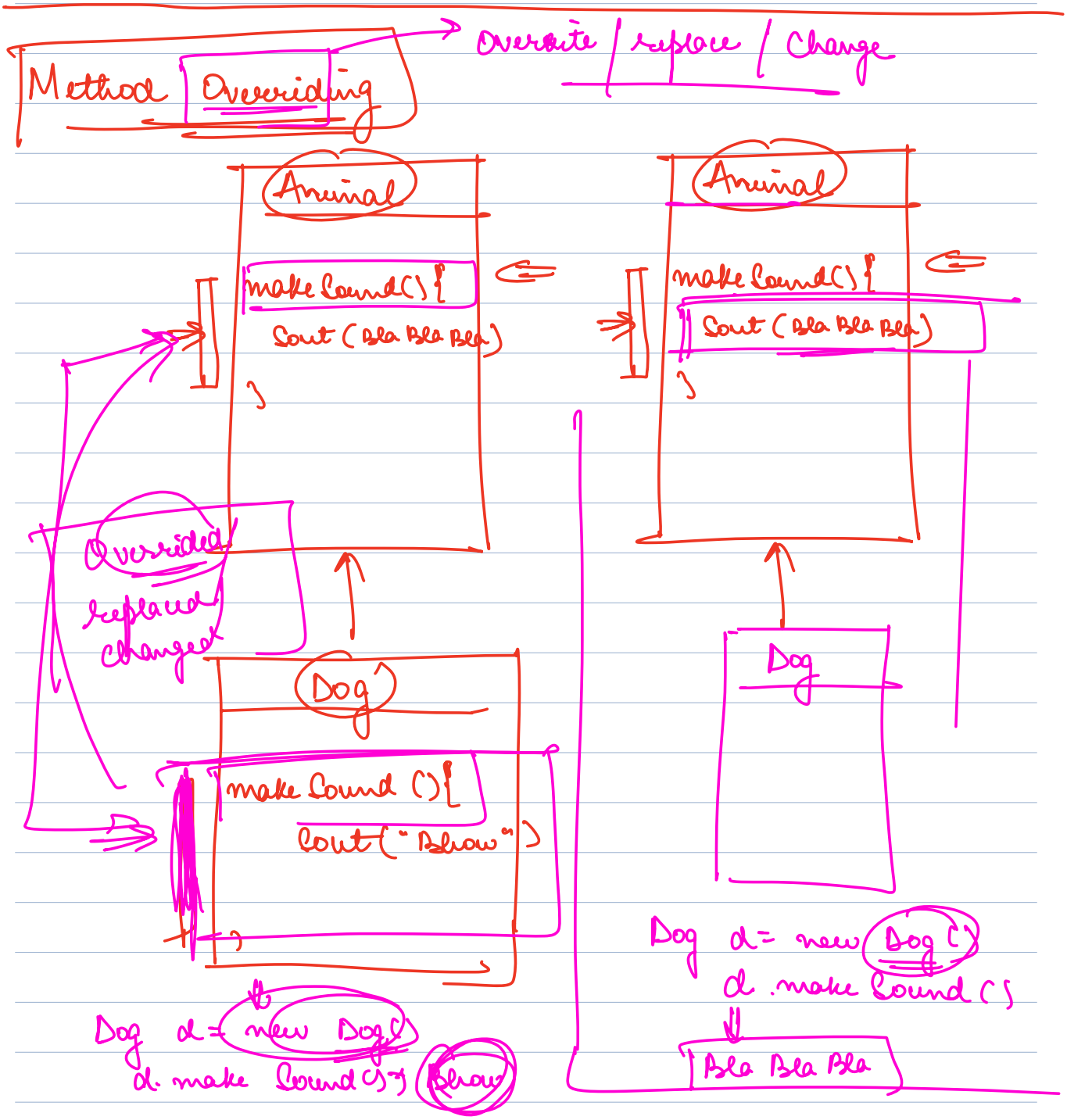
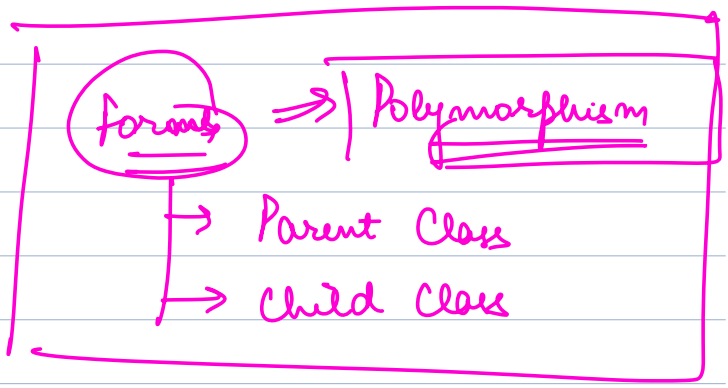
Agenda

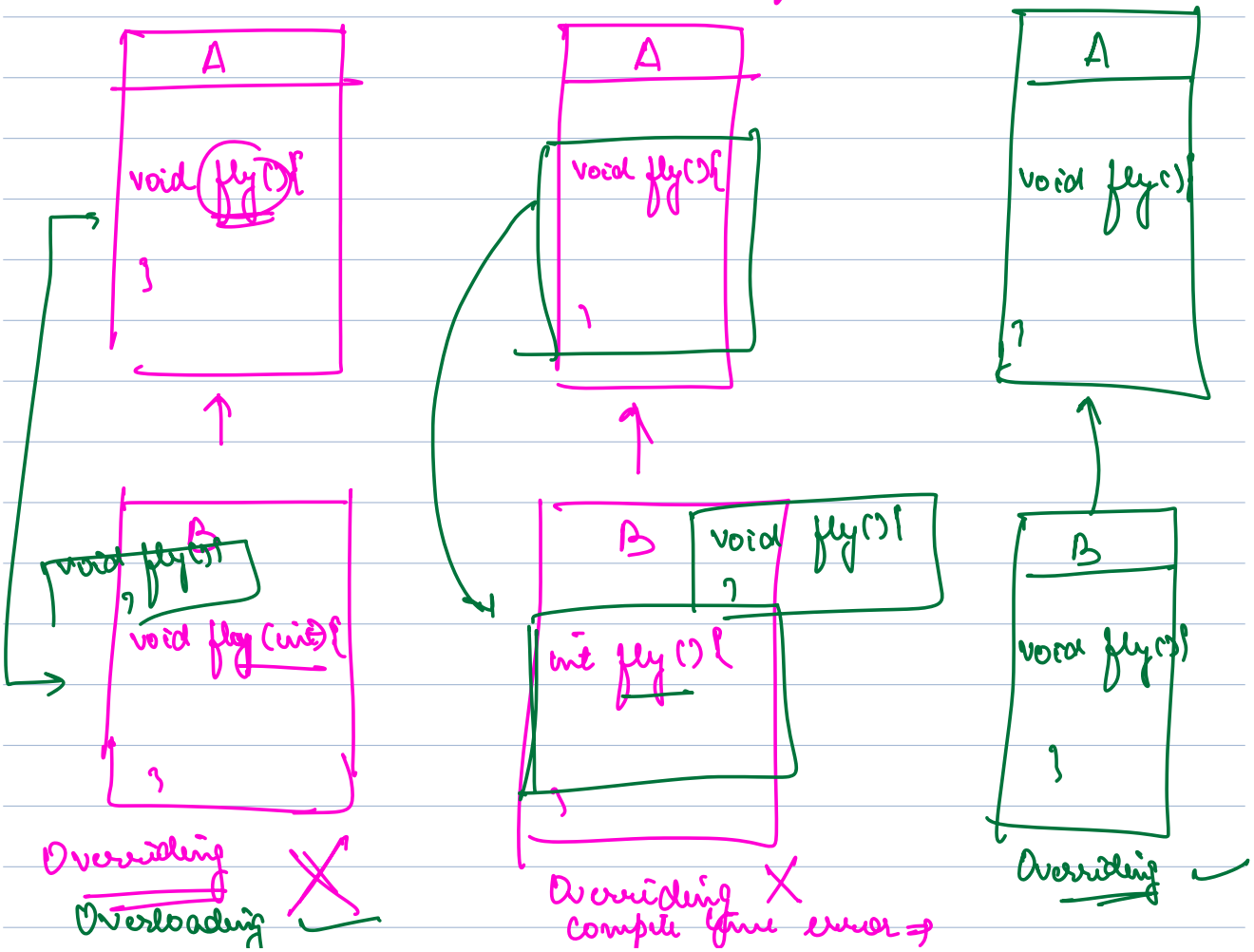
1. Overriding
2. Interfaces
3. Abstract Classes
4. Static

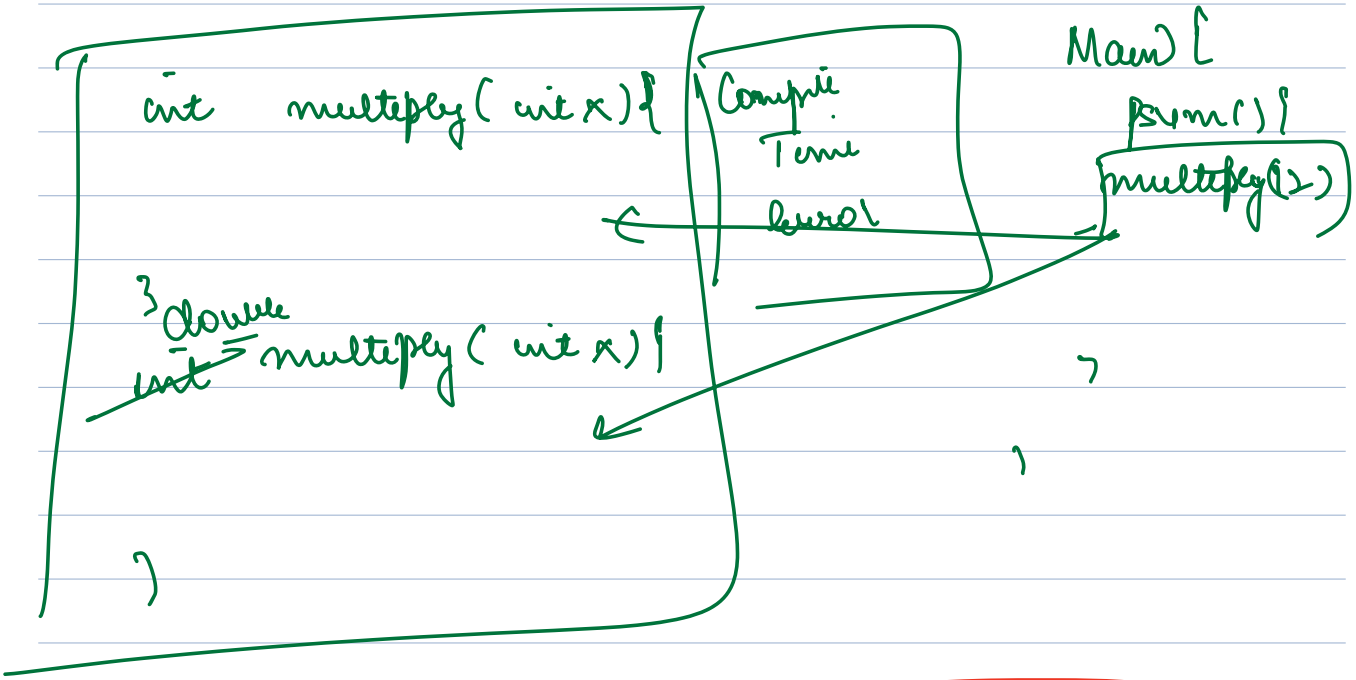


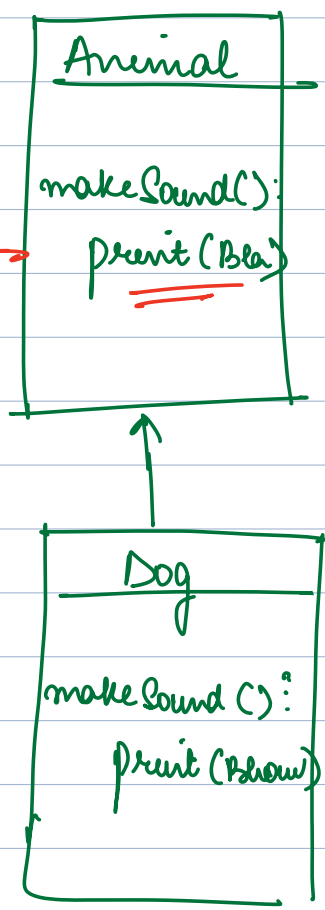
make bound ⇒



Overriding ⇒ A method with same signature and same return type present in the parent class as well as in the child class.
Method of child class overrides the method of parent class.







Animal a = new Animal()
a.makeSound() → Bla

Dog d = new Dog()
d.makeSound() → Bhow

Dog d = new Animal()
d.makeSound()

Animal a = new Dog();
a.makeSound()

→ Bhow

⇒ Method that gets printed is of the object that is stored in the variable, not of data type of variable.

```
A {
    doSomething()
    ⇒ hello
}

B extends A {
    doSomething():
    ⇒ hi
}

C extends B {
    doSomething:
    ⇒ Heya
}
```

```
List<A> as = {new A(), new B(), new C()};

for (A a : as)
    as.doSomething()
```

⇒

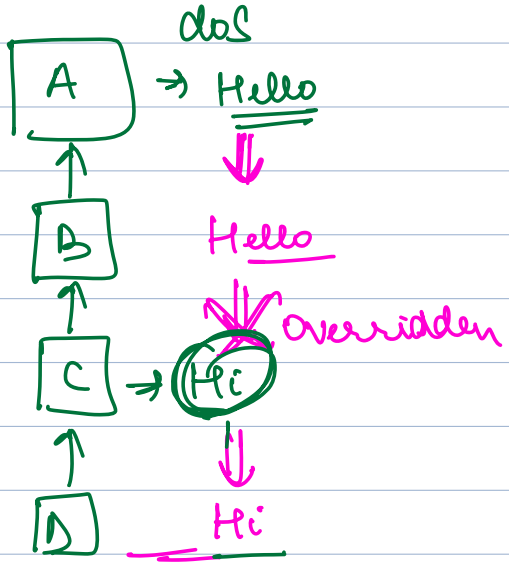
```
hello
hi
heya
```

```
A   get A() {
        int i = rand();
        if (i % 3 == 0)
            return new A();
        elif (i % 3 == 1)
            return new B();
    } else  return new C().
```

```
A  a = getA()
a.doSomething()
```

→ I will only know at runtime what will get printed

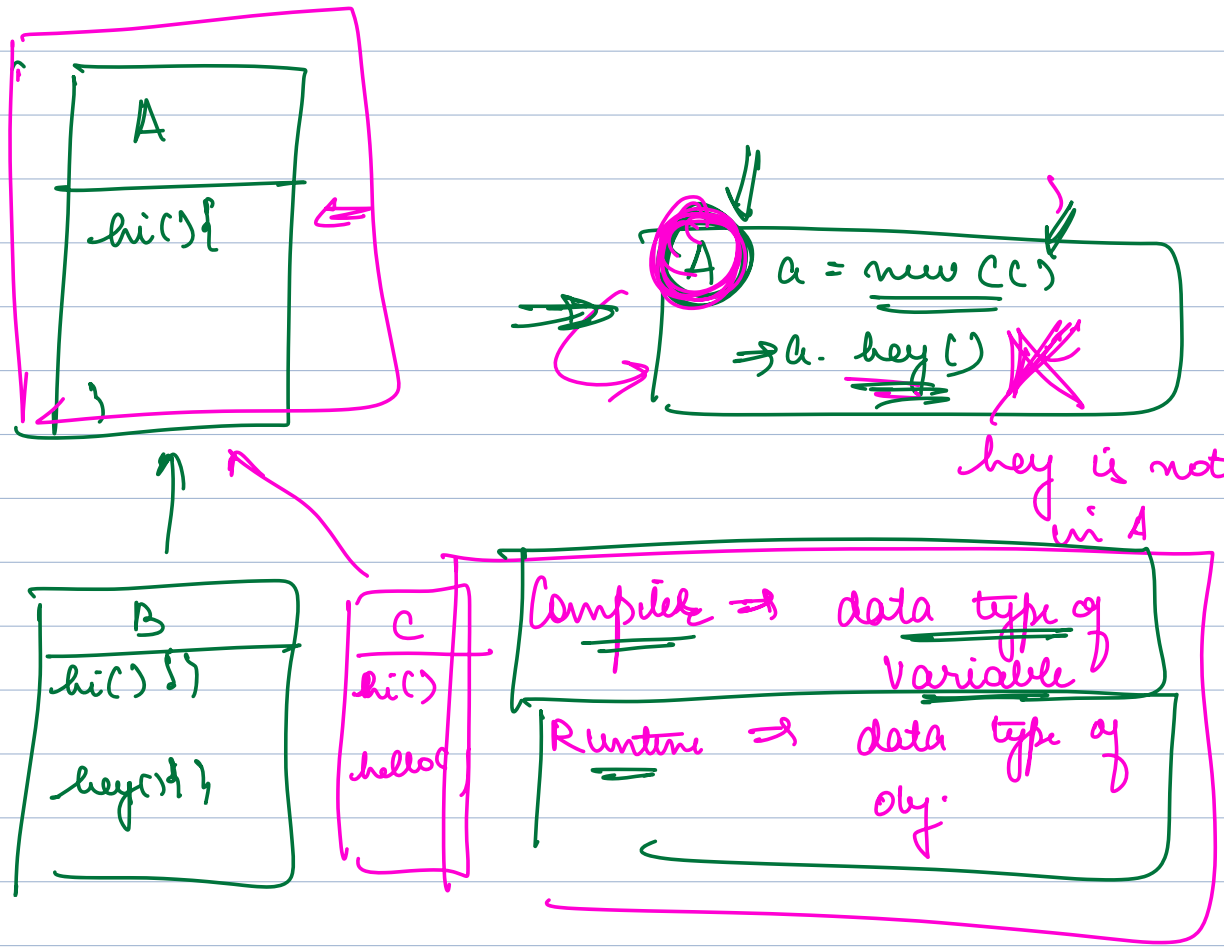
→ Runtime Polymorphism

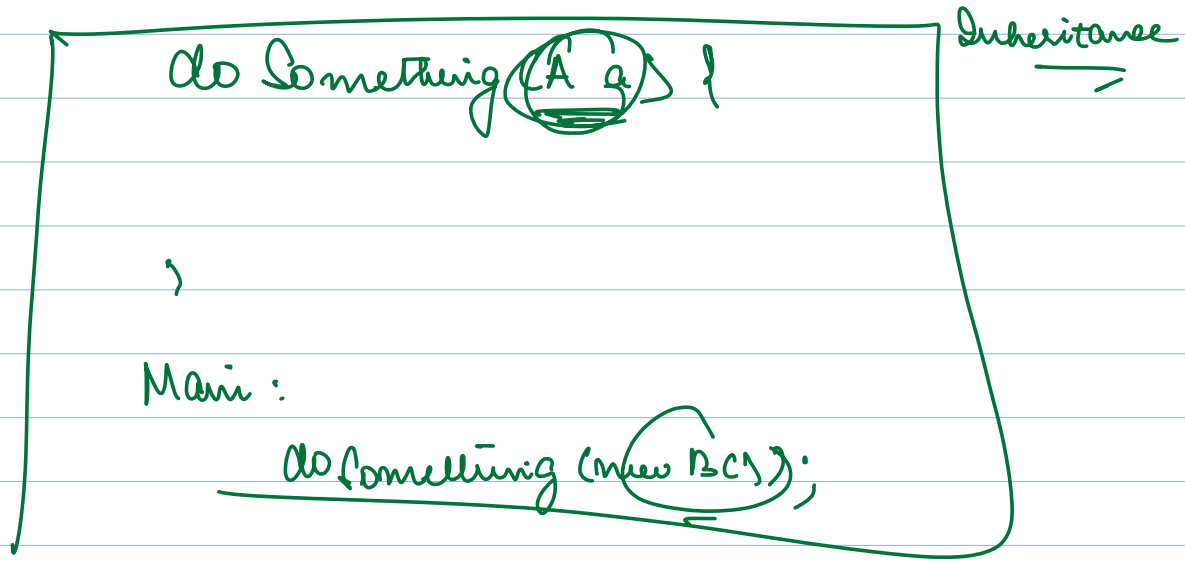


A a = new D();
a doS() → Hi

A a = new B()
a.doS(); → Hello

B b = new D() → Hi


A
hi(){
}
A
a = new C()
a. hey()
hey is not in A
B
hi(){}
hey(){}
C
hi()
hello
Compiler ⇒ data type of Variable
Runtime ⇒ data type of obj.

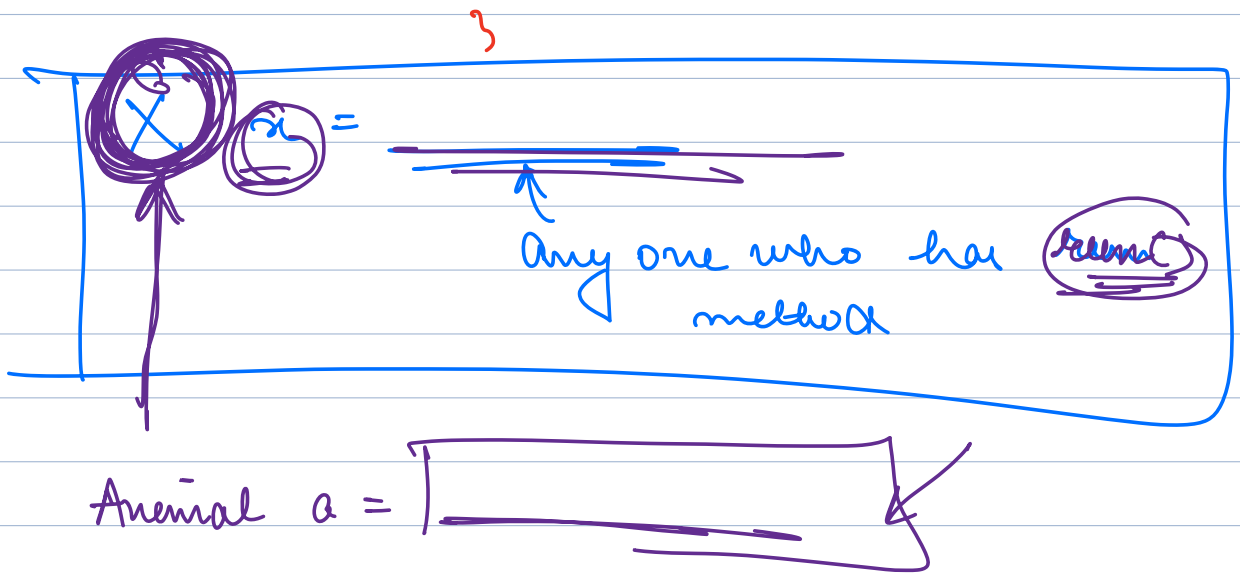

Inheritance
doSomething(A a){
}
Main:
doSomething(new B());

INTERFACES

Race

Object

race (List<> participants) {



Animal a =

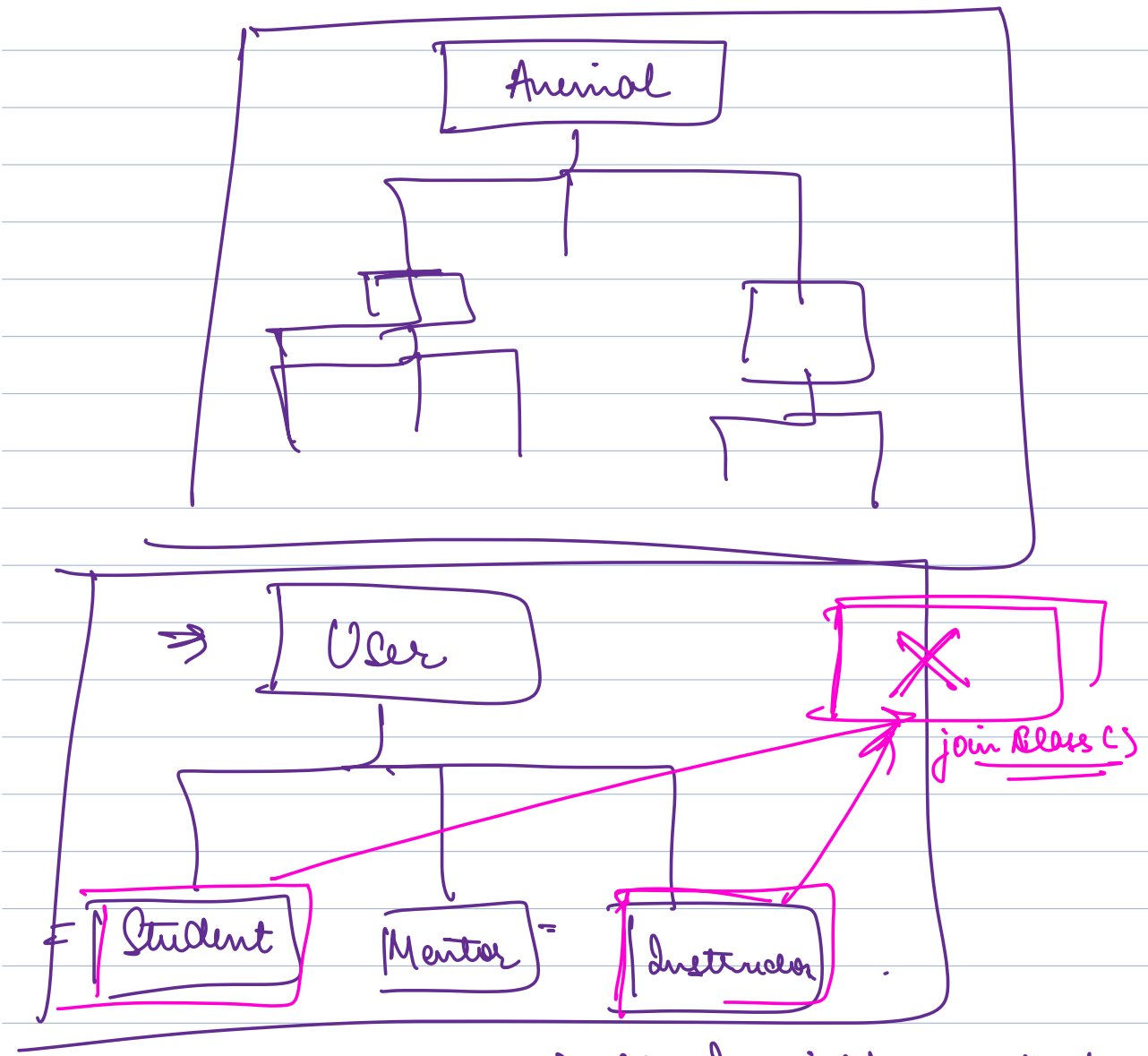
⇒ Sometimes there are situation when we group entities by behaviours they support instead of who they are.

class Animal {

attrs

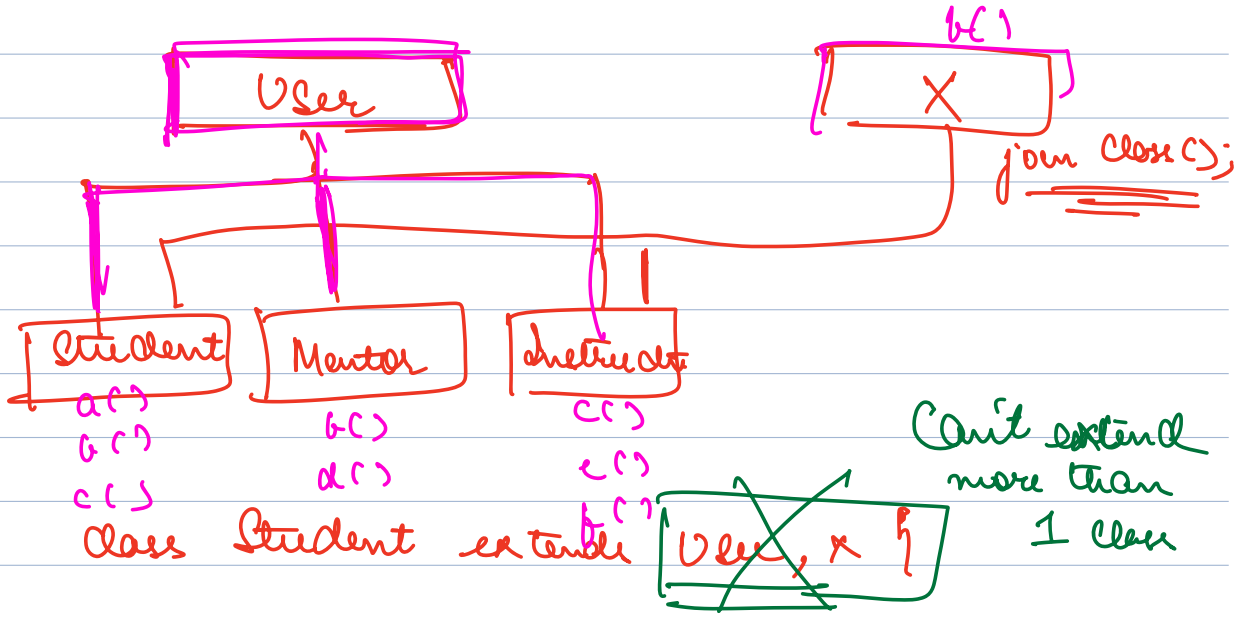
method

}



⇒ eg: I might want to club entities that can learn.

⇒ Clubs entities that can join something

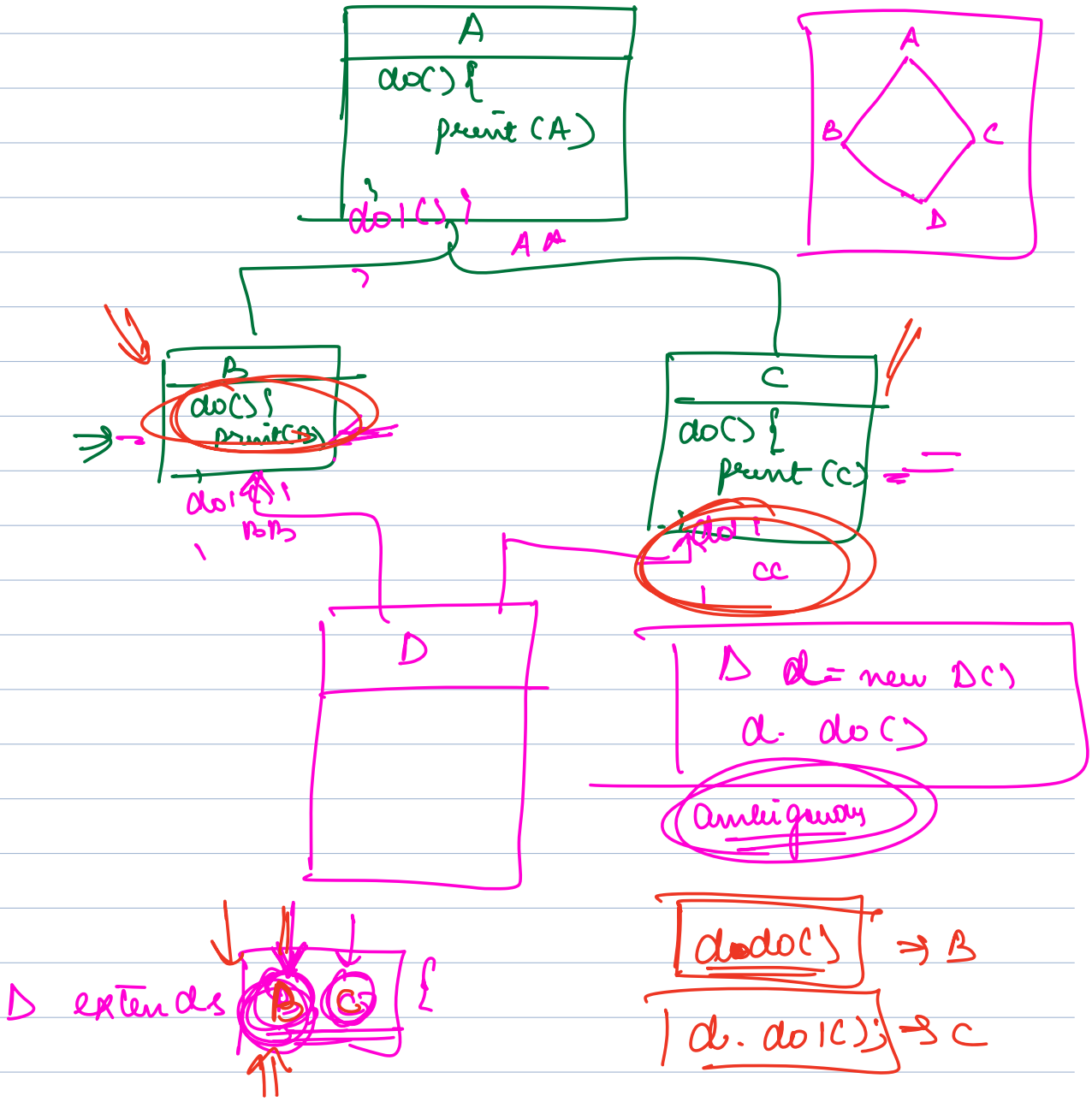


new Instructor
new Student()

X attendee = attendee.join Class

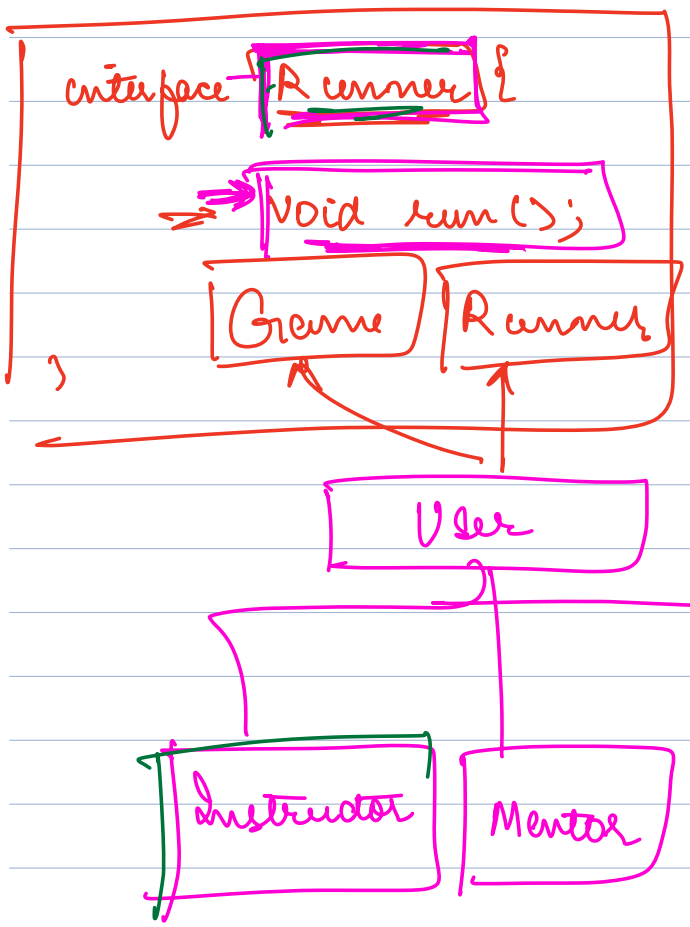
⇒ Most of prog lang (incl Java) don't support multiple inheritance.

Why? ⇒ Diamond Problem

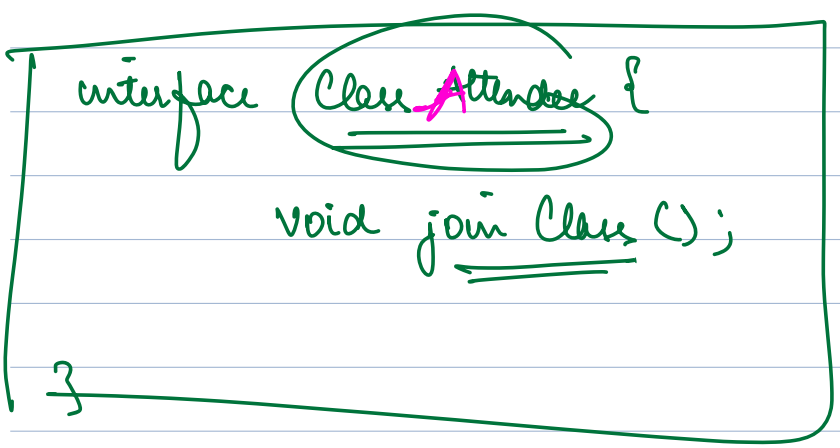

A
do(){
print(A)
}
do1(){
}
A A
A
B
C
D
B
do(){
print(B)
}
do1(){
}
B B
C
do(){
print(C)
}
do1(){
}
CC
D
D extends B, C {
D d = new D()
d. do()
Ambiguous
d.do()
⇒ B
d. do1()
⇒ C

INTERFACES

⇒ blueprint of a behaviour
⇒ stamp
⇒ when you want to classify ~~classes~~ based on behaviours they support



→ interface doesn't have any imp of methods
→ one class can imp any # of interfaces
→ can't create an object of interface



For classes that want to be identified as that interface:

1. implement that interface
2. implement all methods of the interface

```
Class Student extends User implements ClassAttendee {

    void joinClass() {

        cout ( Student joined Class );

    }

}

Class Instructor extends User implements ClassAttendee {

    void joinClass() {

        cout ( Instructor joined Class )

    }

}
```

Runner r; ⇒

```
ClassAttendee ca = new Instructor()
                   new Student()
                   new ClassAttendee()
```

Interface == Contract

implementing == I will have all methods as per the req of the interface.

Animal is anyone who can breathe, eat, make sound

```
interface Animal {
    void breathe();
    void eat();
    void makeSound();
}

class Human implements Animal {
    void breathe() {
        =
    }
    void eat() {
        =
    }
    void makeSound() {
        =
    }
}
```

```
interface Stack {
    void push(int x)
    int pop()
    int top()
}
```

```
Class LinkedListStack implements Stack {
    void push() { }
    int pop() }
    \
}
```

class ArrayStack imp Stack.

Class DoubleQueue

⇒ Stack s = new LinkedList Stack();

List l = new ArrayList

Map m = new HashMap();

⇒ Code to interface not an impl

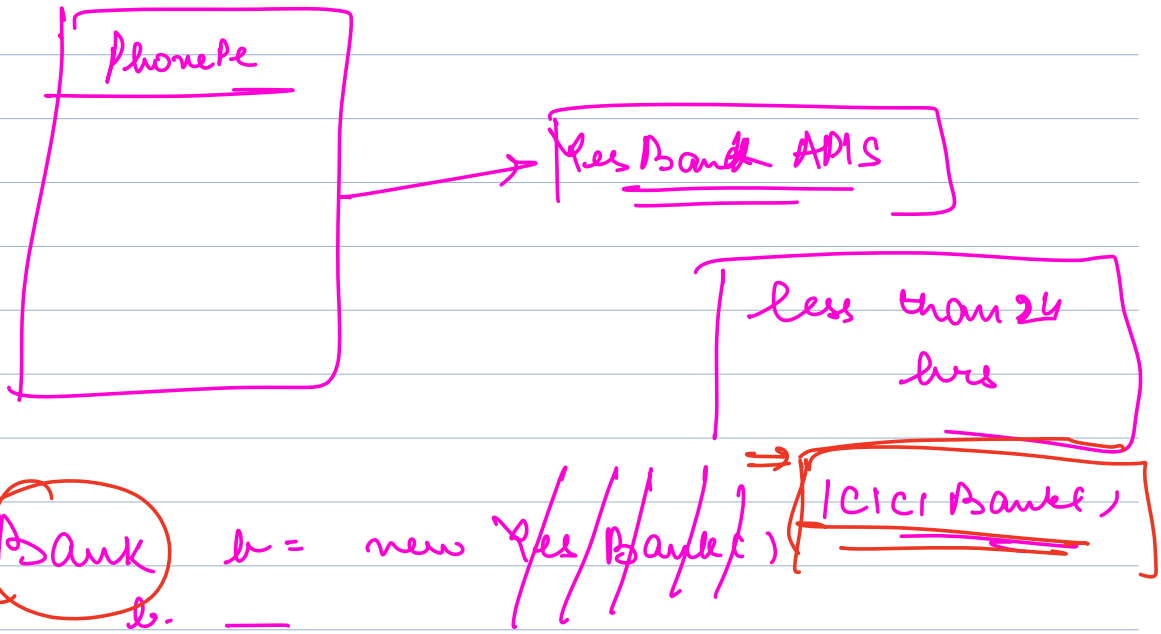
⇒ More maintainable and reusable

⇒ List l = new ~~ArrayList()~~ LinkedList();

l.add()
l.remove()

⇐ methods are of interface.

Yes Bank ⇒ RBI banned digital transaction

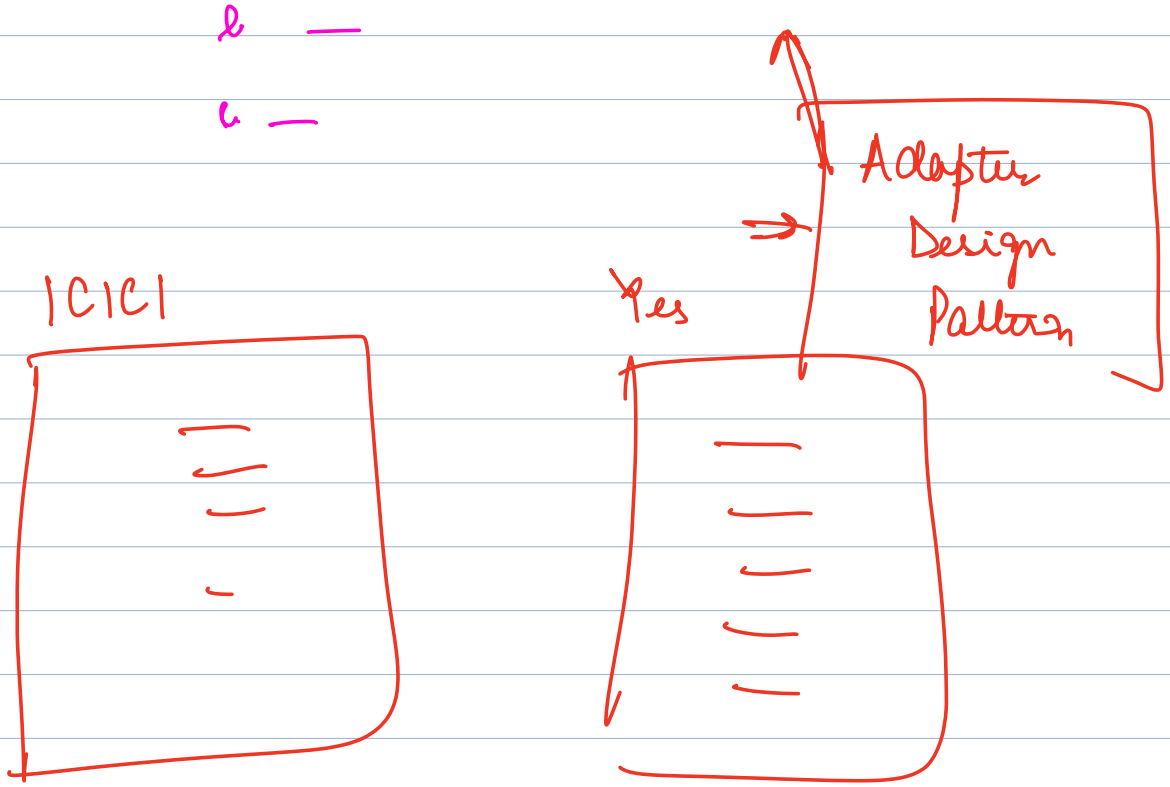


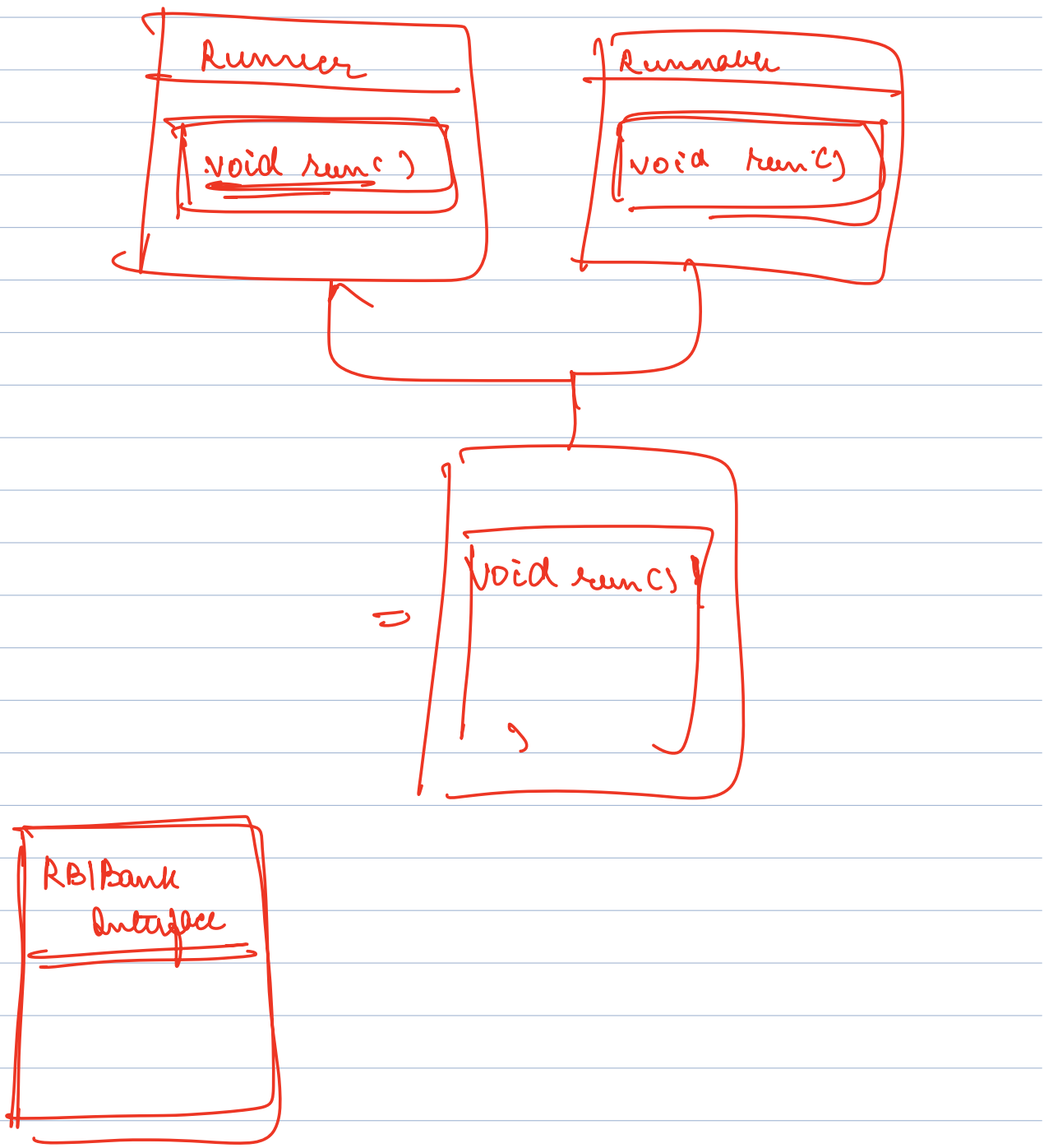
Bank b = new Yes Bank();

b. —

b —

b —



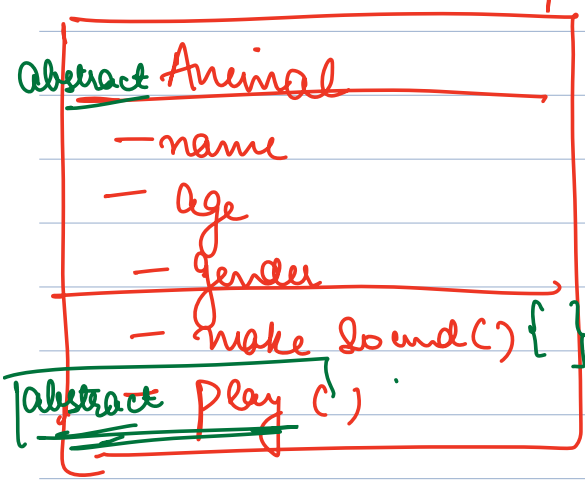

Runner
void run()
Runnable
void run()
=
void run()
{
}

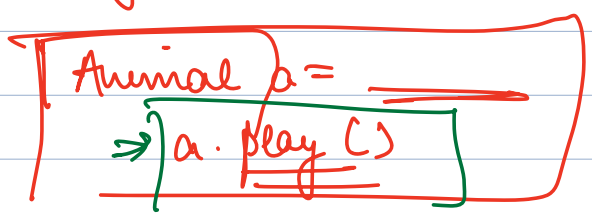
RBIBank
Interface

Runner r = new User()
new Student()
new Mentor()

Abstract Classes

↳ to rep an entity





→ As the entity is not completely known, I should not be allowed to create object of that

→ For such an entity, the methods that are not defined are declared abstract

→ As well as, the class is also declared Abstract

→ The child classes of that class must implement all abstract methods Or themselves should be declared abstract

abstract class Animal {

String name
int age
String gender.

void makeSound() {

Sout (Bla Bla Bla)

}

abstract void play();

}

abstract class Dog extends Animal {

void clean() {

}

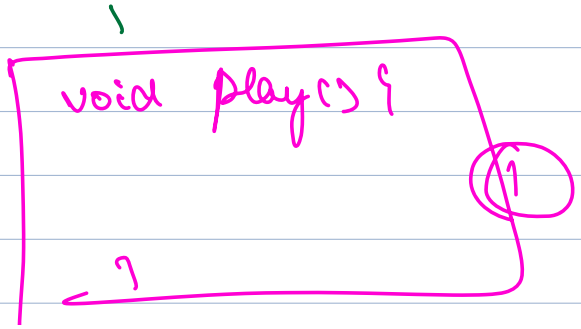
void barks() {

}

}

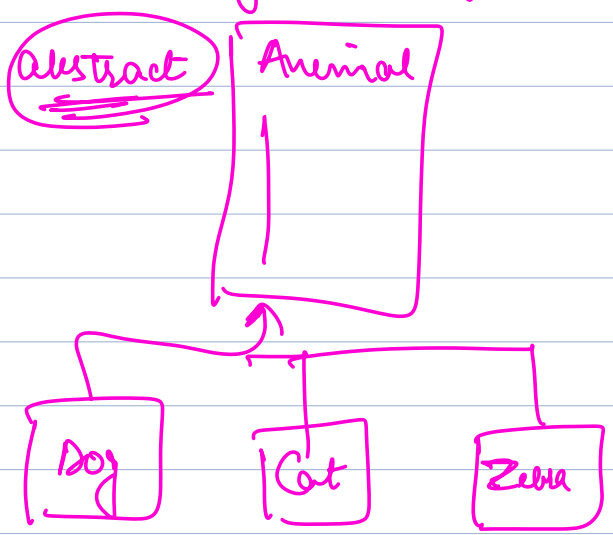
void play() {

}



→ Use an abstract class only if you are defining an entity

→ When you are completely defined but you don't want your obj to be created

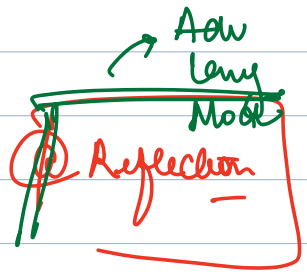


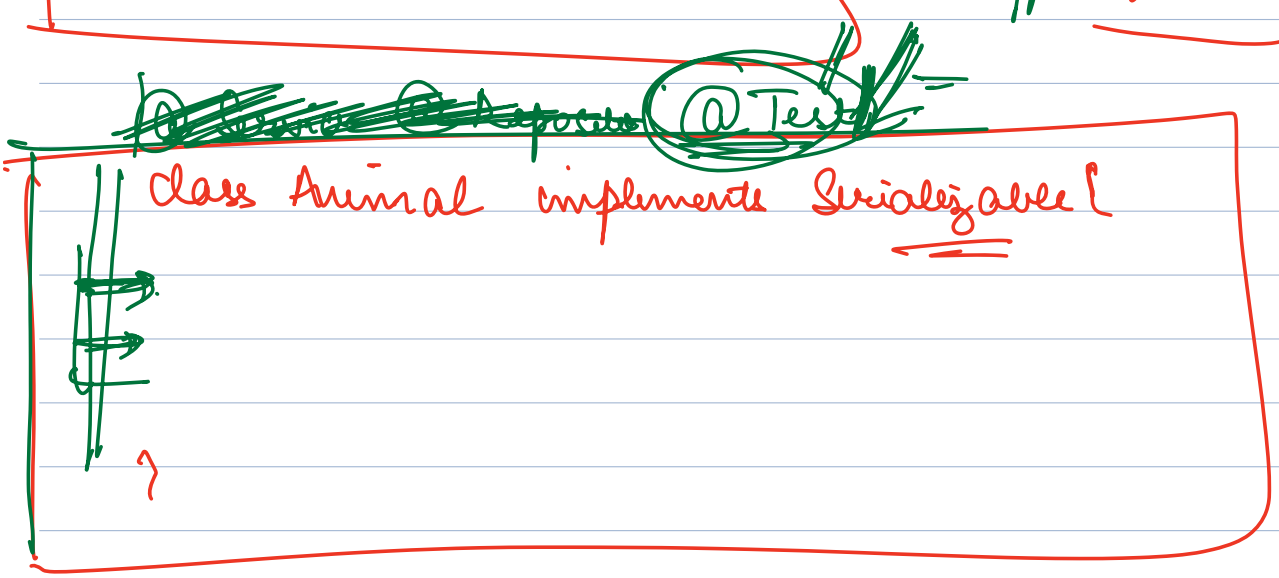
Abs Class → Not completely defined
or
you don't want to create objects of it

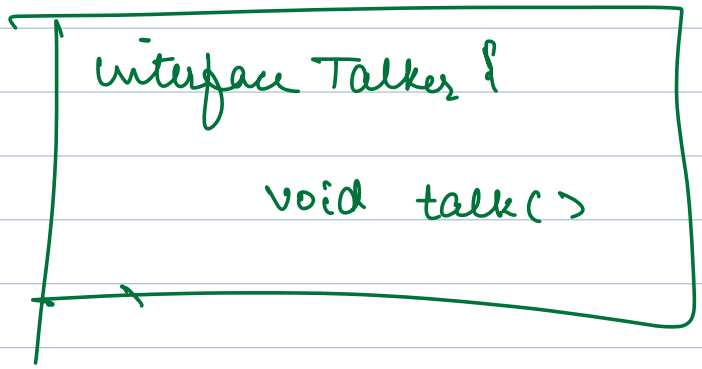
Abs Class	Interfaces
→ Entity (Something about what it is doing)	→ behaviour / contract
→ Attrs + behaviour	→ behaviours
→ Can extend 1 abs class	→ Can implement as many interfaces

Marker Interface ⇒ Interface which has no method

interface Serializable { }







implements Talky

List<Talker> talkers;

implements A, B, C, ... {

}

}

Spring framework

Object → JSON

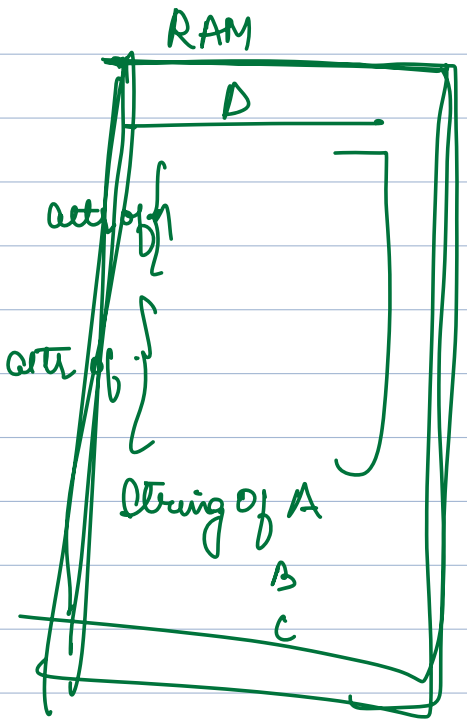
implement Serializable

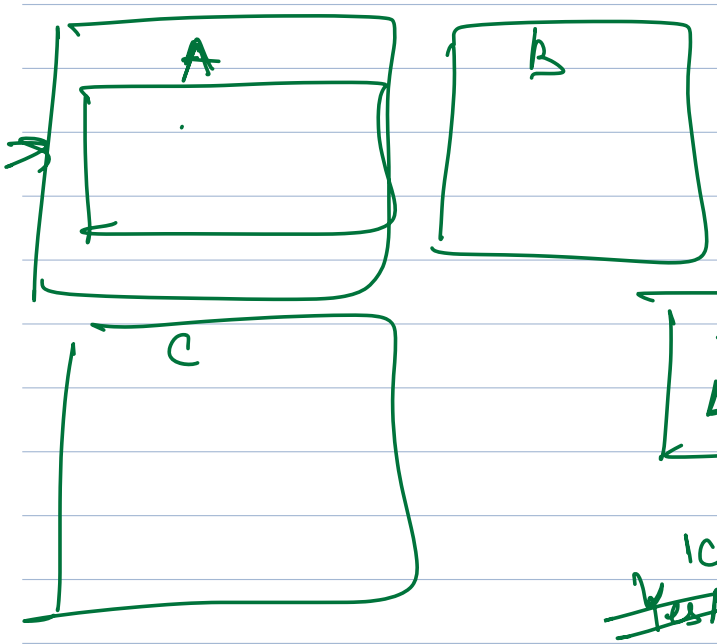
OOP
- Why
- Principle
- Pillars
- Interfaces

interface with only 1 method

SOLID → 1 design principle

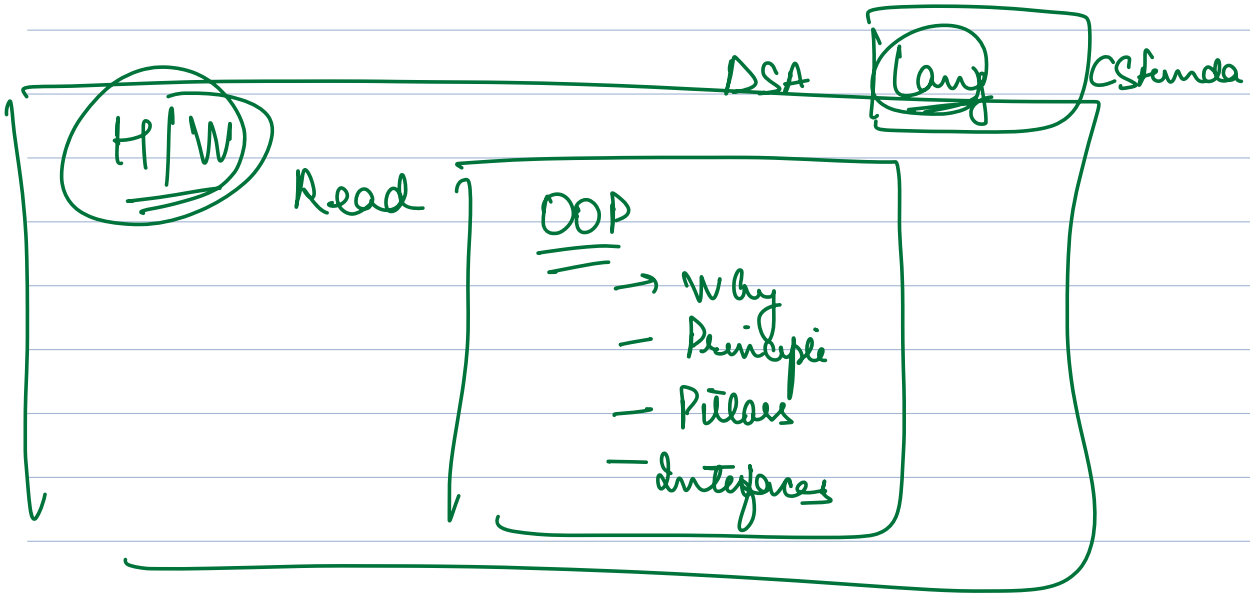
```
D d = new D()
d.doACS
```





Adapter Design Pattern

~~YesBank~~ ICICIBank y = new ~~Yes~~ICICIBank();


DSA
Lang
CSfunda
H/W
Read
OOP
→ Why
– Principle
– Pillars
– Interfaces