Reusable

② Maintainable

③ Understandable

④ Extensible

⑤ Modular

S → Single Responsibility Principle

Every code unit (class / fn / package) should have a single, well defined responsibility. There should be ONLY ONE reason to change the code unit.

Bird {

are there Bird in bird

① when I have to add a new Bird

② add new attributes of Bird

SRP

Class Bird {

Species

color

2 Responsibilities

① Manage all attributes & behaviour of a

# of wings
flying speed
lifespan

fly()
eat()
chirp()

}

bird

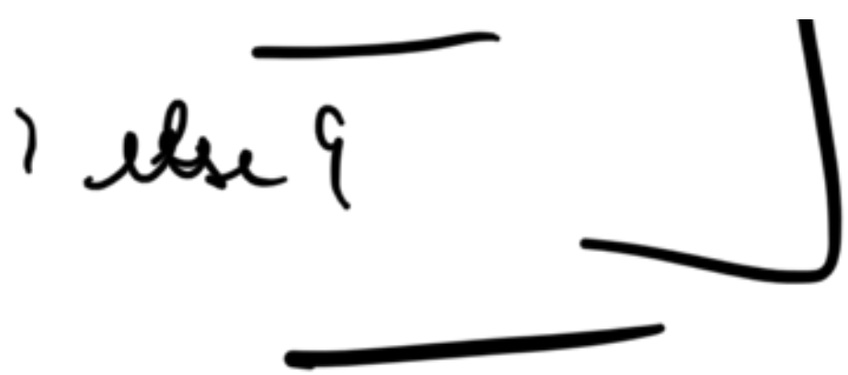
① Manage how every bird will fly

③ How will each bird eat

fly() {

if (species == "pigeon") {

} else if (species == crow) {



How to identify if SRP is getting violated

① A lot of if-else] 90% Switch-case

→ when something is resp for multiple types ⇒ violation of SRP

when if-else are true to implement business logic.

② Monster Method

⇒ One having logic for multiple other things than what their name says.

```
Database {

    Save(File file) {
```

→ delegated to some other method

R1

FileSystem sys = File (file)
Sys.open()
String file = sys.read

R2

Query q = new String (
Select
from
)

createQuery()

R3

q.save()

LoggingServer s = new Logger
s.log(q)

.log( ) R4

Monster method

Mix

⇒ Save file (file) {

file.open(file);

DB.createQuery(file.Name)

DB.Save

Logger.log(file);

}

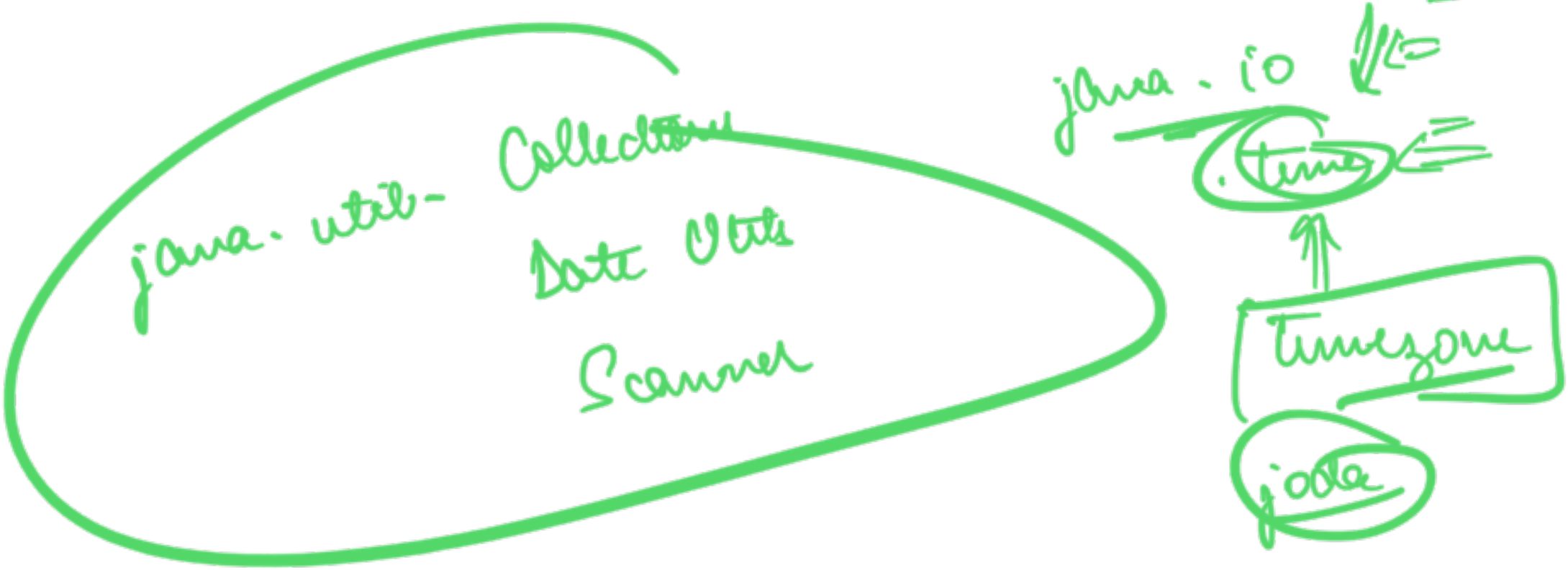
SR

③ Util Classes / Packages ← (common helpers)

→ Will end up violating SRP

→ Don't allow you to even create a folder called Util.

→ junkyard of anything and everything that is needed by multiple classes.



com.java.date

come.jave.collection

.java.io

com.java.util

src/util/

DateUtil.java

TimeUtil.java

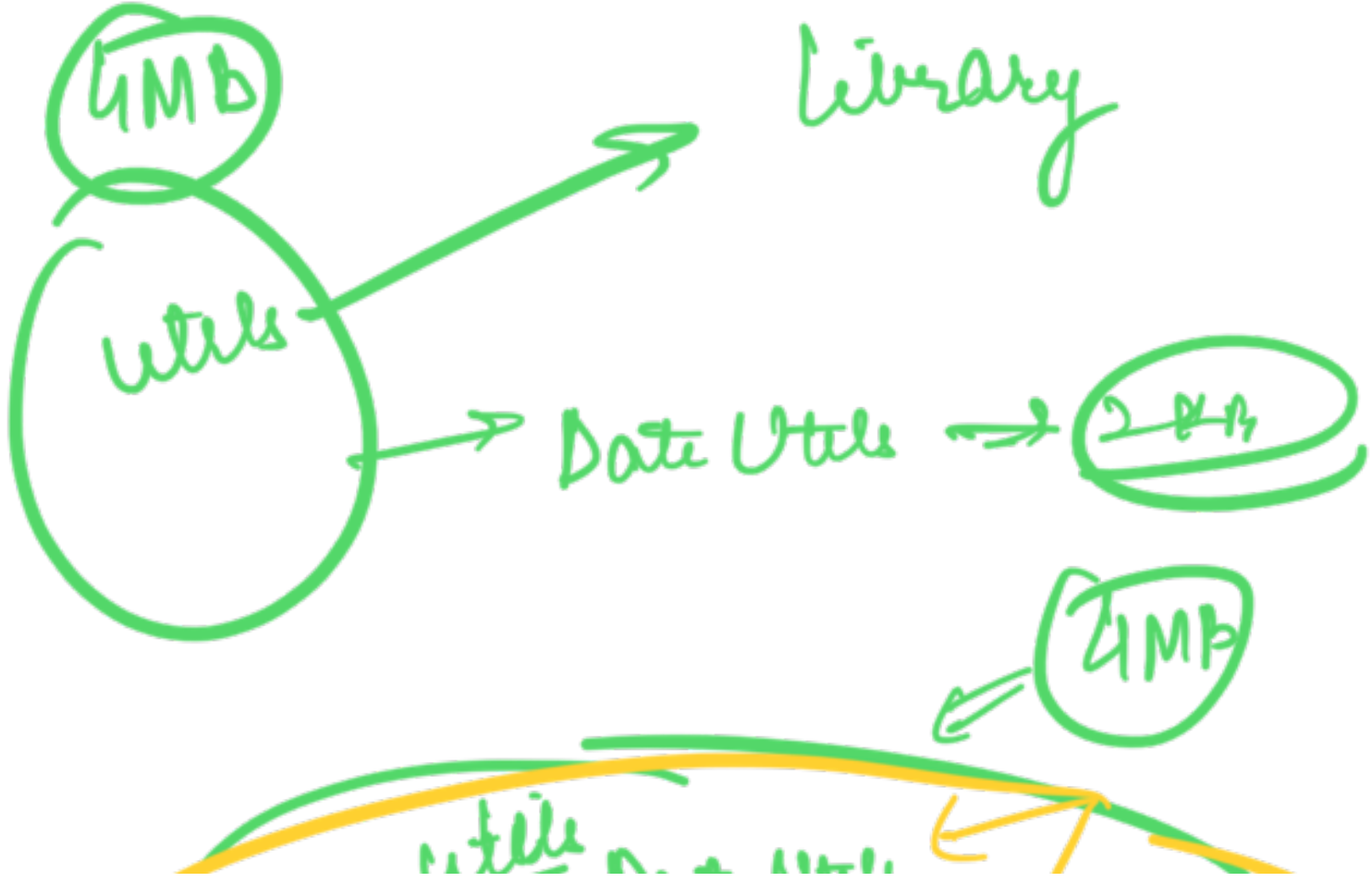
CalUtil.java

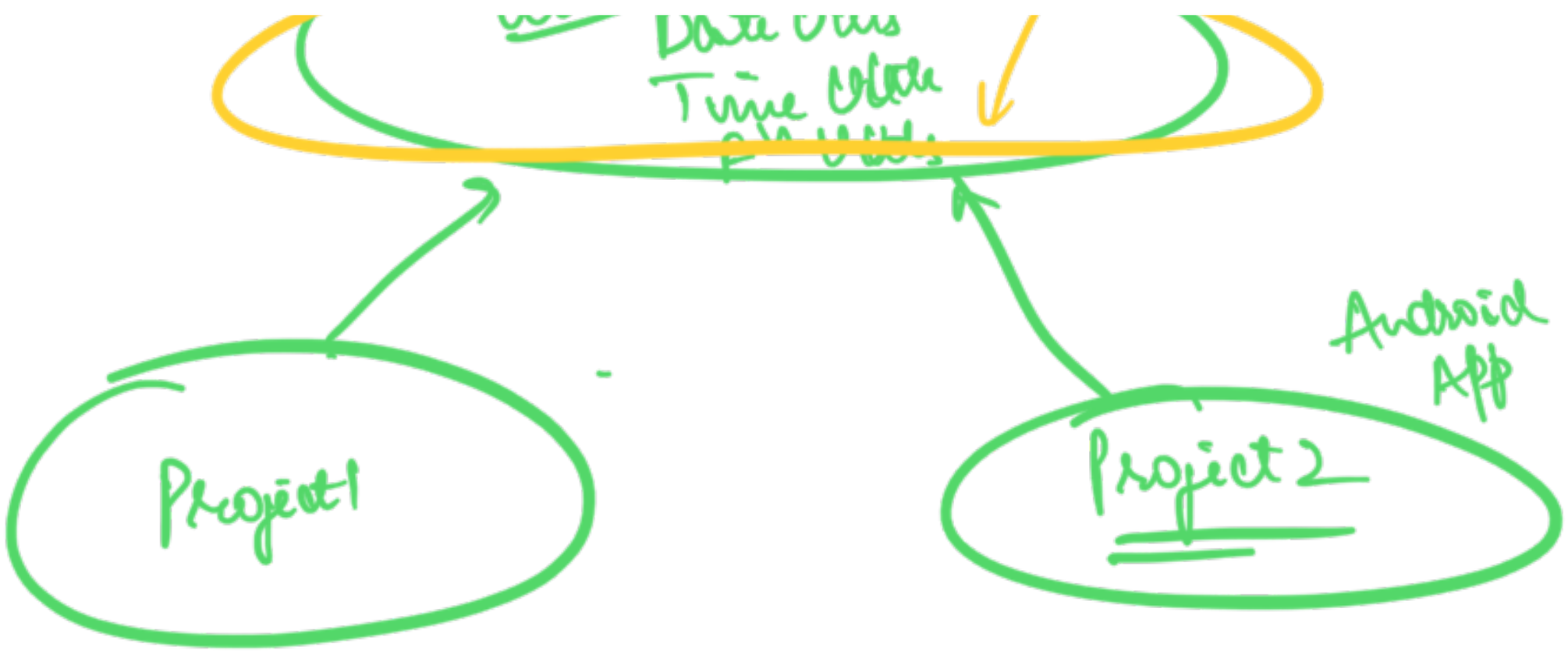
JSONUtil.java

DBUtil.java

src / dat / Date.java

/ time / Time.java





Create Time.java

Benefits of CRP

1. Reusability

2. Test (Unit Test)

3. Maintain

4. Modularity

Class ↔ Test Class

Method → More than one Testing

Test all diff cases

Too many code units

⇒ Following any DP 100% is impossible.

DP are also subjective

⇒ Able to tell Pros and Cons of your choice

O ⇒ Open/Close Principle

When I am adding a new feature to my codebase, my codebase should be

OPEN for extension

but CLOSED for modification

Class Bird {

Species
#of legs
Color

fly() {
if Species == parrot
elif Species == pigeon
else
}

}

Grow

O/C fr
Modifying X

Flying
Abstract

abstract

interface

O/C ⇒ ✓

O/C Principle == SAP

↳ Liskov's Substitution Principle

Objects of a Parent class should be replaceable by object of child class w/o the client needing to change its code base.

The codebase of client should keep working as-is.

Parrot()

Penguin()

Crow()

Bat()

Bird b = new Bird();

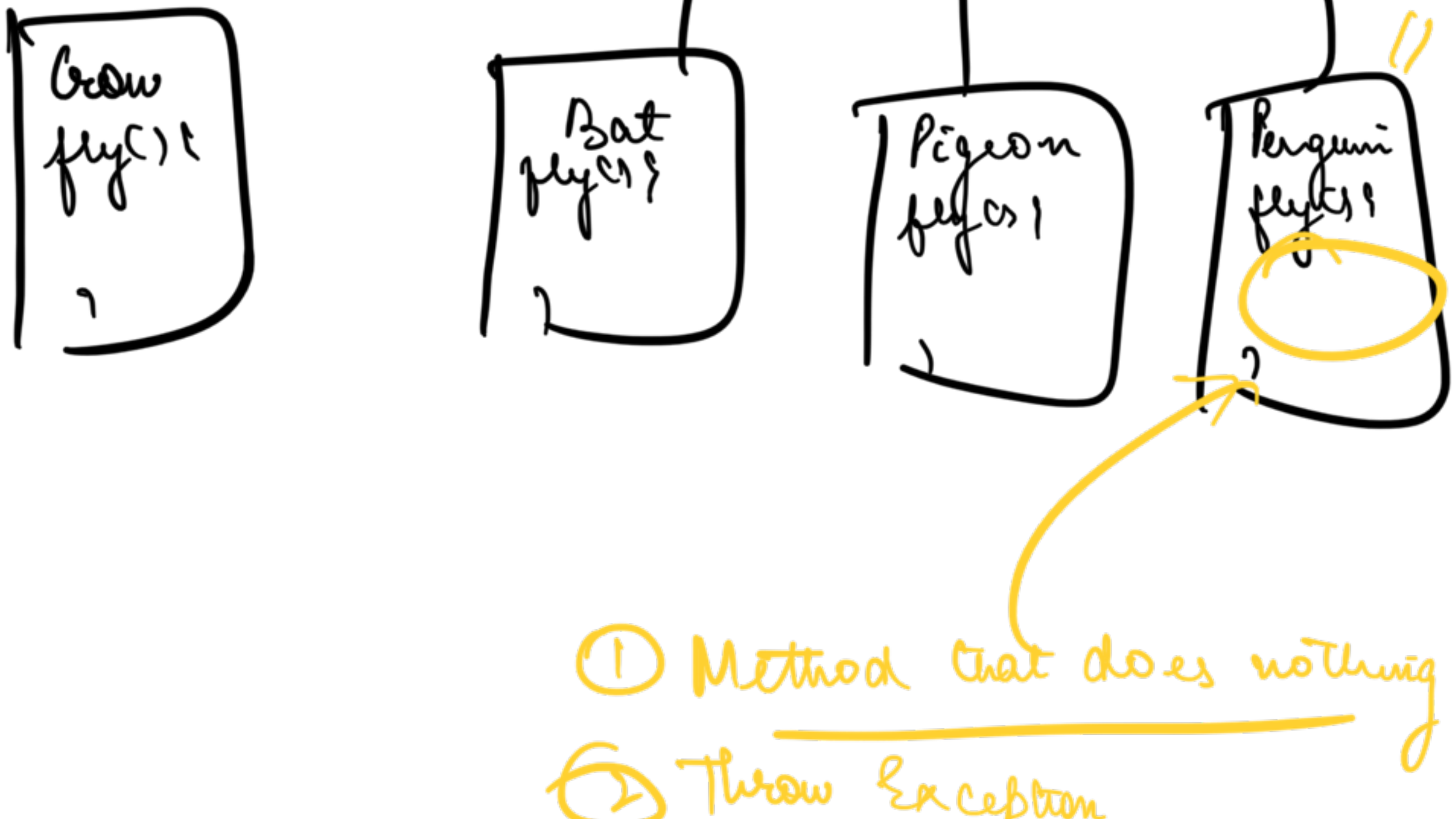
① handle any

X No code change needed here.

(2) Code should behave as expected

Another defⁿ —

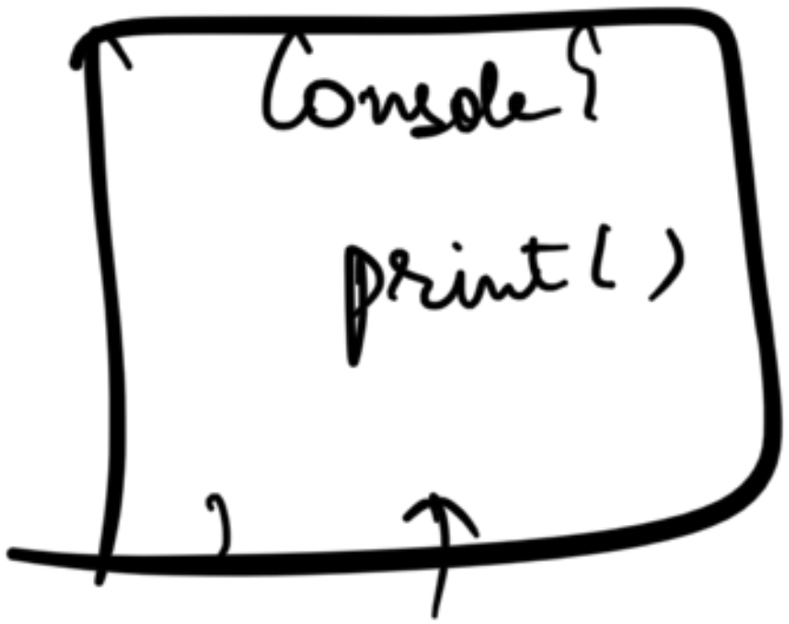
→ Subclass shouldn't give a different definition / behaviour to the Methods of a parent Class.


Crow
fly()
Bat
fly()
Pigeon
fly()
Penguin
fly()
1 Method that does nothing
2 Throw Exception

Bird Race {
- List <Bird> ;

Crow, pigeon, Penguin

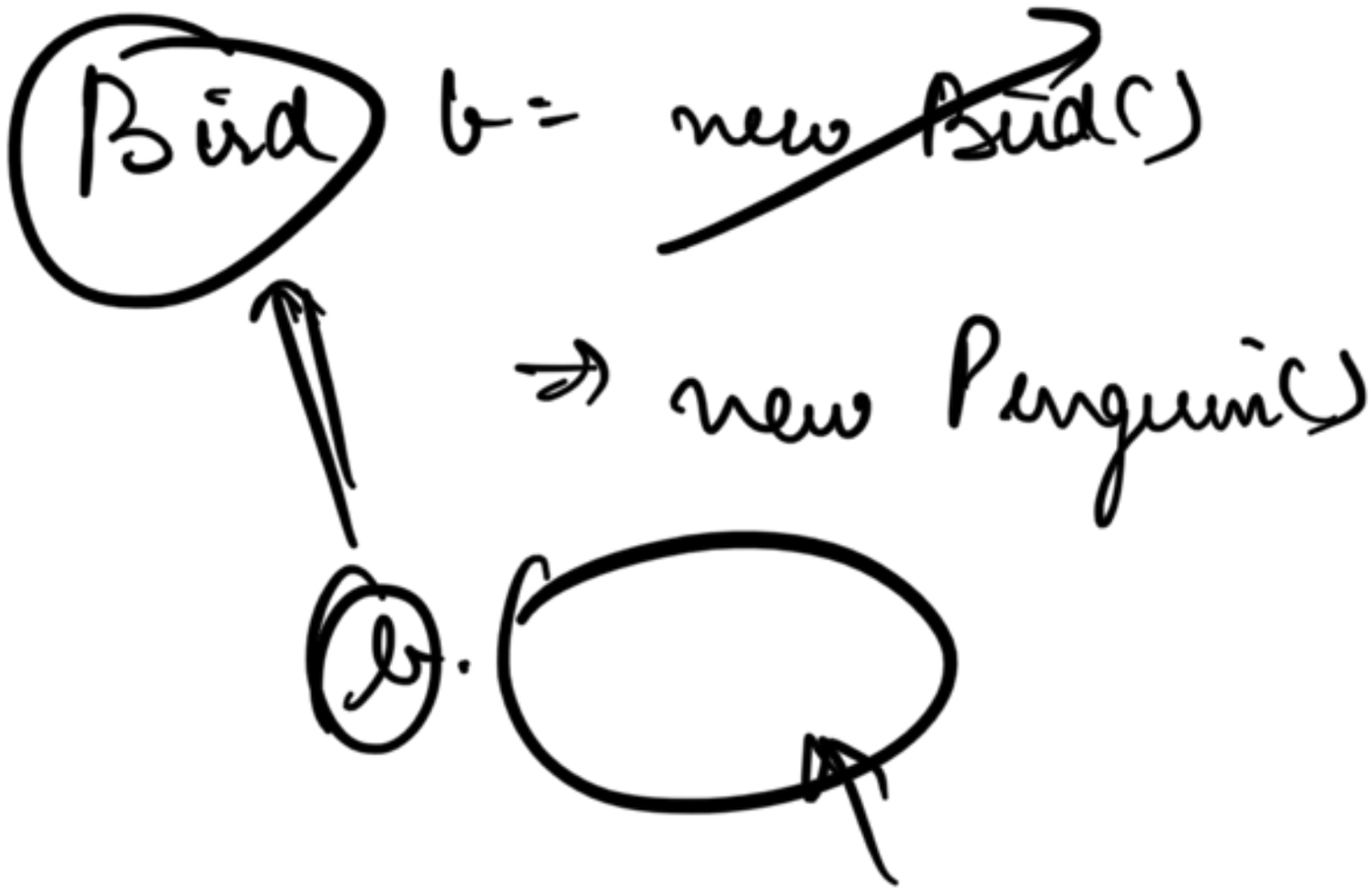
start() {
for (Bird b: birds) {
b.fly()
}
}
}



Page Printer {

print ( ) ⇐ Diff meaning of print

}



① Method of a child class should not:

a) Give New meaning to any method of a parent class.

Bird c

fly() {

}

}

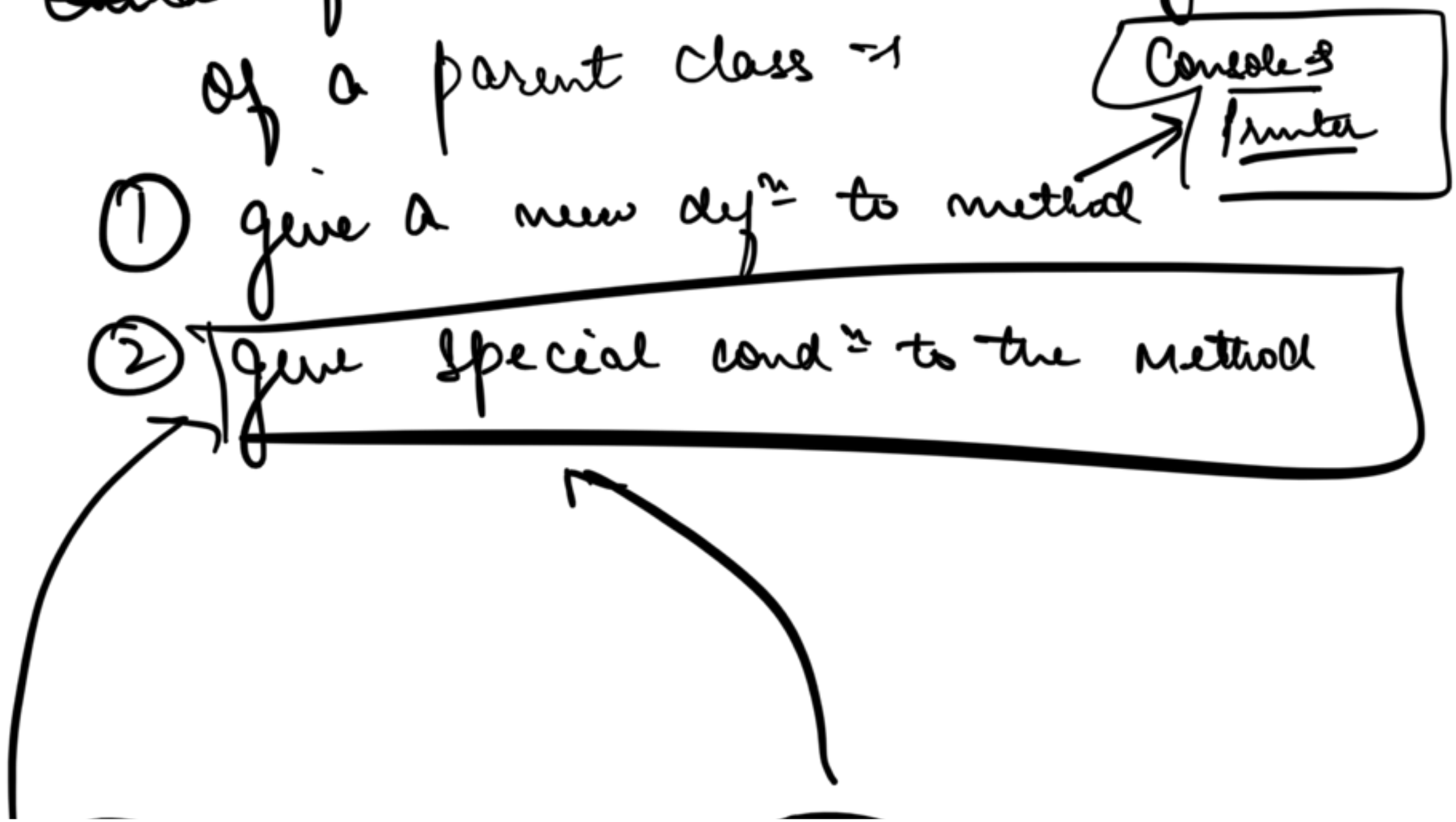
② ~~Rea~~ While overriding any method, child class shouldn't give a new behaviour

- → if no response → Not imp
- → if exception → Not imp

for (Flyable f : flyables)

L ⇒ Liskov's Substitution Principle

If child class overrides any behaviour of a parent class -1



fly()

fly() {
throw Exception
}

→ Composition Instead of inheritance

Animal {
makeSound() → Animal will

}

Cooker

makeSound() {

}

We override when the particular behaviour is done by the child in a diff way.

pigeonC14
Polygon
getArea()
Rect
Tri

① dimension of class

② how to calculate area

AreaCalc area = new RectAreaCalc();

25 more Mins

5 Min 10:48 AM

1, D, Code

I ⇒ Interface Segregation

① Keep interfaces as light as possible

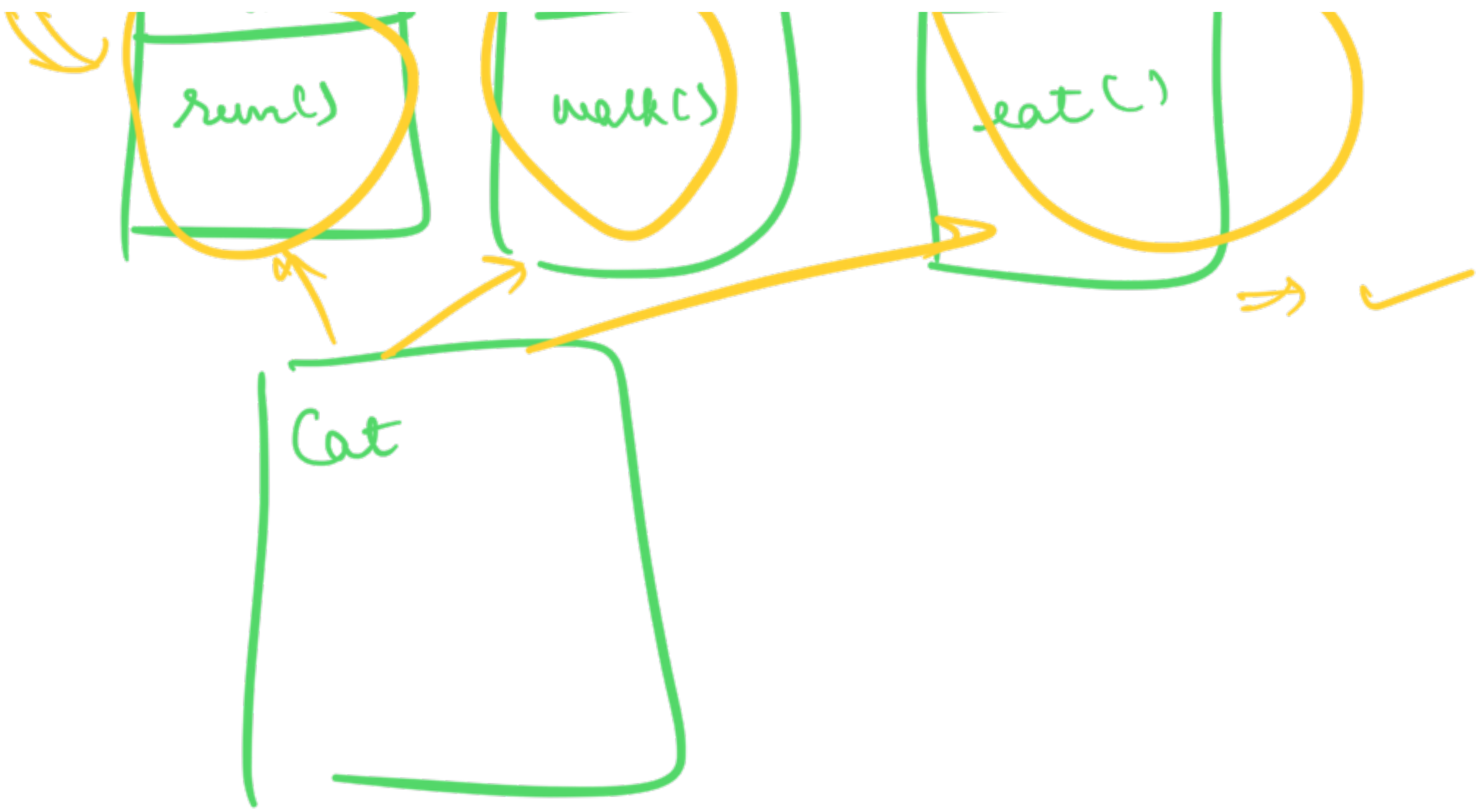
↳ ideally have 1 method.

(2) Interfaces should not be bulky

(3) Clients should prefer implementing multiple smaller interfaces rather than one big interface

(4) Make interfaces very SPECIFIC ⇒ SRP

⇐ ✗ Interface Segregation



class Cat implements Runnable,

---

(5) D ⇒ Dependency Inversion Principle

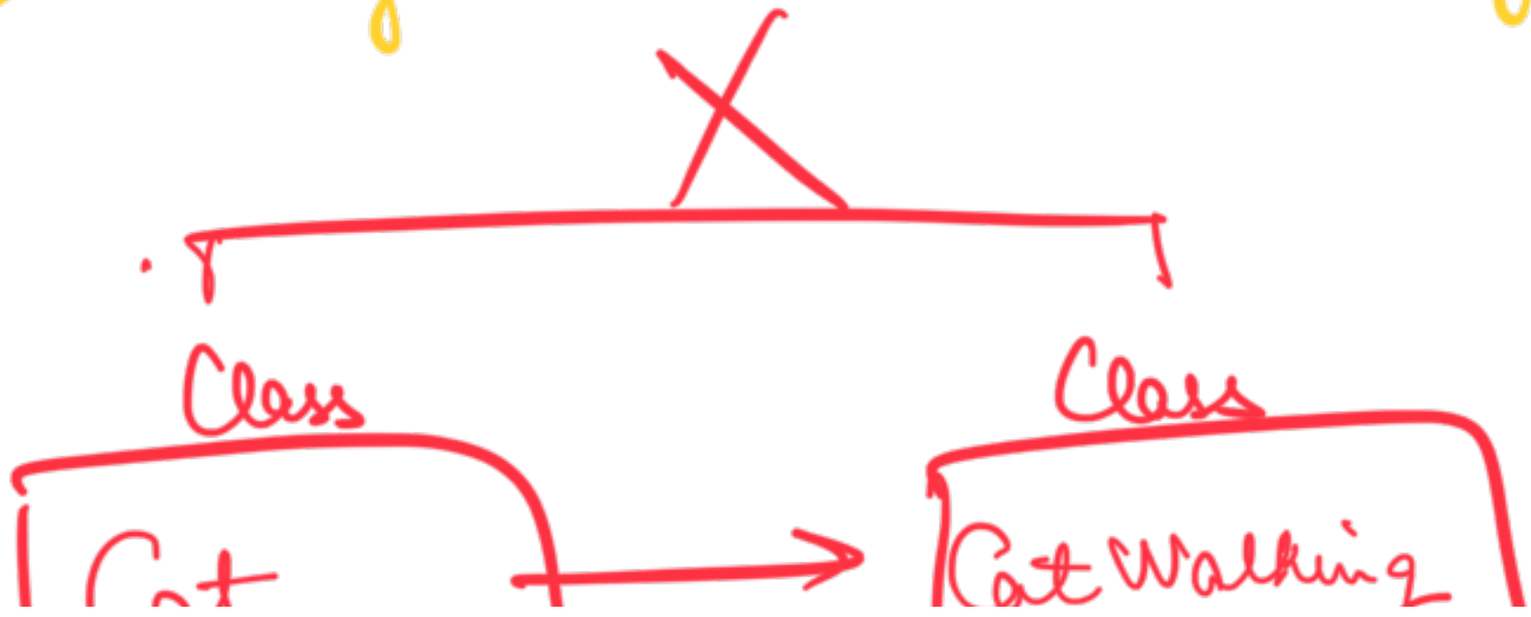
No 2 Concrete classes should directly depend on each other. Instead they should depend on each other via an interface in b/w

→ Loose Classes ✓

Cat {

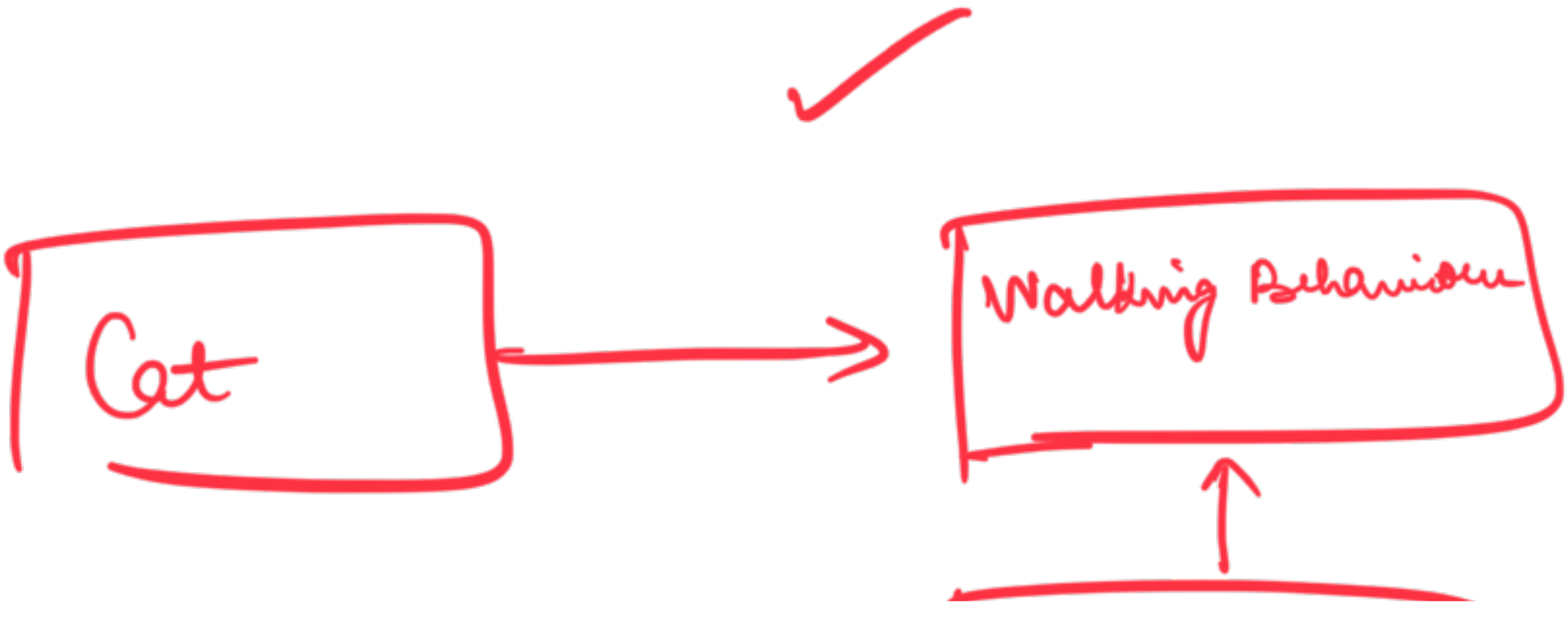
Cat Walking Behaviour = new Cat Walking Behaviour

}



Cat c

Walking Behaviour = new Cat Walking Behaviour();



Cat Walking Behaviour

Case Study

Yes Bank was banned by RBI

< 24 hrs
1 night

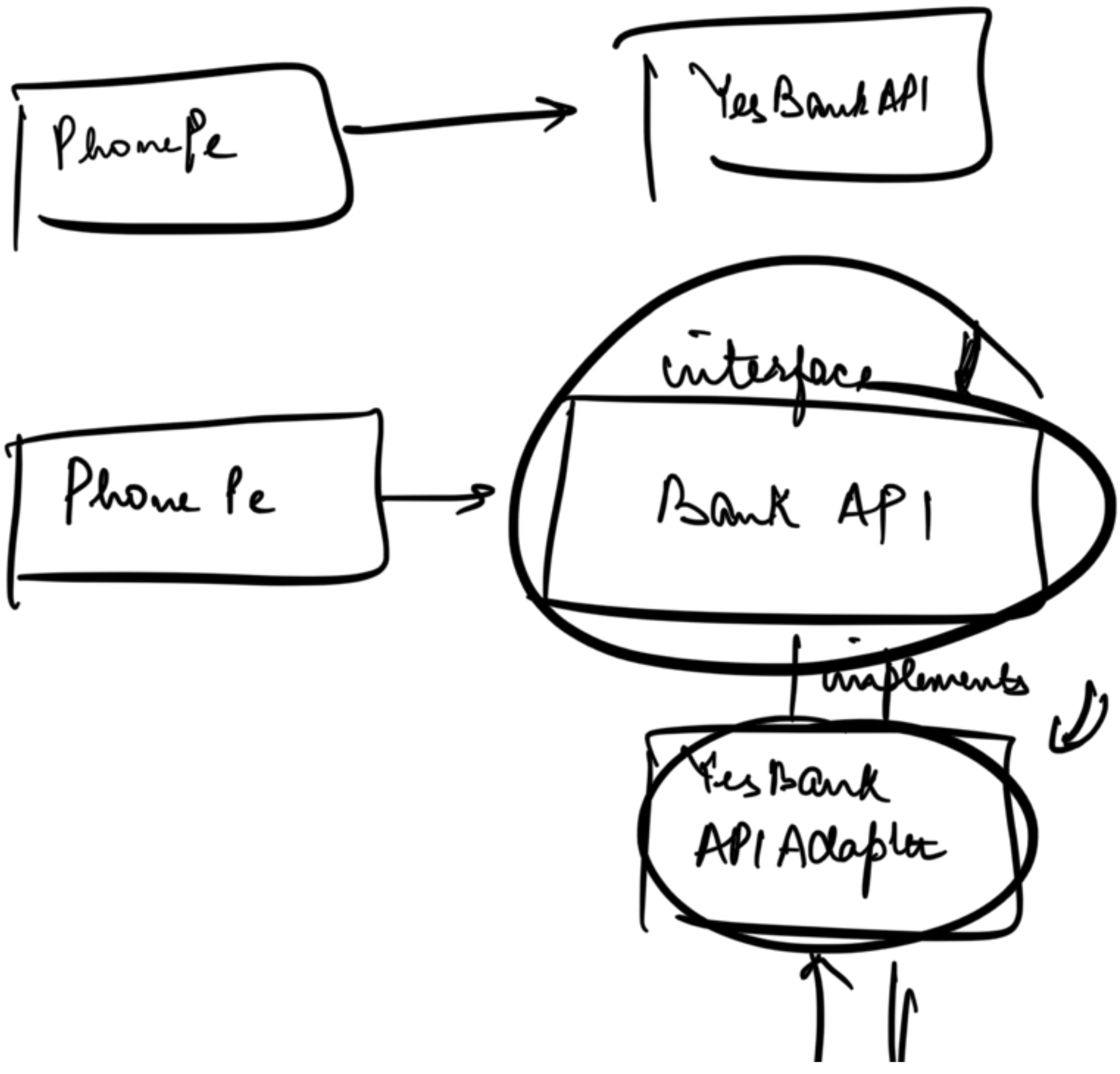
PhonePe § BankAPI ICICI PayAdapter()

api = new

pay(){
  api.getPaymentInfo()
}

getBalance(){
  api.getBalance()
}

```
receive() {
    api.receive()
}
```


PhonePe
Yes Bank API
PhonePe
interface
Bank API
implements
Yes Bank
API Adapter

YESBankAPIAdapter implements BankAPI {

YESBankAPI = new YesBankAPI

}

ICICIBankAPIAdapter implements BankAPI {