1. Good Evening
2. Lecture begins at 9:08 pm
3. Topic - Inheritance & Polymorphism

Agenda

1. Access Modifier 1 → Public, Private, Default
2. Method overloading (polymorphism 1)
3. Inheritance
   - Basics ✓
   - Cfon chaining (super) ✓
   - Access Modifier 2 → Protected
4. [Method over-riding (polymorphism 2)]
5. 3 Rules for questions around inheritance & polymorphism

Access Modifiers 1

- Public, Private, default

Encapsulation

- Capsule
  - putting the medicine together
  - protect the medicine from environment

Encapsulation in OOPs

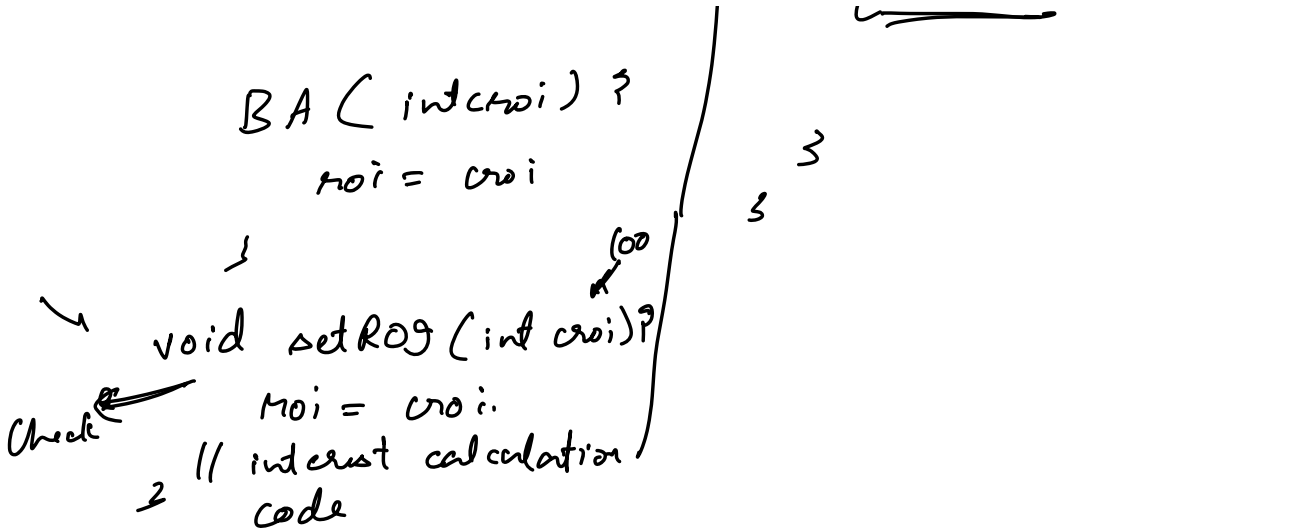
- Classes
  - Binding attributes & behaviour together
  - Protect the attributes from outside world

```
class Student {
    int age;
    String name;
}
```

```
class Client {
    main() {
        Student s = new Student();
        s.age = 10;
        s.name = "Sumedh";
    }
}
```

```
BankAccount {
    private int roi;
    private int interest;
}
```

```
Client {
    main() {
        BA a1 = new BA();
        a1.roi = 100;
```



private AM → Allows only the methods of class to change an attribute

public AM → Allows change from anywhere
{ same class, other class in same pkg, classes outside the pkg also }

default AM → pkg private
[AM not specified]
→ same class
→ other classes in same

pkg

X [→ classes outside the pkg]

Access Modifier → can be applied to methods even constructors

[A class with a private constructors is possible]

Singleton Design Pattern

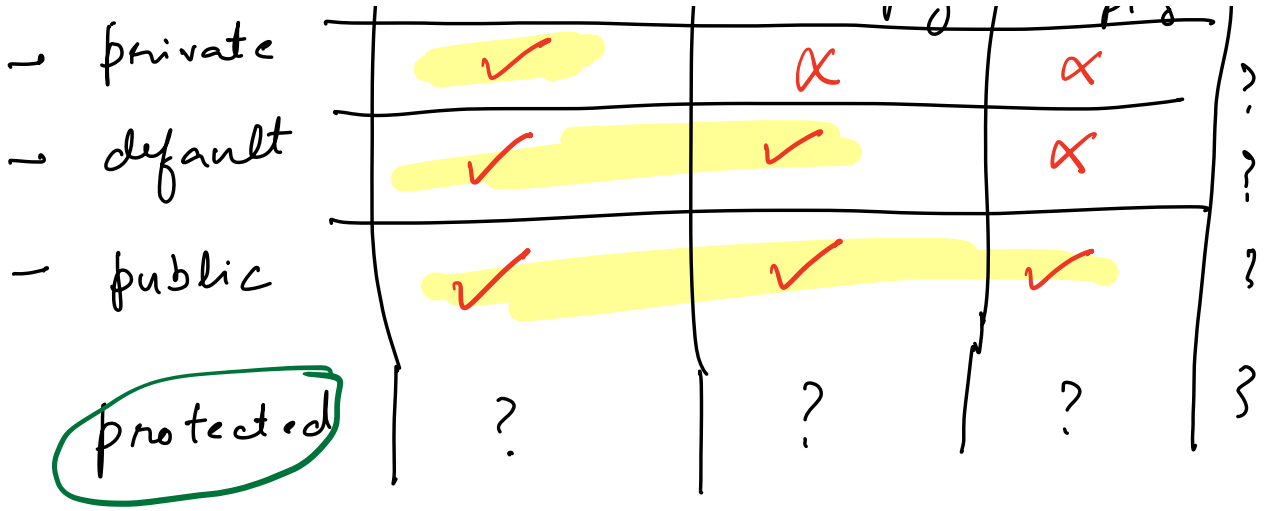
```
class A {
    private A() {
    }
}
```

```
class Client {
    main() {
        A o = new A();
    }
}
```

Why to make a cton private?

→ To control the number of objects that can be created

Same class	Class in same pkg	Class outside pkg



Polymorphism | [Method Overloading]

Poly - Morph - ism

Poly → Many or Multiple

Morph → Form

Many Forms

↳ Same code can take many forms

Polymorphism → Compile time [Method overloading]

Polymorphism → Run time [Method overriding]

```
class A {
    void fun() {
        syso("Hello world");
    }
    void fun(String name) {
        syso("Hello" + name);
    }
}
```

Method overloading

Having more than one fn. with same name, but different signatures.

☆ → What is signature?

→ How will these be called?

→ [Is the decision to call the fn. made at compile time or run-time?

✡ → Very important tricky interview question.

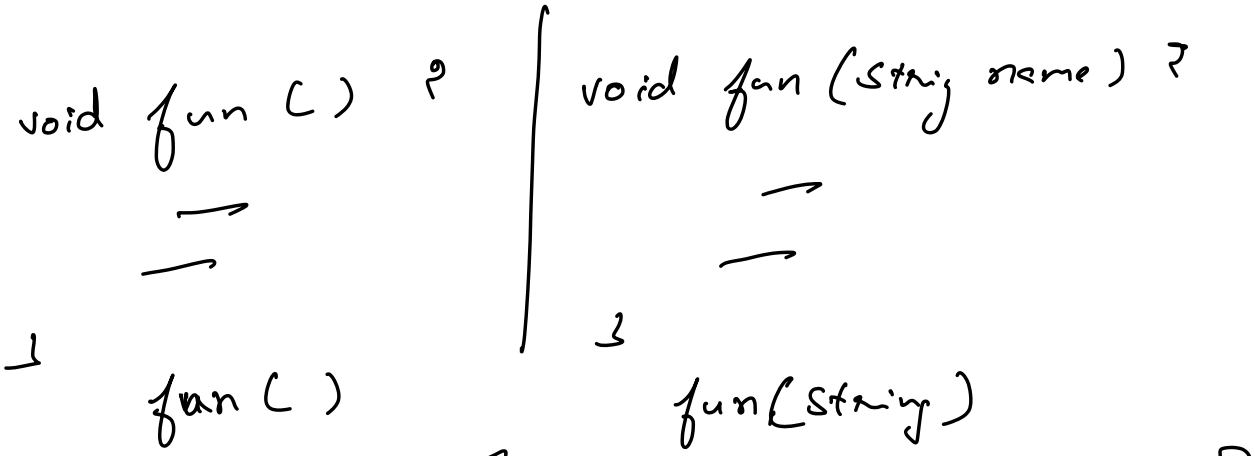
```
class Client {
    main ()
        A obj = new A()
        → obj.fun();  ✓
        → obj.fun("Universe"); ✓
}
```

Decision to call a method from a set of overloaded methods is made by the compiler, based on the signature.

↓

Compile-time

✡ [Compile-time or static binding or early binding
Polymorphism]



Signature → [fn. name & data-types of parameter list]

☆ The names of the parameter are not a part of signature.

☆ [Return type] is also not a part of signature.

→ Two fns. in same class cannot have same signature

✗
```
void fun ( String s, int i ) ;
void fun ( String x, int j ) ;
```

→ Name of params is not a part

of signature

```
void fun (String s)
String fun (String s)
```

Return type is not a part of signature.

Array.sort(↙ int[], byte[], short[] / String[] ↓)

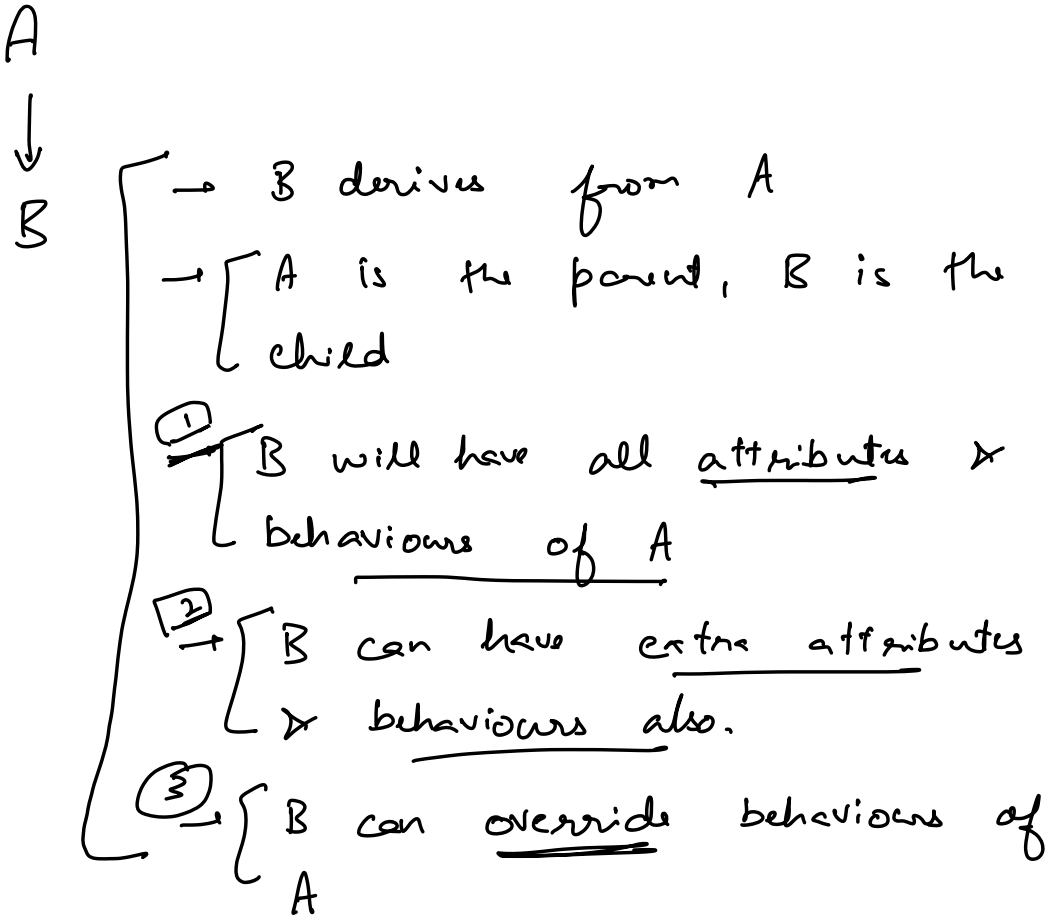
Arrays.sort have overloaded implementations.

Break → 10:00 – 10:08

- Inheritance
- Polymorphism 2

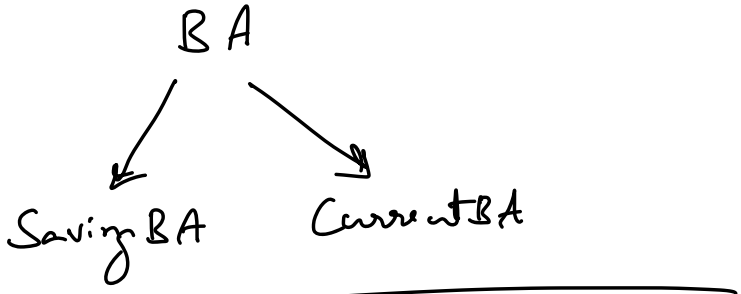
Inheritance

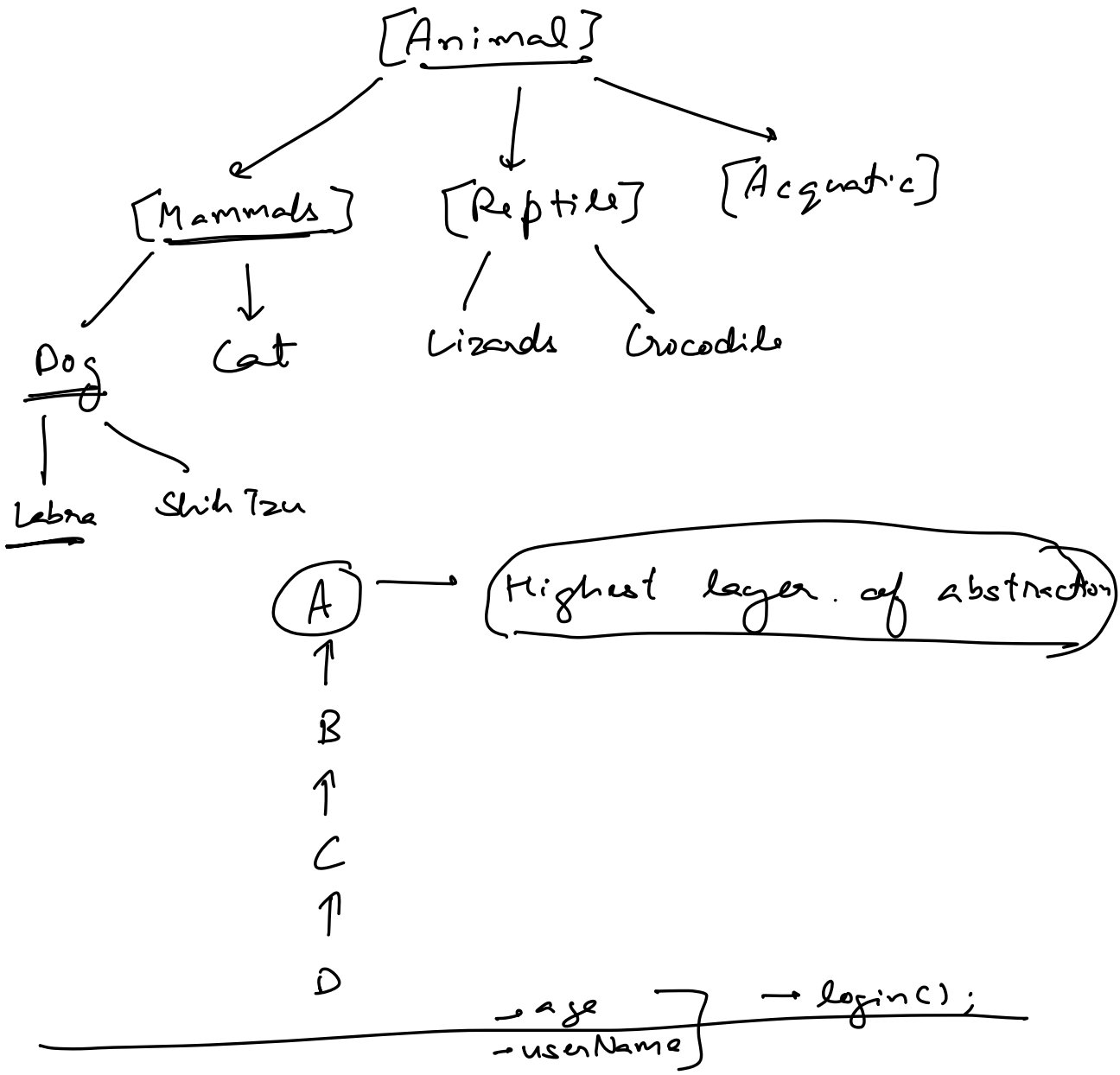
↳ Relationships in hierarchy

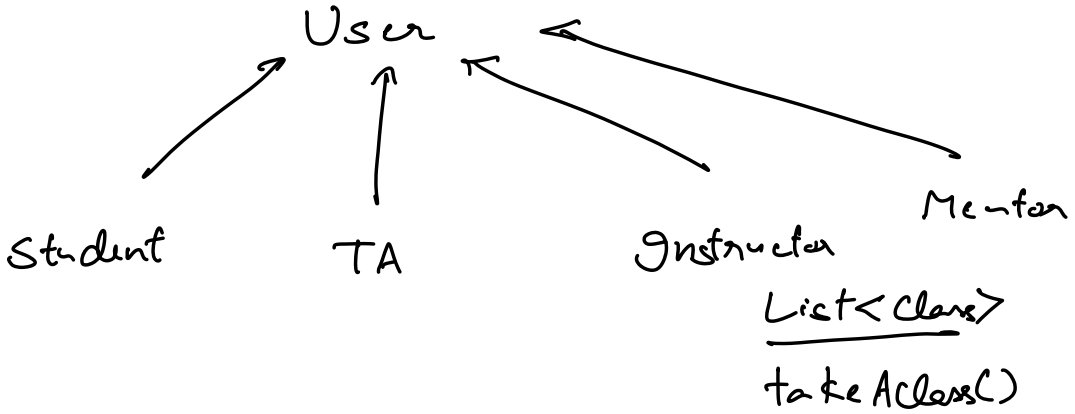


Out of A & B which one is more general & which one is more specific ?

Child classes are more specialized than parent classes





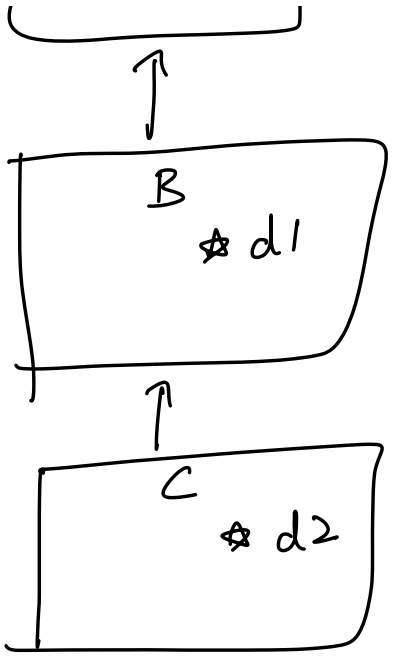


IS-A : Inheritance allows us to implement "is-a" relation

- Labrador is a Dog
- Dog is a Mammal
- Mammal is a Animal

Constructor Chaining

- A
  - d



B has d1

C has d, d1 & d2

If we make an object of C()

→ Who should initialize d2? →

C's const → " " " d1?

B's const → " " " d?

A's const

C obj = new C()

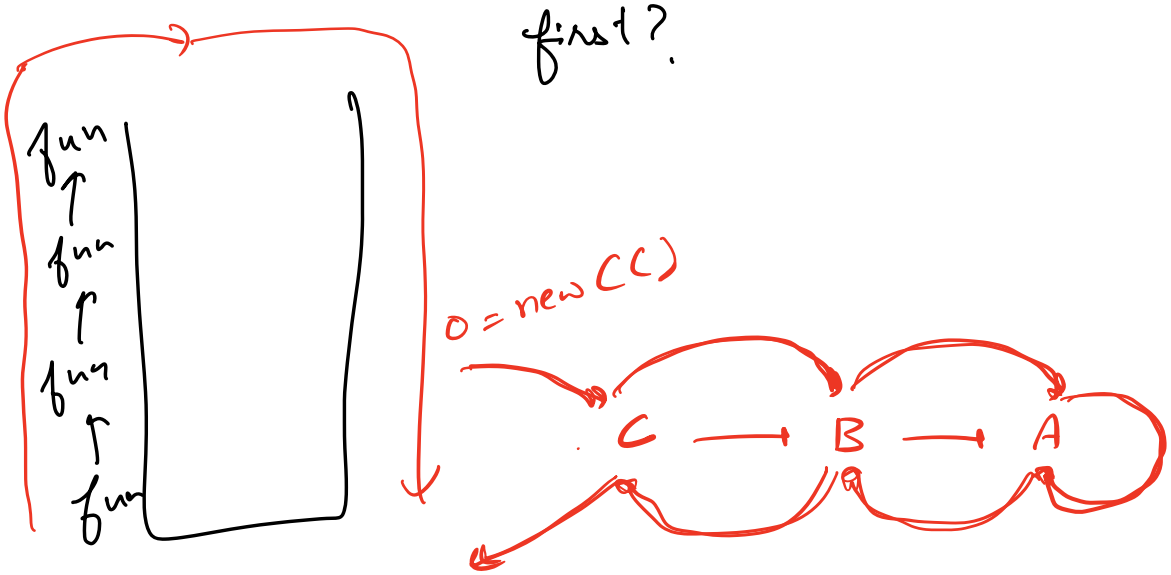
We are calling only C's constructor.

C's ctor will invoke B's ctor.

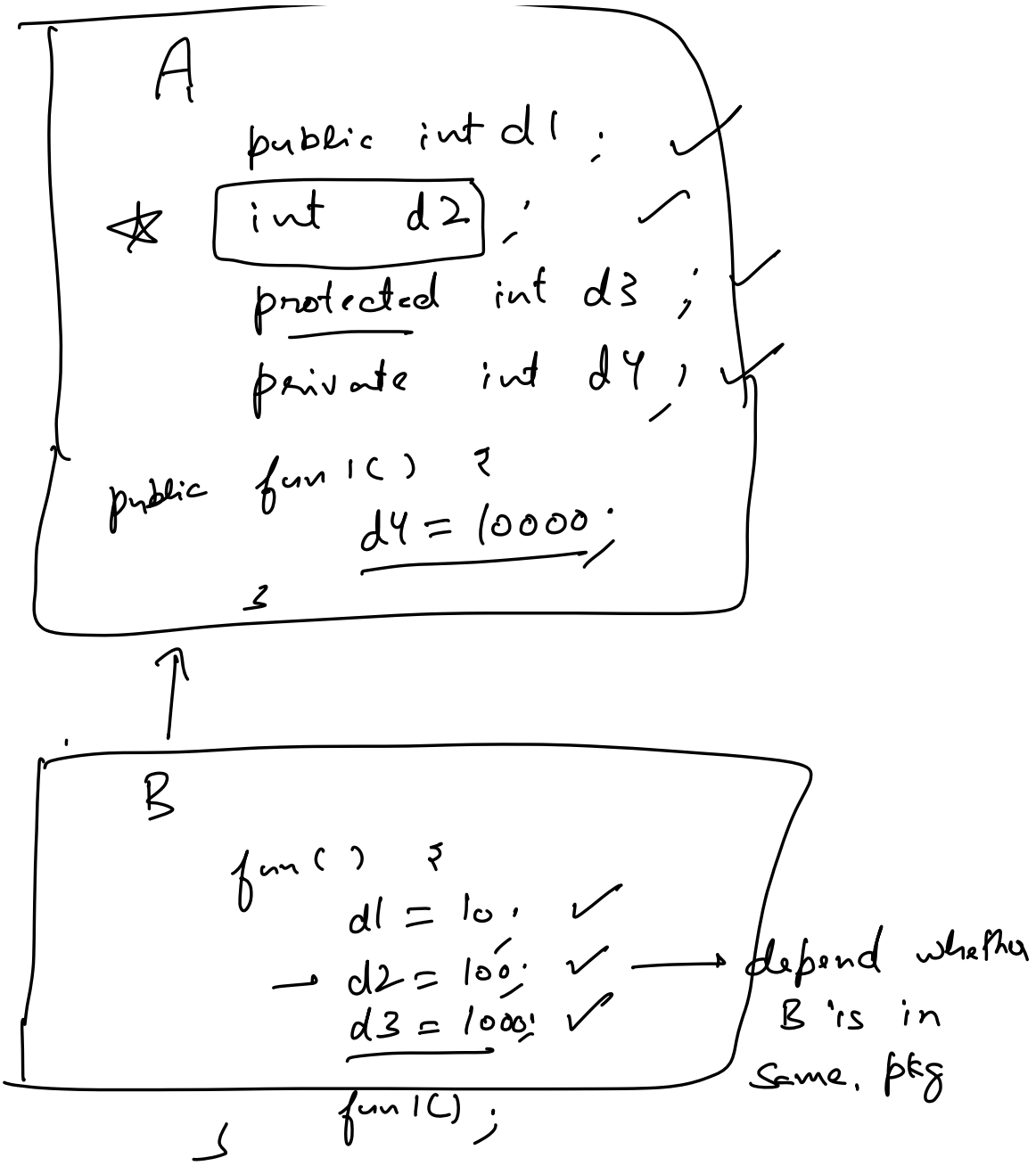
B's ctor will invoke A's ctor

A
↑
B
↑
C

1. Whose ctor will be called first?
2. Whose ctor will finish first?



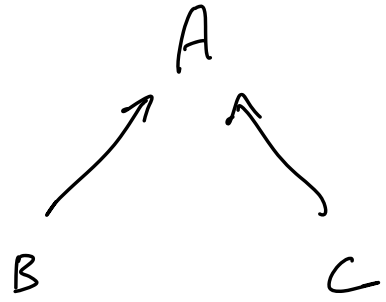
Inheritance : Protected AM



	Same class	Normal class, same pkg	Derived class, same pkg	DC, other pkg	NC, other pkg
Private	✓	✗	✗	✗	✗
default	✓	✓	✓	✗	✗
Protected	✓	✓	✓	✓	✗

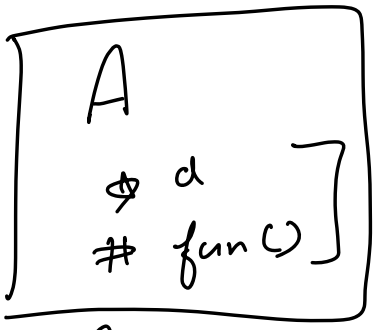
Public	✓	✓	✓	✓	✓

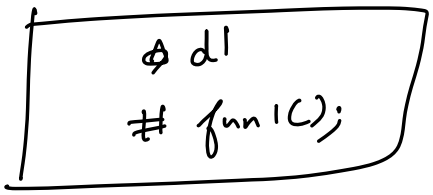
Polymorphism in Inheritance




```
A a = new A()      B b = new B()
a = new B()        b = new A()   X
a = new C()        C c = new C()
                   c = new A()   X
```

```
A a = new A()      B b = new B()
  Or                 Or
A a = new B()      B b = new A()   X
```





A a = new A()

a.d ✓

a.fun ✓

A a = new B();

a.d ✓

a.fun() ✓

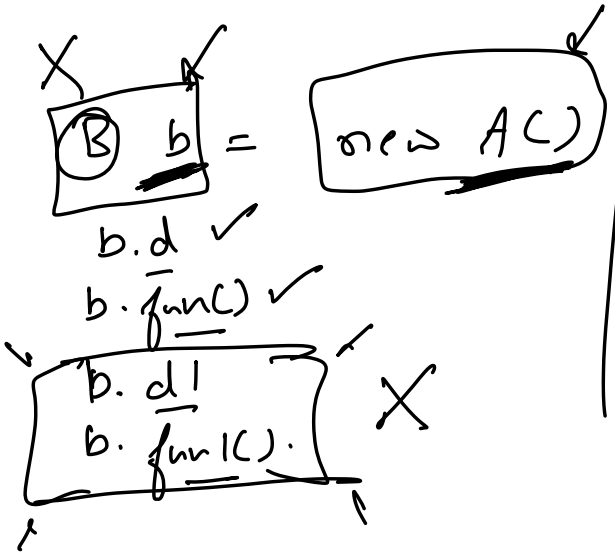
B b = new B()

b.d ✓

b.d1 ✓

b.fun ✓

b.fun1() ✓



A a = new A() ✓

A a = new B() ✓

B b = new B() ✓

B b = new A() ✗

→ Compile time error

```
A a = new A()
A a = new B()
```
→ Polymorphism

```
List<A> list = { new A(),
                 new B(),
                 new C(),
                 new D() }
```

A
↑
B
↑
C
↑
D

Method - overriding ✓

A
↑
B

or

run-time polymorphism

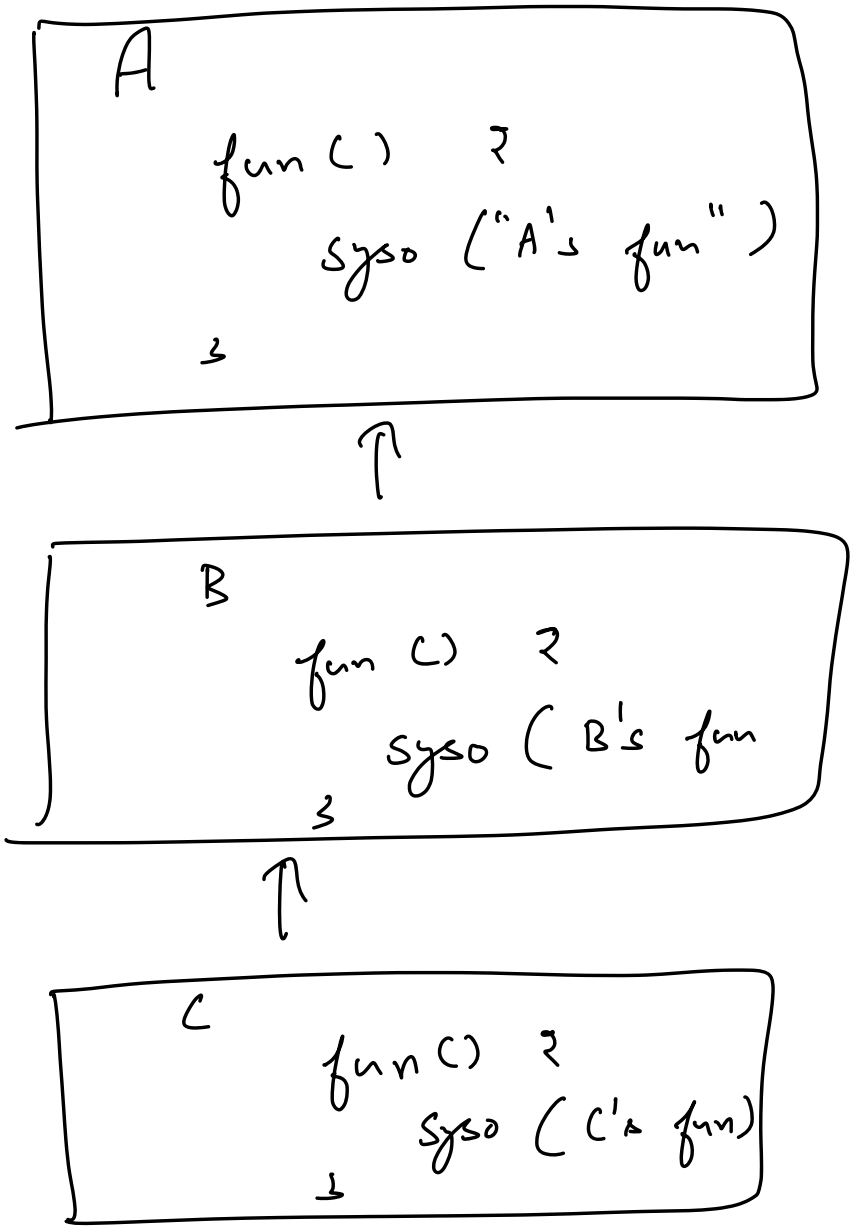
or

dynamic binding

or

late binding

1. everything that A has
2. can add more attrs & behaviours.
3. can override behaviours of A



```
A a = new A()
a.fun();  ⟶ A's fun
```

```
a = new B()  ✓
a.fun();  ⟶ B's fun
```

```
x a = new C();
a.fun()  ⟶ C's fun
```

```
Client {

    main() {
        A a* = getAnObject()
        a.fun()
    }
    A getAnObject() {
        if (uselect == 1) {
            return new A(); ✓
        }
        else if (uselect == 2) {
            return new B(); ✓
        }
```

```
{
    C ?
        return new C();
}
```

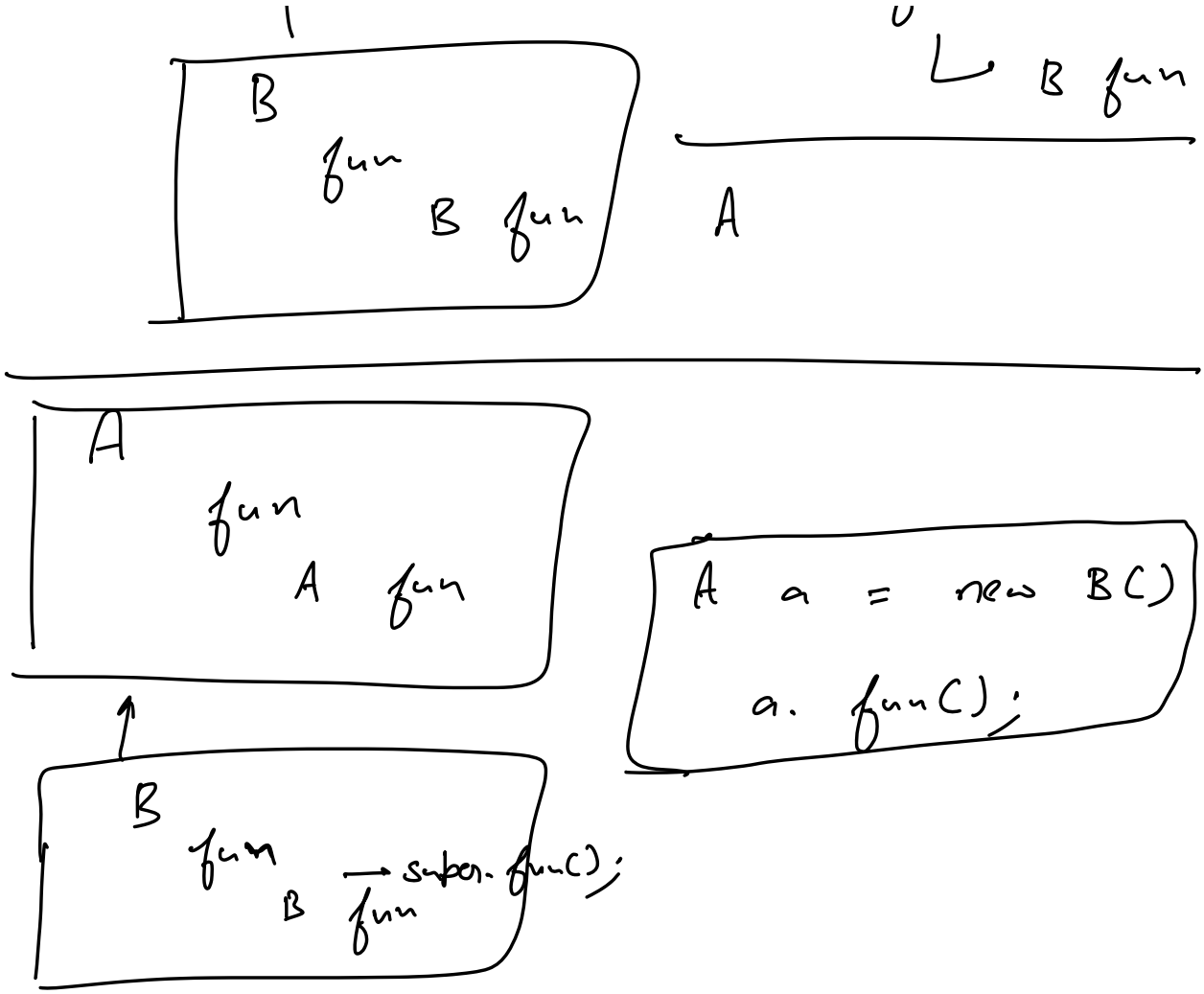
The decision to call the fn. is not made at compile-time.

It is made when program is running & JVM decides which fn. to call based on the instance [not ref] of object

```
A a = new B();
a.fun();
```

```
A
    fun
        A fun
```

```
A a = new B()
a.fun()
```

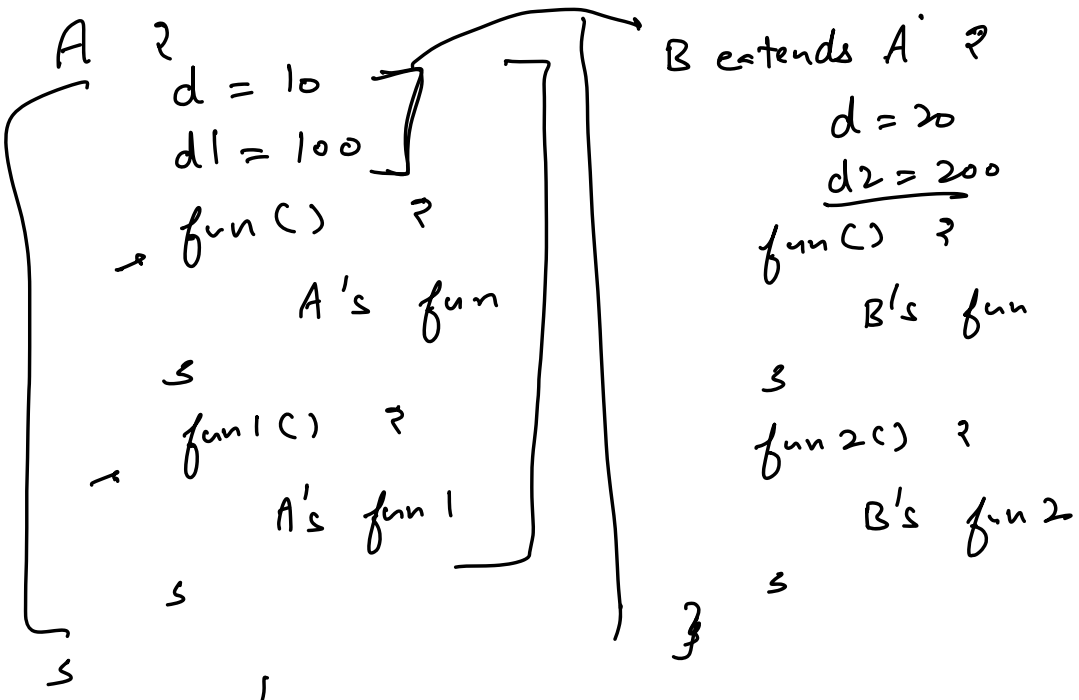


Memory Diagrams ▷ Rules

R1 – Compiler sees the reference on LHS, & JVM sees the instance on RHS

R2 → Compile-time decisions are made by compiler, JVM makes run-time decisions.

R3 → On conflict JVM will do what reference says [applies to data-members]



Case 1

A a = new A();

sout(a.d) ✓ 10

sout(a.d1) ✓ 100

sout(a.d2) ✗ CTE

a.fun() A's fun

a.fun1(); A's f1

a.fun2(); CTE

Case 2

A a = new B() ✓

sout(a.d) ✓ (Rule 3 → 10)

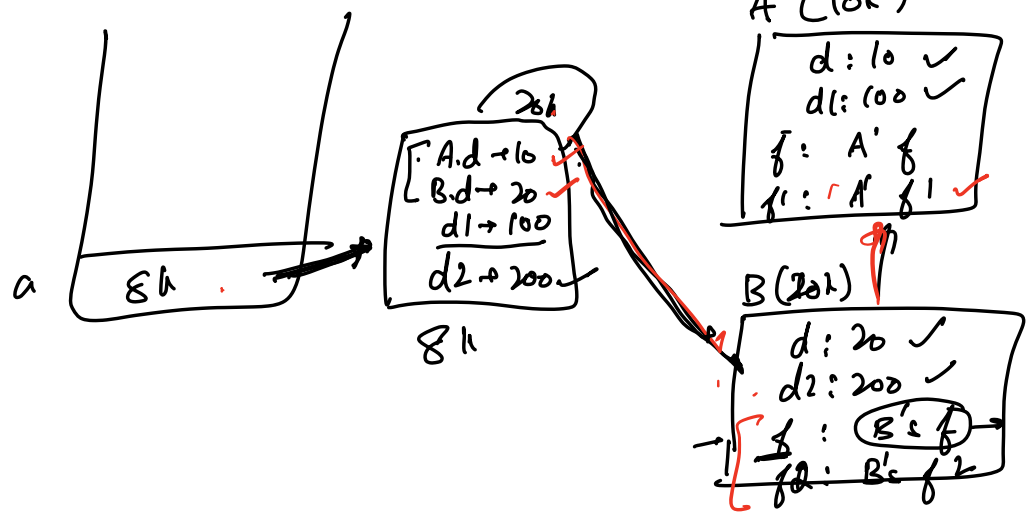
sout(a.d1) ✓ 100

sout(a.d2) CTE

a.func() B's func.

a.f1() A's f1

a.f2() CTE



Case 4

B o = new A();

sout(o.d) ✓

sout(o.d1) ✓

Not relevant

this doesn't compile

sout (o.d2) ✓

o.fun() ✓

o.fun1() ✓

o.fun2() ✓

Case 3

B o = new B()

sout (o.d) ✓ [R3 : 20]

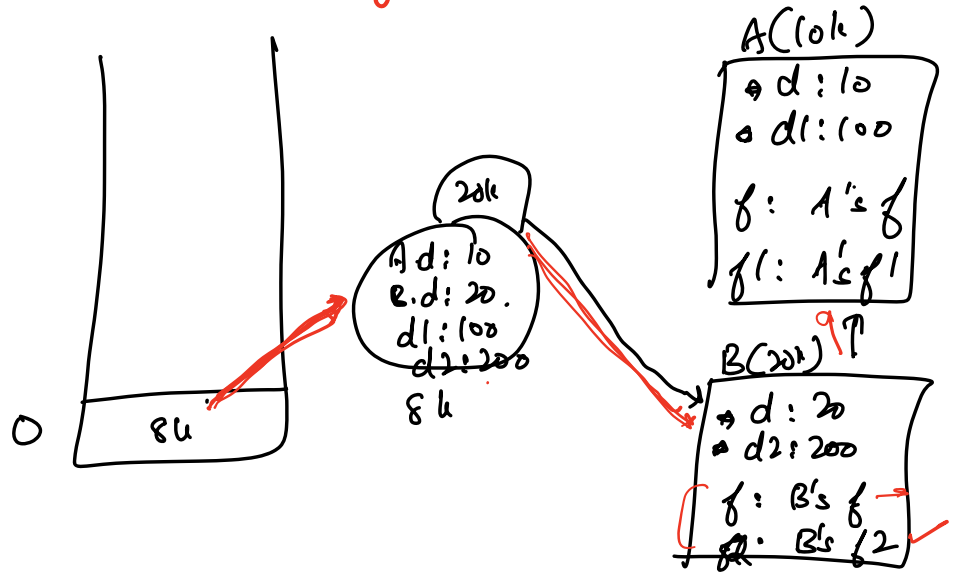
sout (o.d1) ✓ 100

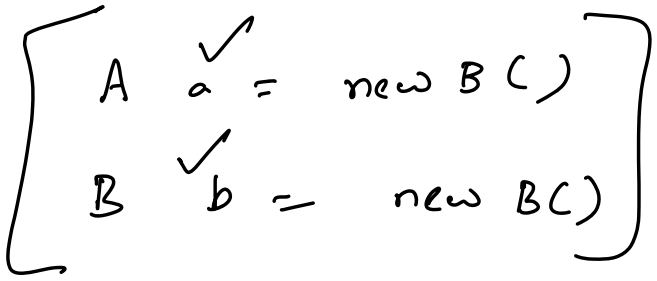
sout (o.d2) ✓ 200

o.f() ✓ B's f

o.f1() ✓ d's f1

o.f2() ✓ B's f2




A a = new B ()
B b = new B()
a.d → 10
b.d → 20

1. Good Evening
2. Lecture begins at 9:08 pm
3. Topic - Static, interfaces, abstract classes.

Agenda

1. static → data members
   - methods
3. Interfaces ✓
4. Abstract → method
   - classes
2. Final → method
   - class
   - data member
   - variable
5. Interfaces vs Abstract classes.

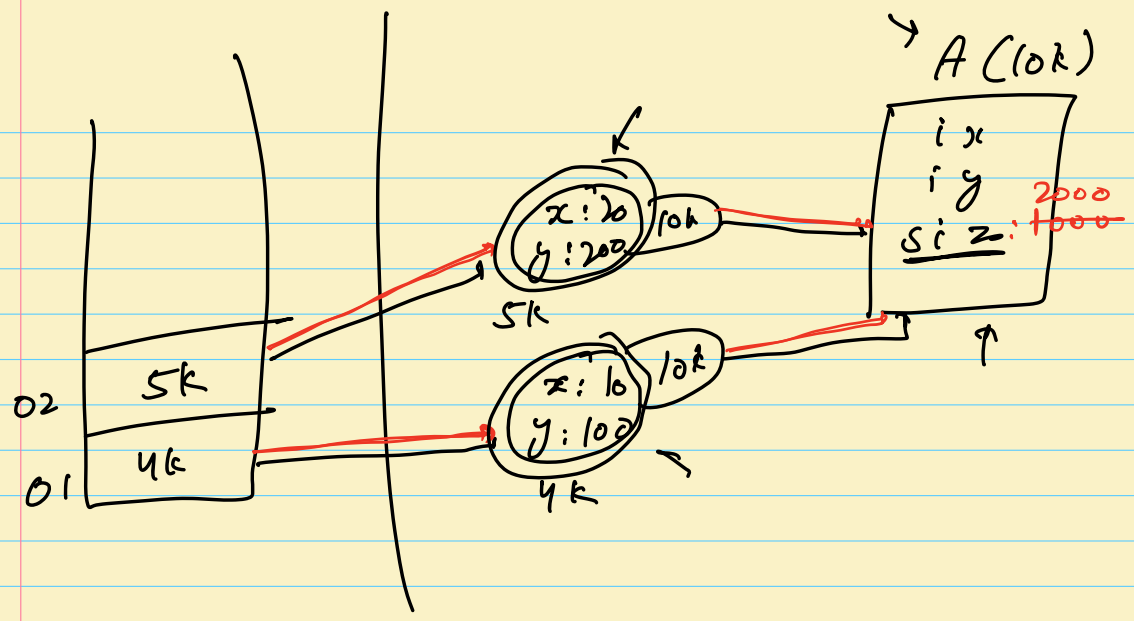
Static
- data members ✓
- methods
- class [not today]

Static data-members

```
class A {
    int x;
    int y;
    static int z;   ☆
}
```

Static data-members don't take memory in objects, instead they take memory in class

```
A o1 = new A()  ✓
A o2 = new A()  ✓
```



01. x = 10

02. x = 20

01. y = 100

02. y = 200

[[01. z = 1000]
[02. z = 2000] ✓]

Warning

Preferable

A. z = 3000;

[Static data-members are shared by all instances of a class.

↳ Saves memory

e.g. class Student {

String name;

int rollNo;

static String schoolName;

}

☆ We need not create an instance for accessing data-members.

A.z = 3000; { o1 & o2 are not necessary }

o1.z = 1000; { Warning: static data-member should be accessed via class name }

class Point {
int x;
int y;
Point(xx, yy) {
x = xx; y = yy;
}
}

class Rectangle {
static Point origin = new Point(0,0);
}

Static methods

↳ To call static methods it is not necessary to create an object

```
class Math {
    int abs (int x) {
        if (x < 0) {
            return -1 * x;
        } else {
            return x;
        }
    }
}
```

```
Math m = new Math();
int y = m.abs(-10)
```

Not reqd. Not preferred.

```
int y = Math.abs(-10);
```

Call via class name.

Common use of static methods

└ Utility fns.

Math.abs(-10)

Demo : Math, Arrays.sort

Static DM → They live in class & not objects.

Static methods → Can be called without making objects [directly using class name

```
class A {
    int x;
    static int y;
    void fun() {

    }
```

```
static void sfun() {
    ☆ →
}
}
```

Q1. Can static methods use non-static data members?

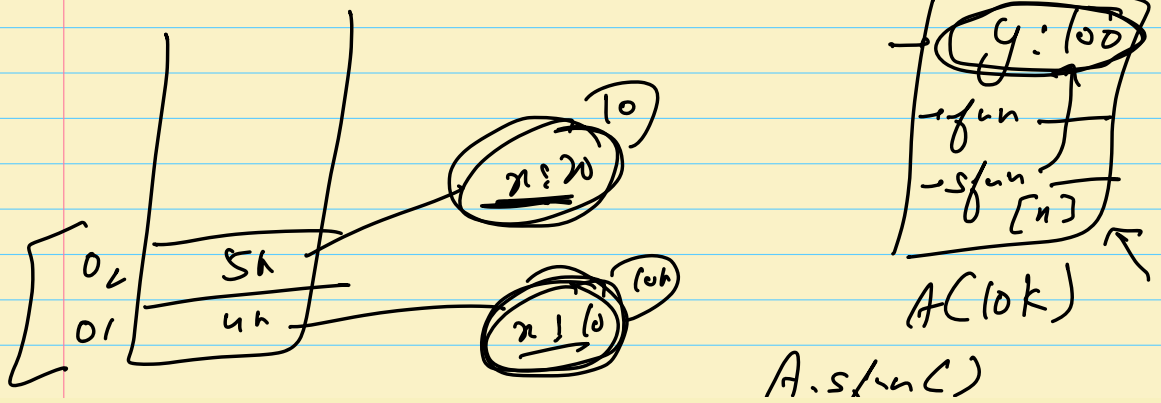
Can we use 'x' in 'sfun'?

No.

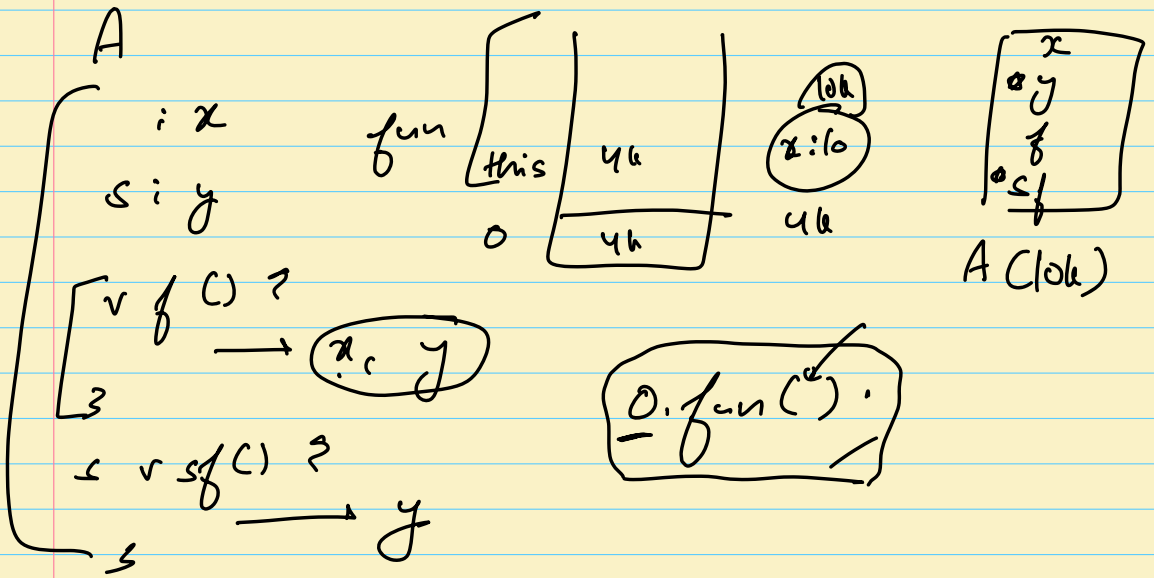
A.sfun()

↓

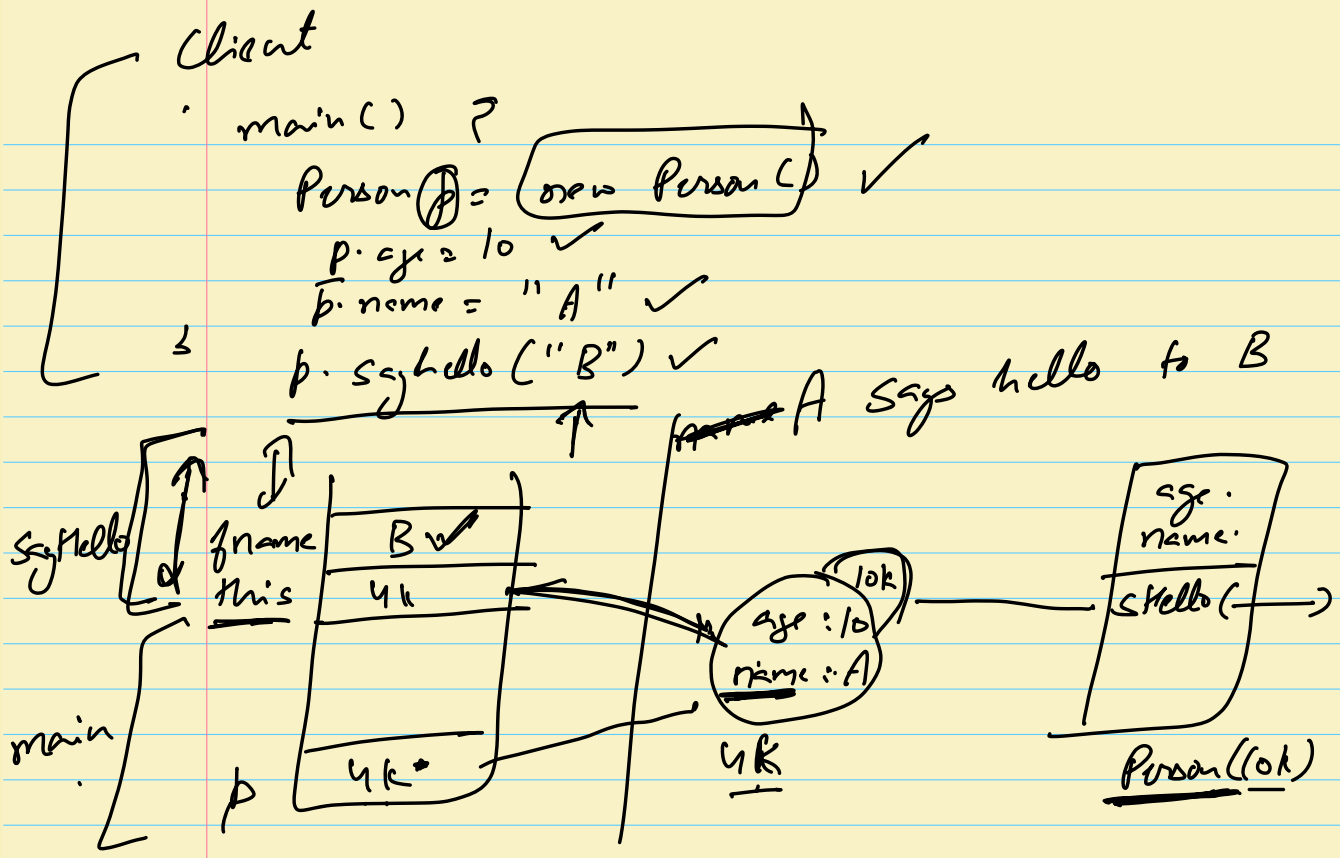
static methods can be called without creating objects. If we are using a non-static data-member there is a problem ∵ they live in objects & we have not created any object.



Q2. Can non-static methods use static data-members?




```
class Person {
    int age
    String name
    v sayHello(String fname) {
        sout(name + " says Hello to " + fname);
    }
}
```



Q3. Can static methods use "this"

↳ No

✱ Static methods can't use	Non-static methods can use
non-static dm, this	[static as well as non-static dm] [also this]

class A {

int x;

int y;

void fun1() {

x ✓ this ✓

y ✓

}

static void fun2() {

this ✗

x ✗ y ✓

}

}

Client

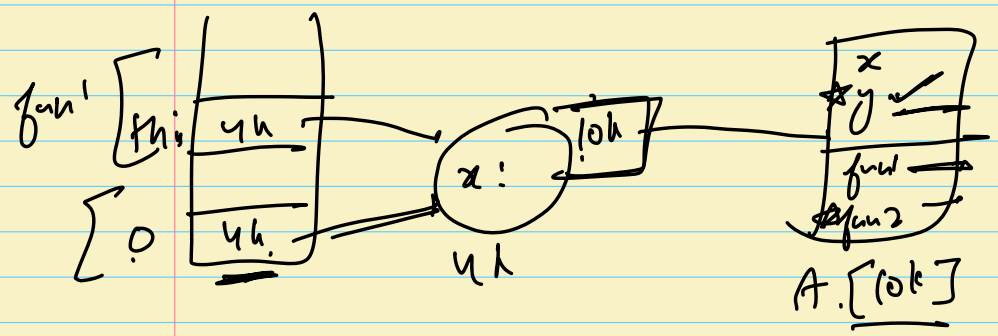
main() {

A o = new A()

o.fun1()

A.fun2();

}



- Can static methods be overloaded?

  Yes, Array.sort

- Can static methods be over-ridden?

```
class A {
    static void fun() {

    }
}

class B extends A {

    }

}

B.fun()
```

- for static methods decision is made at compile time based on class name, hence overriding is not possible.

* Overriding happens when fn. to call is decided by JVM, at run-time, based on the instance

Break → 10:23 to 10:31

Interface, Abstract, Final

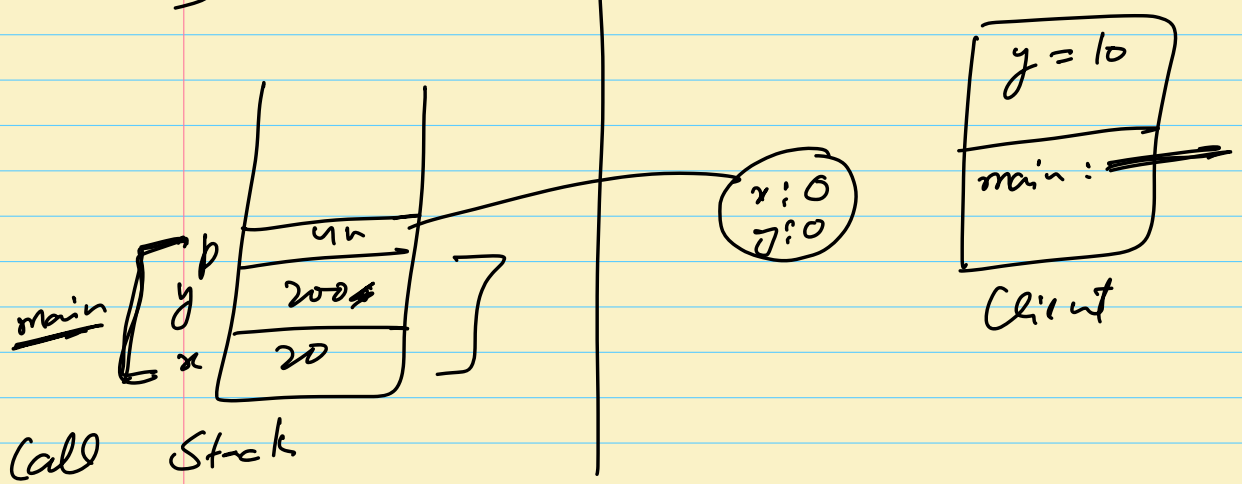
class Client {

  static int y = 10]dm

  static void main () {
    int x = 20
    int y = 200
    Point p = new Point(0,0)
  }

  static void fun() {
    int z = 20
  }

}



---

final keyword

- Variable
  - data-member
- method
- class

final as a variable

```
class Client {

    s v main() {
        final int x = 10;
        x = 20;
    }
}
```

Can't change value of x { it is a final variable }

---

final as a data Member

```
class Person {
    int age
    String name
    static final String species = "Homo Sapiens";
    Person() {
    }
}
```

---

final as a method

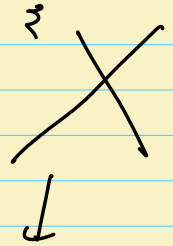
```
class Person {
    final void fun() {
    }
}
```

class VIP extends Person {

void fun() {

}

}



final methods can't be overridden

---

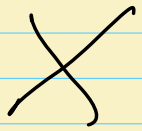
final class Person {

. . .

}

class VIP extends Person {

}



→ A final class can't be derived from.

final → Variables / data-members → can't be changed

final methods → can't be over-ridden

final class → " " extended.

```
class Point {
    int x
    int y
}
```

```
client {
    main() {
        final Point p = new
                Point(0,0);
        p.x = 10; ✓
        p.y = 20; ✓
        p = null;
        p = new Point(-,-)
    }
}
```

Interfaces : Contract of functions

- An Object oriented way to set the API of a category of entities.

```
interface Comparable<Car> {
    int compareTo(Car other);
}
```

class Car implements Comparable<Car>?

int compareTo(Car other)?

{

}

* Interface defines a contract of fns.
  [only declarations, no bodies]

* Any class, if it implements the interface, has to provide definitions for fns. provided in the interface

→ [Loose Coupling]

→ [Code to an interface > not an implementation]

Story!

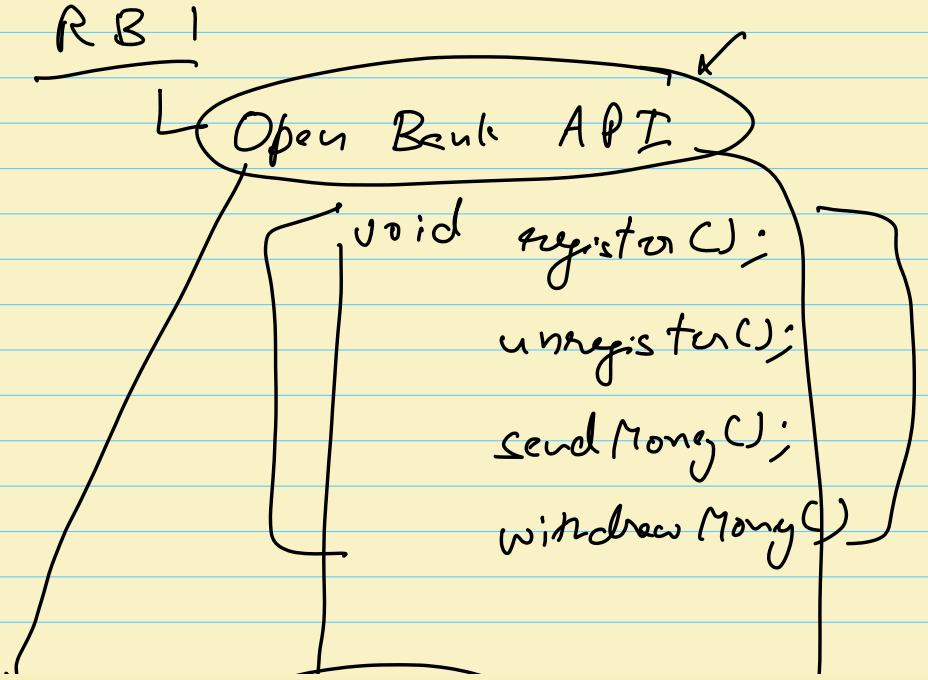
Real Life Example of Loose Coupling

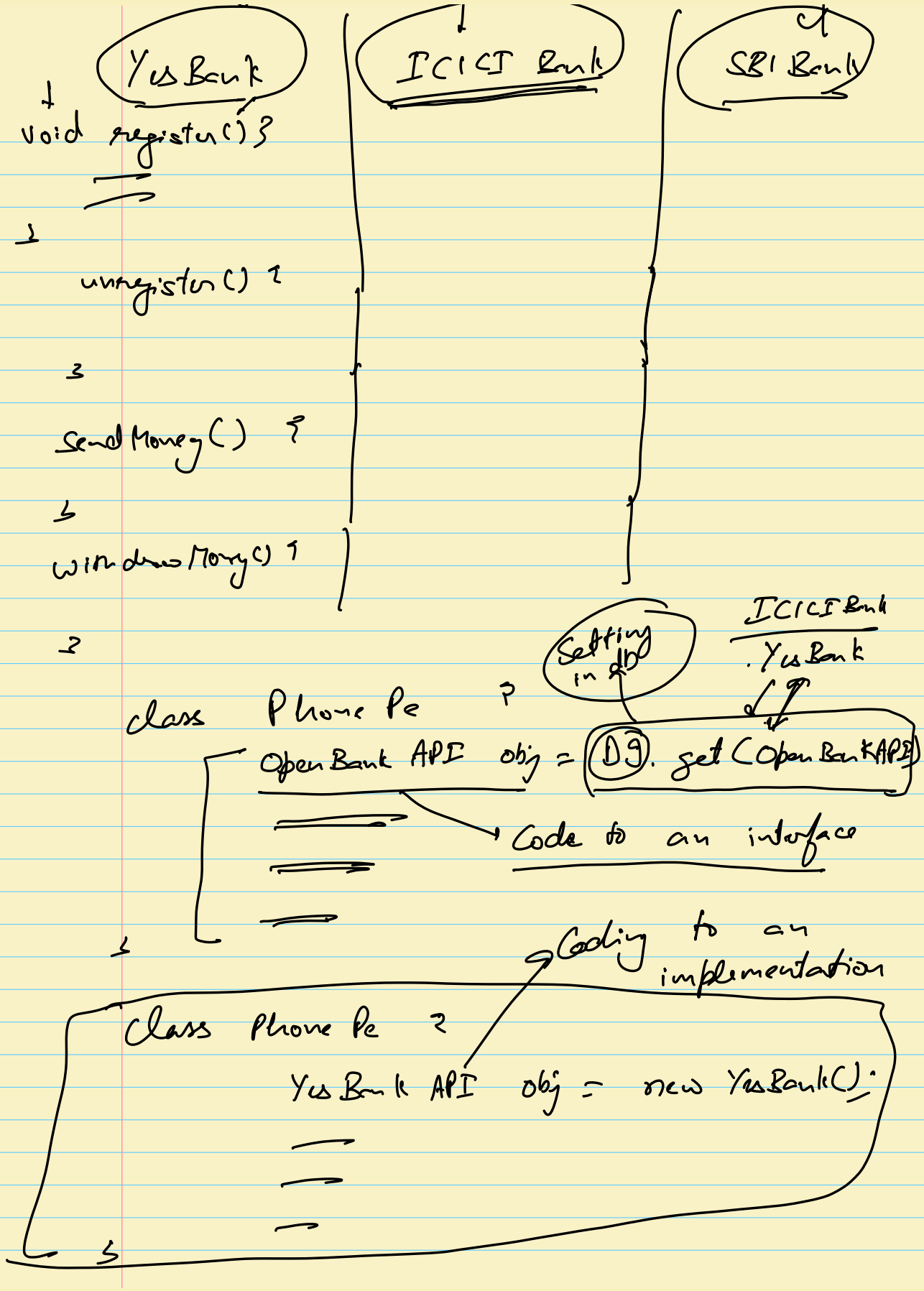
→ Sockets in walls allow us to plug any device with a compatible plug.

✡ USB / Type C interfaces in computer allow loose coupling of many devices with a compatible wire.

Story 2

PhonePe ⟶ Yes Bank




Yes Bank
ICICI Bank
SBI Bank
void register() {
unregister() ;
sendMoney() ;
withdrawMoney() ;
}
class PhonePe {
OpenBankAPI obj = DB.getConnection();
Setting in DB
ICICIBank
YesBank
Code to an interface
Coding to an implementation
class PhonePe {
YesBankAPI obj = new YesBank();
}

Story 3]

PriorityQueue<Integer>

PriorityQueue<Car>

class Car {
    int price;
    int speed;
}

List<Car> cars = ____

Collections.sort(cars);

sort is not able to compare the cars to decide which car is smaller & which is bigger.

We have to provide a way to enable Collections.sort to compare the cars.

. T implements Comparable<T>

Collections.sort (List<T> list) {

```
for ( i=list(); i > 0 ; i-- ) {
    for ( j=0; j<i ; j++) {
        if ( list.get(j) > list.get(j+1) ) {
            swap (list, j, j+1)
        }
    }
}
```

X

```
if ( l.get(j) > l.get(j+1) )
```

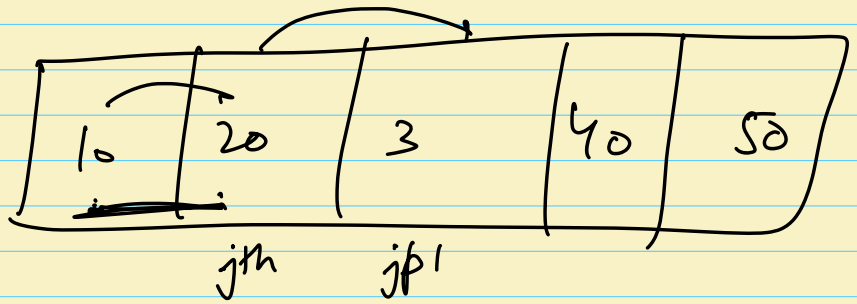
```
Comparable<T> jth item = (Comparable<T>) l.get(j)
Comparable<T> jp1 item = (Comparable<T>) l.get(j+1)
if ( jth.compareTo ( jp1 ) >= 0 ) {
```

compareTo method defined in Car declared in Comparable.

}

```
int compareTo(Car o) {
    return t.speed - o.speed
}
```

-ve	0	+ve
this < other	this == other	this > other
jth < jp1	jth == jp1	jth > jp1



Q1. What are interfaces?

↳ Contract of fns.

Q2. When a class implements an interface?

↳ Class must provide body for fns. declared in interface.

Q3. Collections.sort (List<T> list)

└ what interface does it depend on?

└ Comparable.

Q4. Collection.sort needs to depend on Comparable so it won't get List<Integer>

List<Car>, List<Person> can be passed to it.

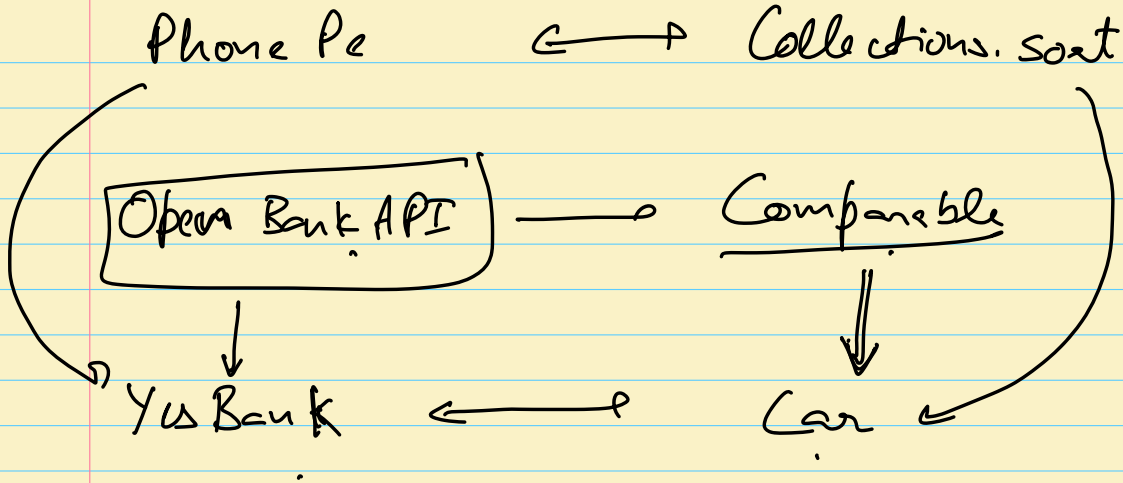
Q5. Car implement Comparable<Car>

int compareTo(Car other) {

return this.sp − other.speed

}

+ve	0	−ve
this > other	this == other	this < other
j > jpl	j == jpl	j < jpl

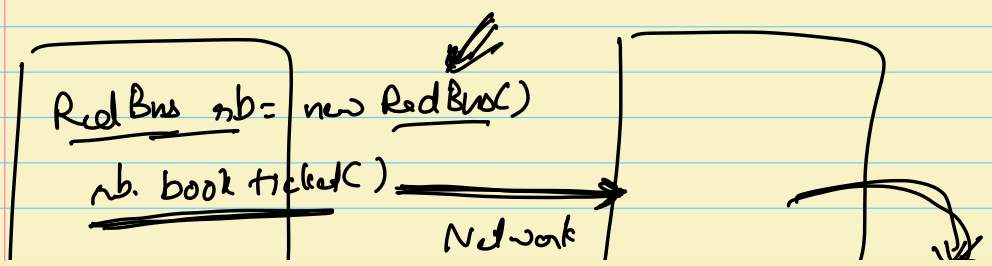


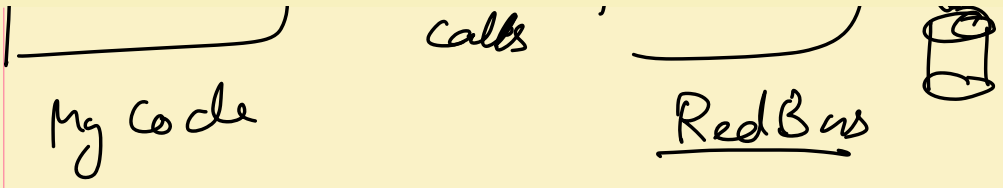
Story 1 → Sockets, USB

Story 2 → Phone Pe – Yes Bank

Story 3 → Collections.sort

20-30 minute { Interfaces & Abstract Class }





Proxy Design Pattern

Creational — 5

Structural — 2

Behavioural — 3

Decorator
Adapter

Strategy
Builder
Observer

1. Good Evening
2. Lecture begins at 9:08 pm
3. Topic → SOLID Principles

Agenda

1. ✓ Recap interfaces [with interview questions]
2. Abstract keyword → Method
   → Class
3. Interfaces vs Abstract classes

Break

4. SOLID Principles intro
5. SRP
6. OCP

Recap on interfaces

→ A contract of functions

- ✱ Loose Coupling
- ✱ Code to an interface & not an implementation.

Story 1 → Socket & Plug
USB & Type C

Story 2 → PhonePe → YesBank, SBI, ICICI → Open Bank API

Story 3 → Comparable & Collections.sort

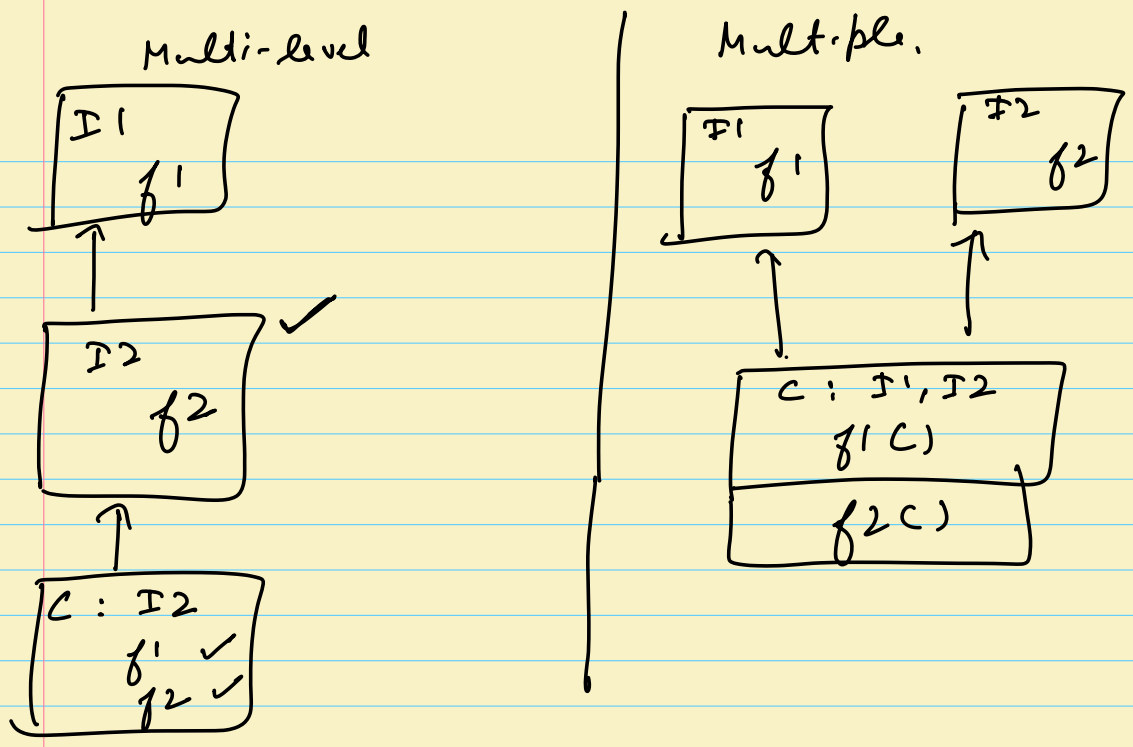
Q1. Do interfaces support multi-level inheritance? Yes

```
interface I1 {
    void fun1();
}

interface I2 extends I1 {
    int fun2(String s);
}
```

```
class C implements I2 {
    void fun1() {
    }
    int fun2(String) {
    }
}
```

Q2. Do interfaces support multiple inheritance?

```
interface I1 {
    void f1();
}

interface I2 {
    void f2();
}

class C implements I1, I2 {
    void f1() {
    }
    void f2() {
    }
}
```




```
interface I1 {
    void f1();
}

class C implements I1 {
    void f1() {
        ==
    }
}

void fun(String s, int :) {

}
```

Signature : fun (String, int)

- ☆ class extends class
- ☆ class implements interface
- ☆ interface extends interface

Q1. Multi-level inheritance via interfaces ✓

Q2 Multiple inheritance via interfaces ✓

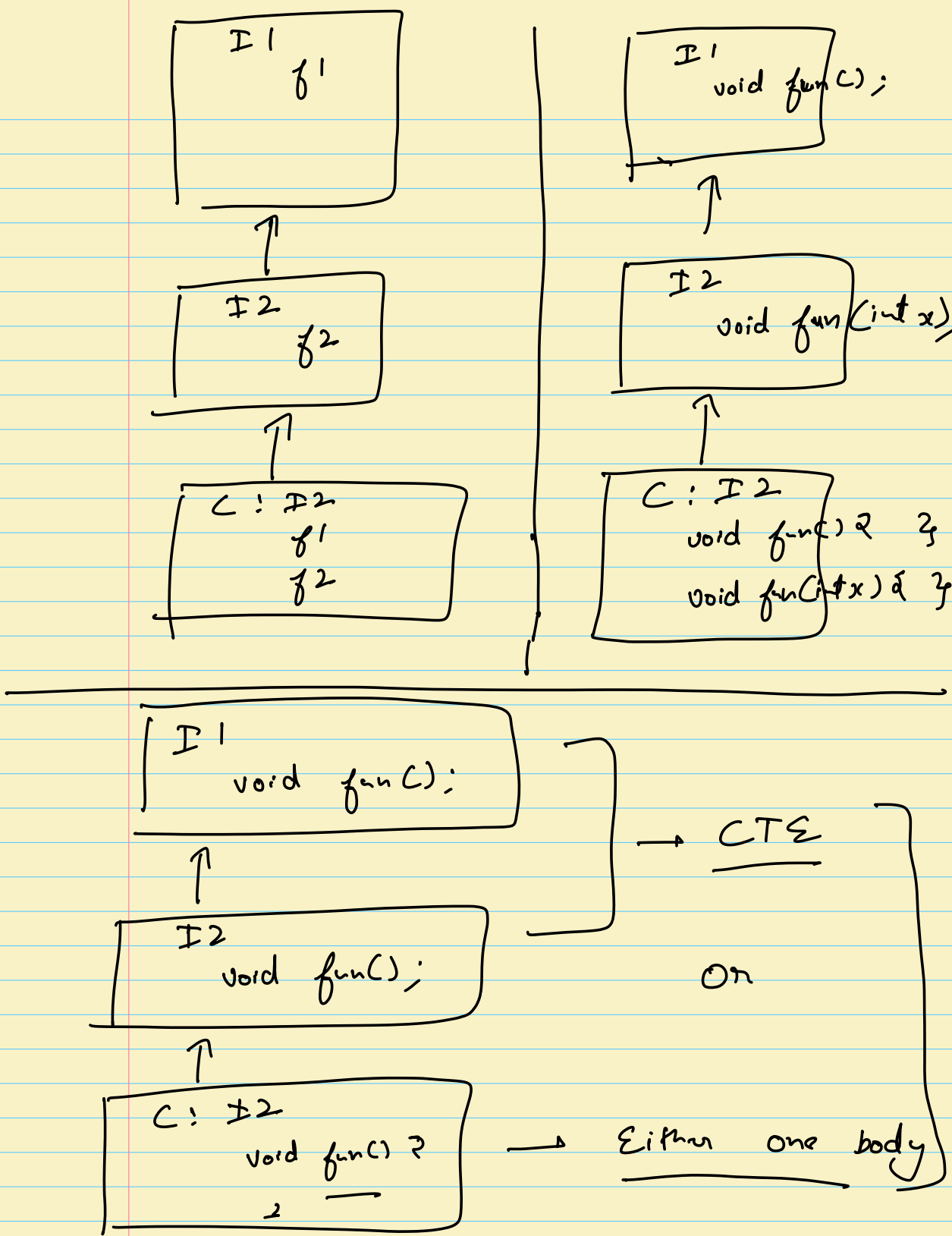
Q3 Cfors

Q4 Objects

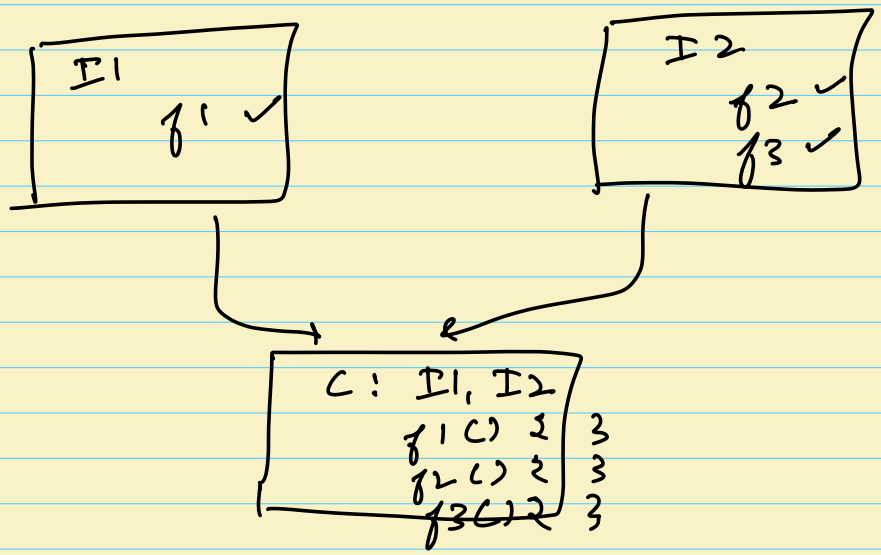
Q5. Data members

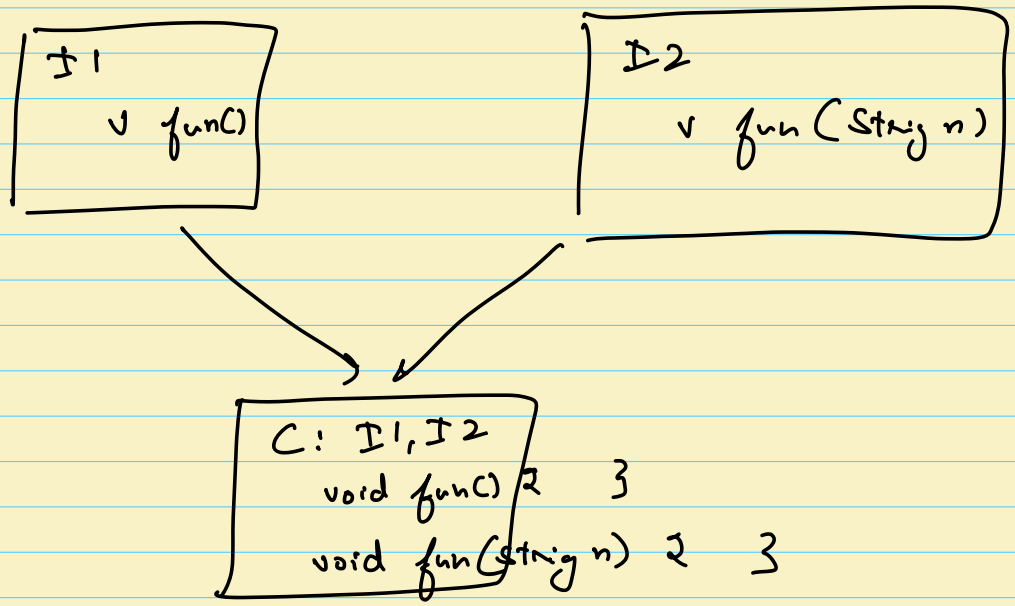
Q6. Fns. definitions in interfaces

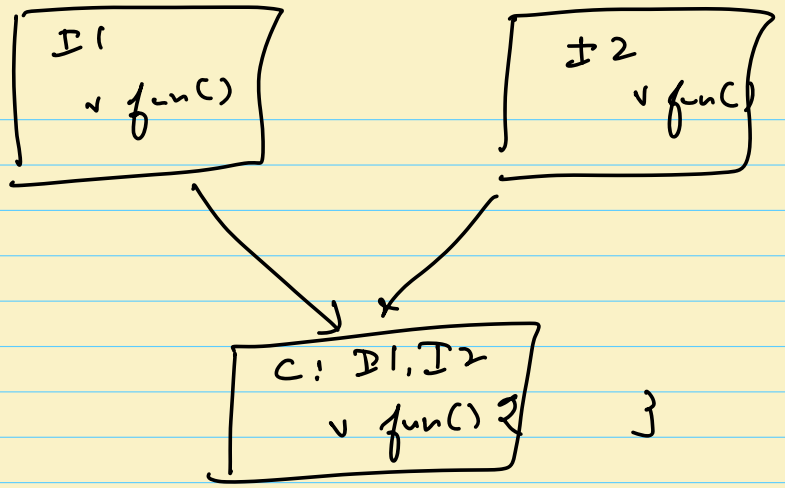
Q7. Diamond problem. :


I1
f1
I2
f2
C : I2
f1
f2
I1
void fun();
I2
void fun(int x);
C : I2
void fun() { }
void fun(int x) { }
I1
void fun();
I2
void fun();
C : I2
void fun() { }
→ CTE
OR
→ Either one body

Multiple inheritance

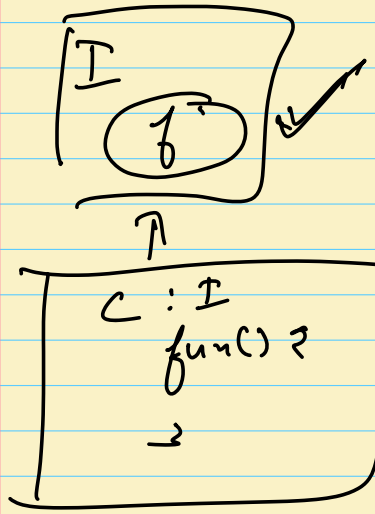






Q3. Ctors in interface ?

↳ Not required 😊 we can't make objects of interface



I obj = new C(); ✓

↳ Use as a reference & not as an instance

```
I obj = new I()
obj.fun();
```
✗

Q4. Can we make objects of interface?

→ No, 😊 fns. don't have definitions.

Qs. Can interfaces have data members?

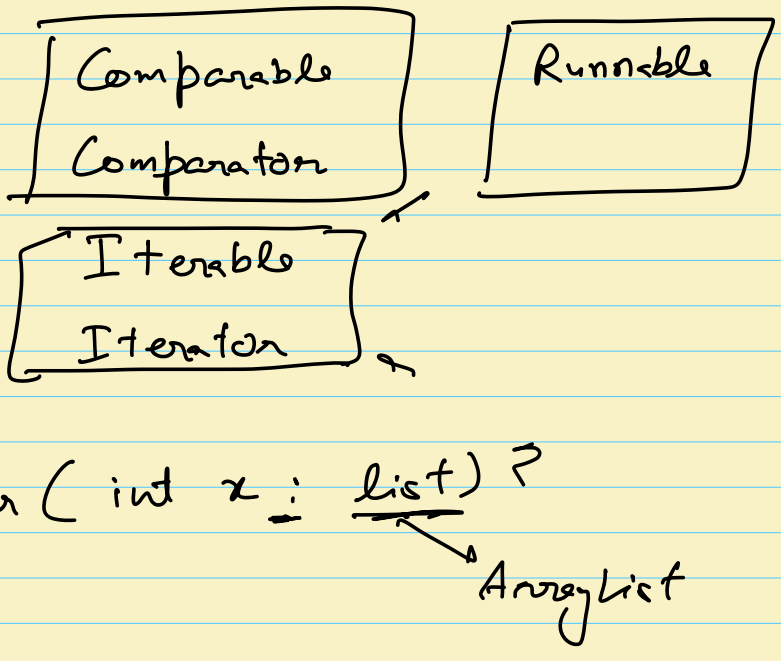
→ OOPs definition says no

→ Java allows them as static & final

→ Interface data members are used to store constants.

Code :

Examples →



BST class → Iterable & Iterator

for (int val : tree) ?

s

Break : 10:24 to 10:33

→ Abstract ← methods / classes

→ Abstract classes vs Interfaces

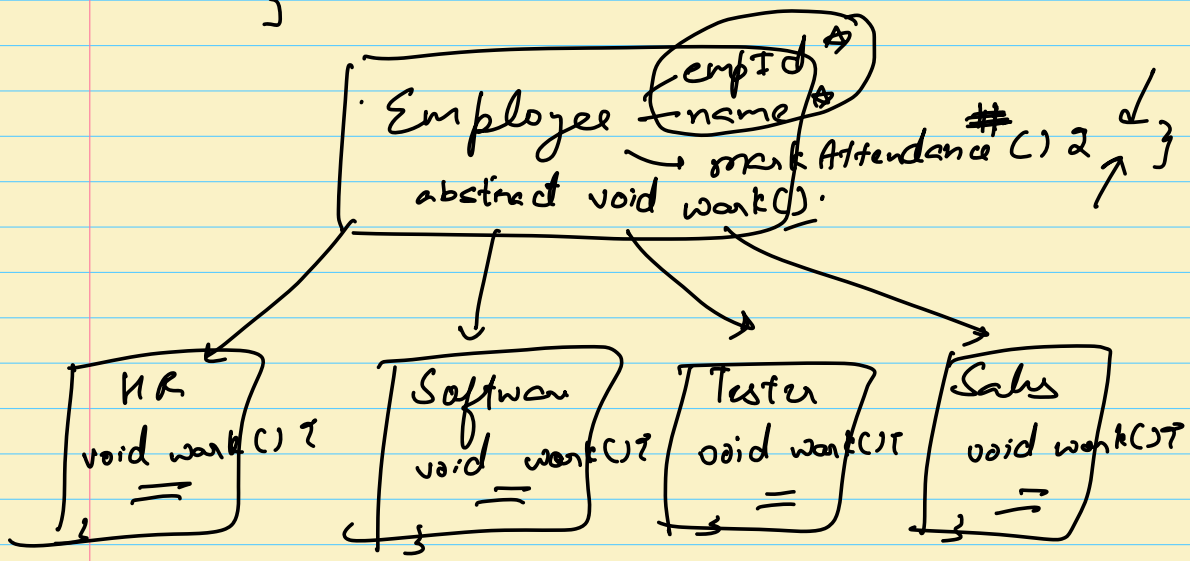
Abstract keyword [nearly opp of final]

final ← variable / data member
final ← method
final ← class

abstract ← methods / class

☆ Similar to interfaces

```
class Employee {
    int eId ;
    String name ;
    void markAttendance() {
    }
    abstract void work();
}
```



☆ A middle ground btw interfaces (pure contract) & classes

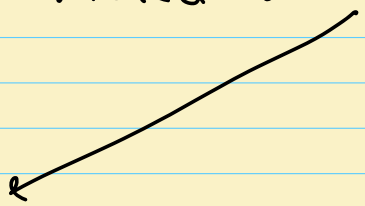
→ Can have attributes {which are instance level}

→ Can have some fns. with definitions

→ Can define contract via methods marked as abstract

* abstract method → A fn. without definition is called abstract
(defining contract like an interface)

* abstract class → If we have any method marked as abstract, then class has to be marked as abstract.

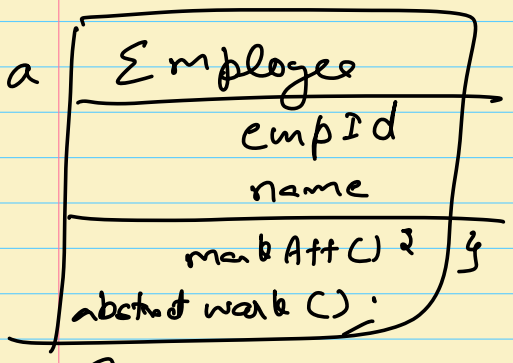


We are not allowed to create objects of abstract classes.

AM → no body

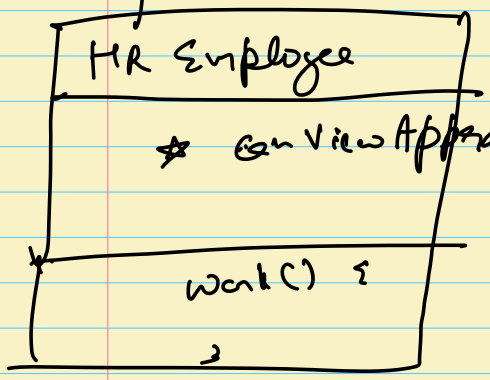
AC → no object

☆ Can an abstract class have a cfor?



Employee (Sn, i eId)?
name = n;
empId = eId;

Employee e = new HREmployee ("S", 2113, true)

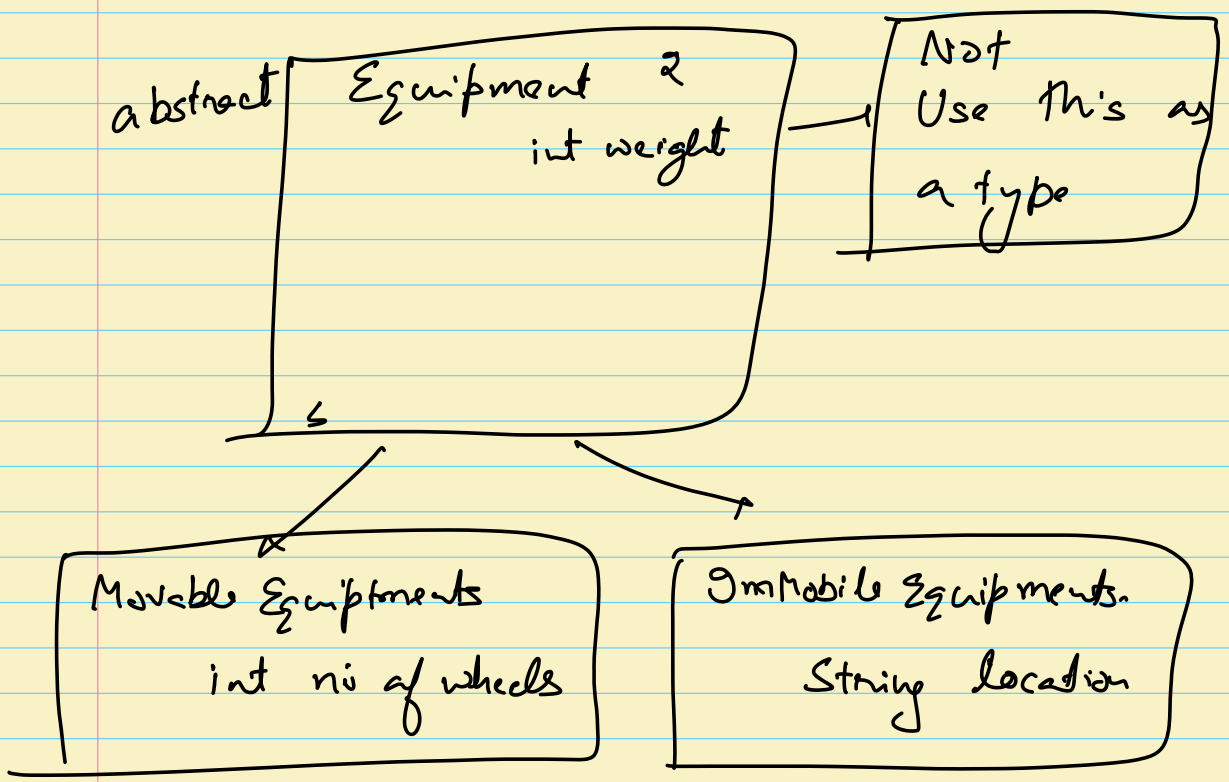


HREmployee (Sn, i eId, b cva)?
super(n, eId);
cvc = cva;

Q2. Does abstract method imply that the class will be abstract?

→ Yes. [To prevent creation of object]

Q3. Does an abstract class necessarily have an abstract method?

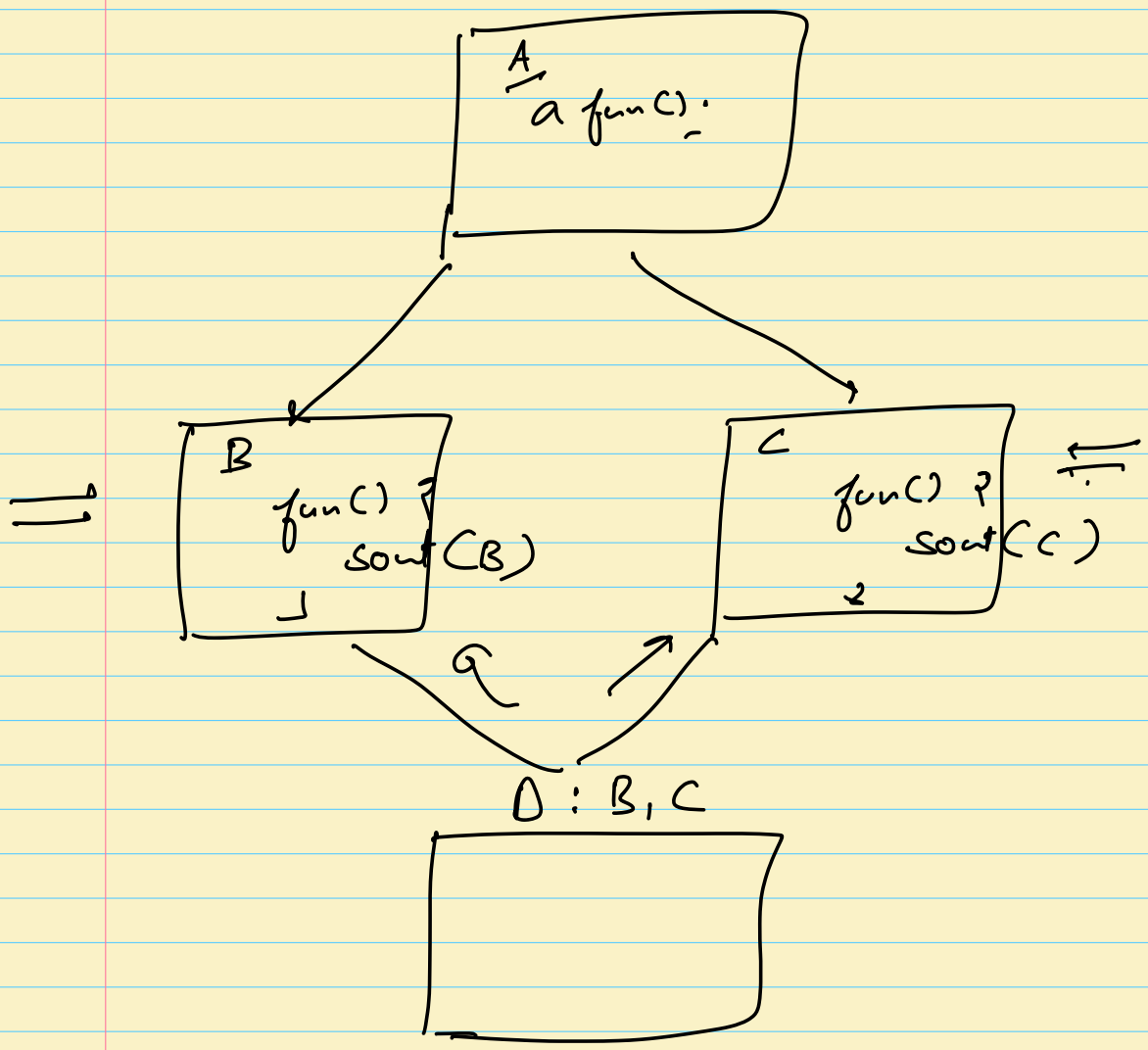


Q Can we do multi-level inheritance via abstract classes?

↳ Yes.

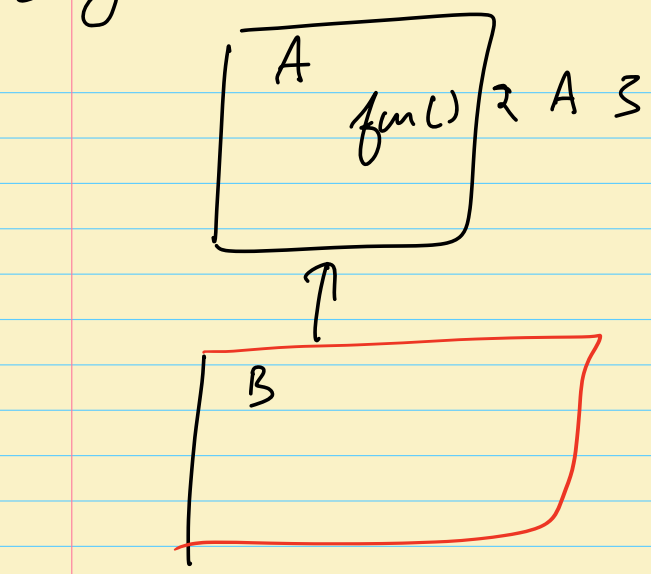
Q Can we do multiple inheritance via abstract classes? → NO

✡ Diamond Problem.



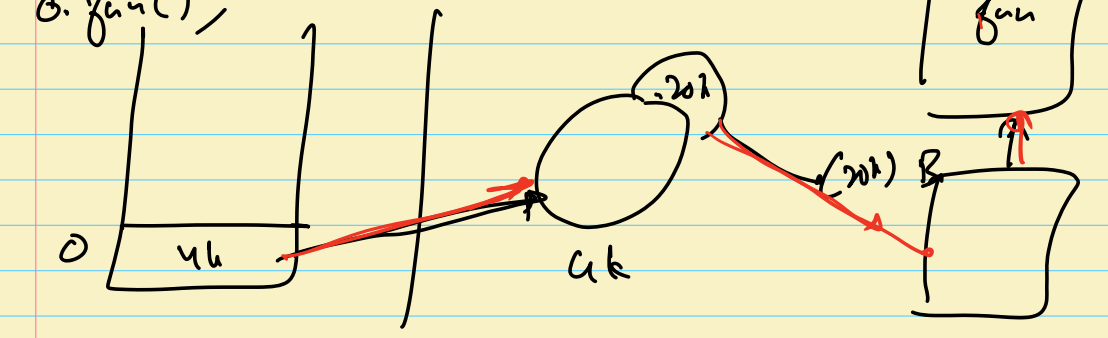

```
A o = new D()
o.fun()
```

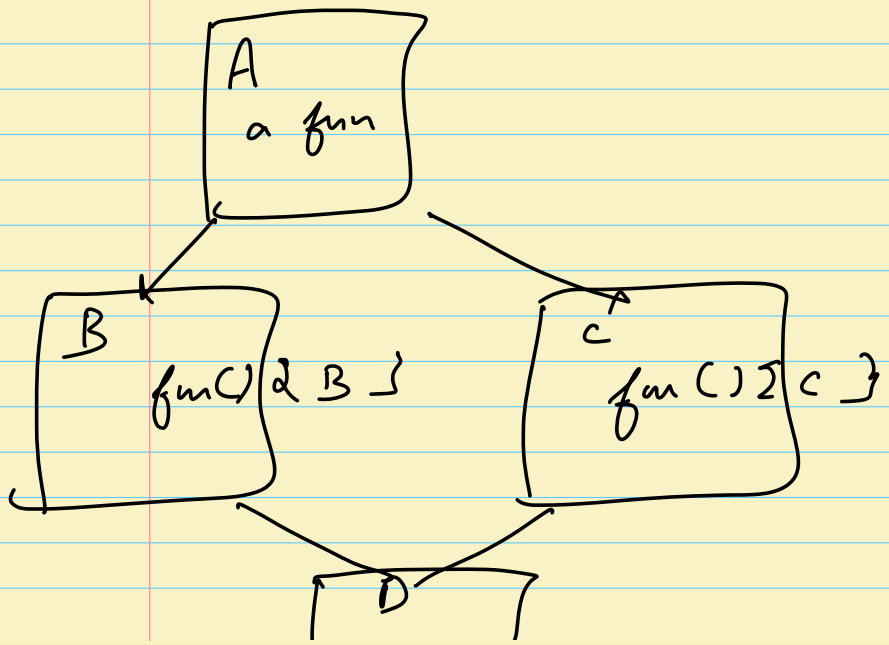
Easy Example

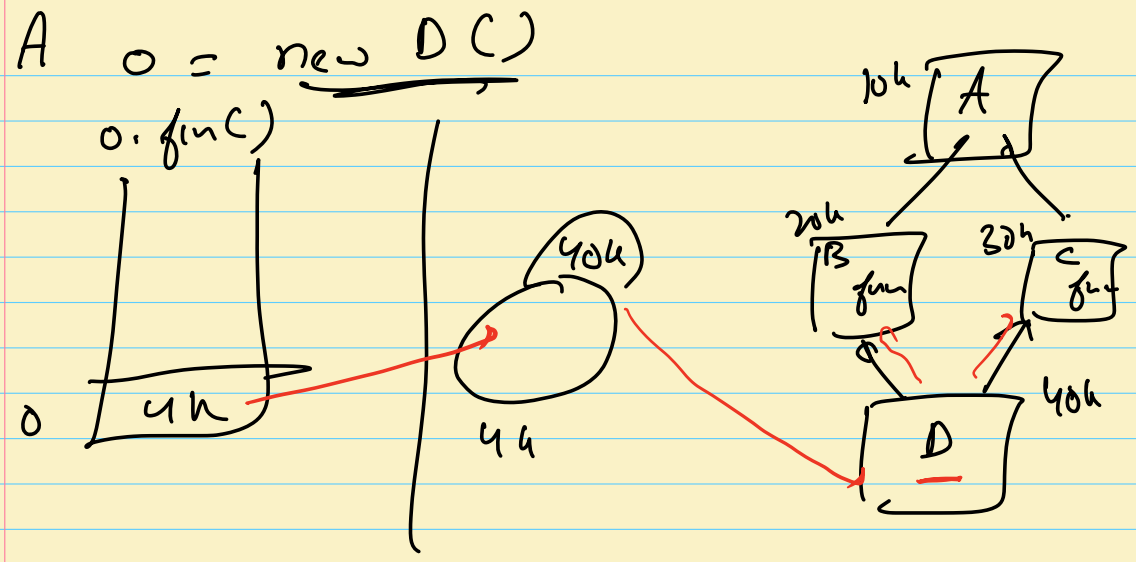


A o = new B()

o.fun();







It is one of the above problem, Java doesnot allow multiple inheritance for classes (allowed for interface)

Differences btw abstract class & interfaces

Abstract classes	Interfaces
→ Not all fns. need to be abstract. We can have methods with definitions. ✓	→ All fns. are abstract ✓

→ Ctor allowed ✓	→ Ctor not allowed ✓
→ Data members are instance level ✓	→ Data members are static & final ✓
→ Multiple inheritance not allowed	→ Multiple inheritance allowed.

* default keyword with interfaces.

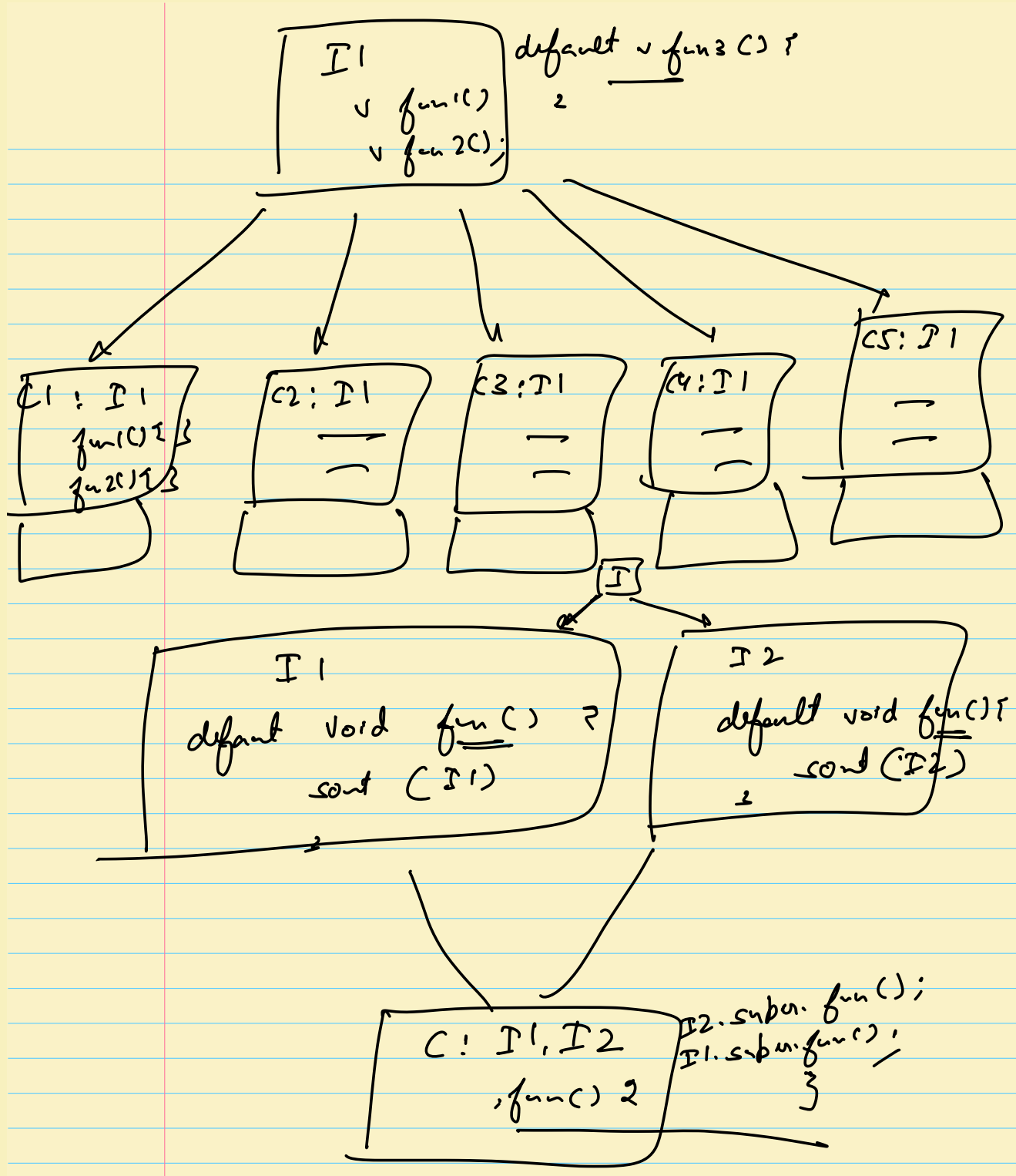
(Java 8)

→ Allowed interfaces with fn. bodies

→ OOPs Violation.

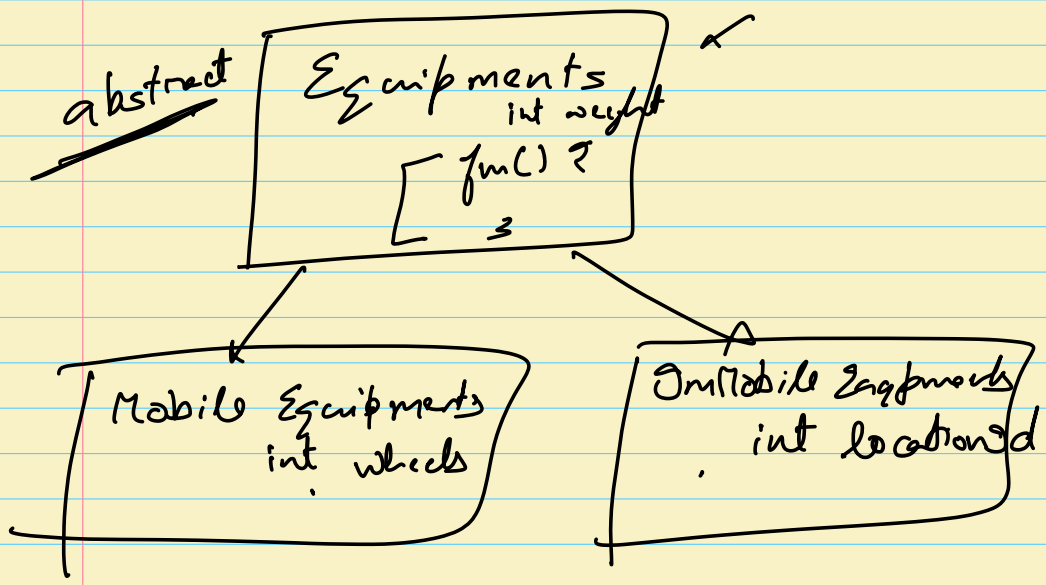
→ Why is this allowed?

→ Diamond problem will happen with interfaces?



I obj = new C();

obj.fun();


abstract
Equipments
int weight
fun() ?
Mobile Equipments
int wheels
Immobile Equipments
int locationId

1. Good Evening

2. Lecture begins at 9:08 pm

3. Topic - SOLID Principles

Agenda

→ SOLID Principles

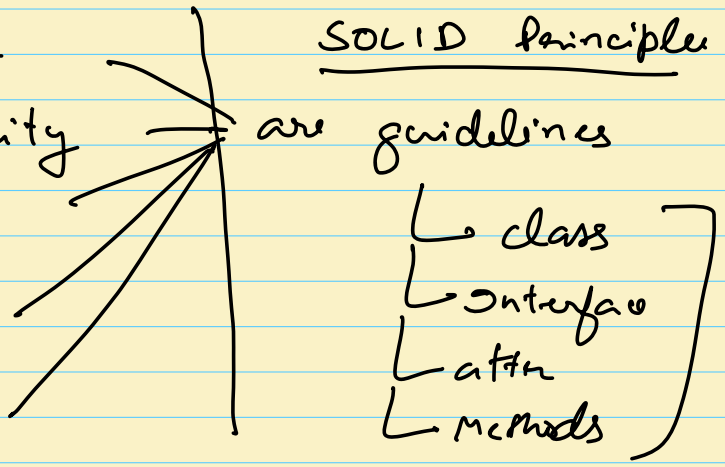
- ✓ S → SRP → Single Responsibility
- ✓ O → OCP → Open Closed
- ✓ L → LCP → Liskov's Substitution
- I → ISP → Interface Segregation
- D → DIP → Dependency Inversion

Introduction : SOLID Principles

Guidelines to aid engineers to do better design of software systems.

Features of a well designed software

1. Extensibility
2. Maintainability
3. Testable
4. Reusability
5. Readable



Design a bird?

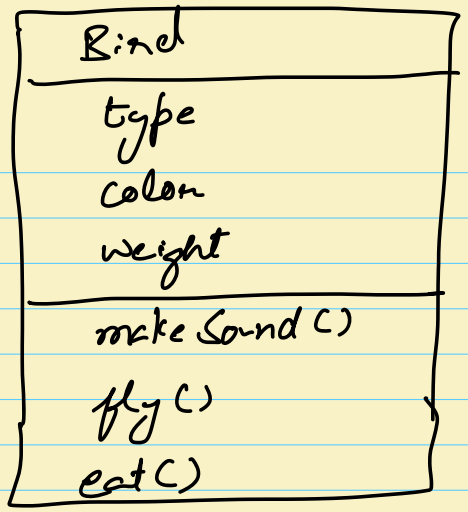
→ LLD is subjective

→ LLD interview are discussion oriented

→ Design is iterative

Design a bird?

VO




```
Bird b1 = new Bird("sparrow", "gray", 500);
b1.makeSound()
b1.fly()
Bird b2 = new Bird("Pigeon", "white", 1000);
b2.makeSound();
```

```
makeSound() {
  if (type == "sparrow")
    [sout("chi chi")]
  elif (type == "pigeon")
    sout("gutargoon")
  elif (type == "crow")
    sout("caw")
}
```

```
fly() {
}
```

squeaks() { } squeak like a pigeon

Problems

1. Readability
2. Testability
3. Parallel development [Merge Conflict]
4. Code Duplication → ☆
5. SRP is violated
   └ Single Responsibility principle.

☆ Every code unit (method, class, pkg) should have a single responsibility

↓

A single reason to change the code

```
makeSound() {
    if (type == "sparrow")
    ==
    elif (type == "pigeon")
    ==
    elif (type == "crow")
    ==
}
```

makeSound can change

1. If there is a new type
2. Sound of an old type changes.

makeSound is violating SRP

How to identify SRP violation?

→ Code Smell { common scenarios for known problems }

1. Multiple if-else clause in a code unit which are not related algorithm

```
void doSomething() {
    inp = takeInput()
    if (inp == 1) {
        // play a song
    }
    else if (inp == 2) {
        // shutdown
    }
    else {
        // open a browser
    }
}
```

└ 3

doSomething can change

1. If a new input has to be handled
2. If the logic to "open a browser" changes.

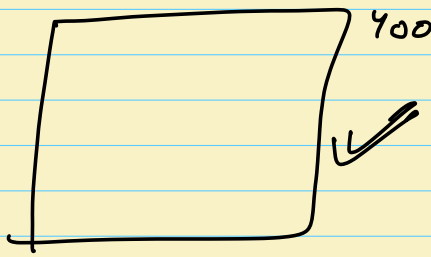
How to resolve ?

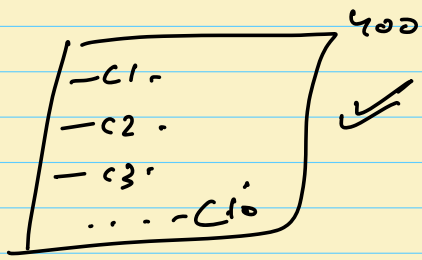
└ Write new code units.

```
void doSomething() {
    inp = takeInput()
    if (inp == 1) {
        playASong()
    }
    elif (inp == 2) {
        shutdown();
    }
    elif (inp == 3) {
        openABrowser();
    }
}
```

→ Handles SRP better than previous one.

doSomething fn. now changes when a new input is identified, & not when logic to "open a browser"



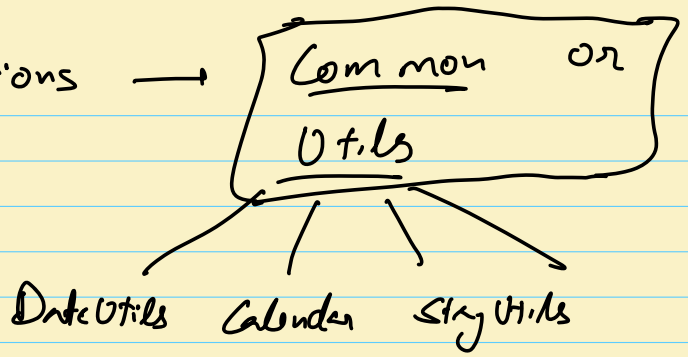


2. Monster Methods : Huge methods trying to do a lot more work than what their name suggests

```
void saveUserToDb(User) {
    [String query = "_____"]  →
    {
        Database db = new Database()
        db.setUrl()
        db.makeConnection()
    }  →
    [ db.execute(query, user);
}
```

```
void saveUserToDb(User) {
    String query = getQueryForSavingUser()
    Database db = getDatabaseConnection();
    db.execute(query, user);
}
```

3. SRP violations → Common or Utils



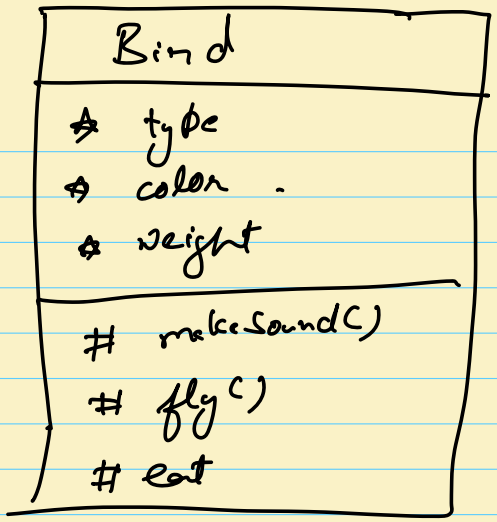
Summary on SRP [Single Responsibility]

1. Every code unit should have only one reason to change.

2. Code Smells for SRP violation
   - If-else / Switch
   - Monster methods
   - Most common [Utils & Commons]

3. Introduce new code-units [methods]

V1




```
void makeSound() {
    if (type == "pigeon")
        makePigeonSound();
    else if (type == "sparrow")
        makeSparrowSound();
    else if (type == "crow")
        makeCrowSound();
}
```

SRP is better preserved when we call separate methods from "makeSound" ∴ now it will change when a new type is introduced & not when logic to produce a specific bird's sound changes.

V0 ⟶ V1 ⟶ V2

Break : 10:17 to 10:27

S O L I D

↳ Open - Closed principle

1. Open for extension
2. Closed for modification

★ Adding new features should generally involve writing new code-units & no/low code change in existing units.

Why is OCP required?

→ Regression Testing

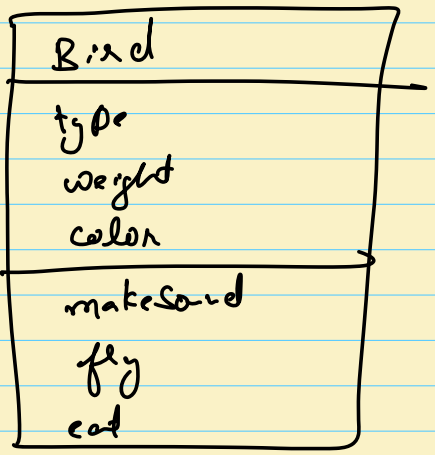
* When new code is written, old functionalities should not alter.

↓

OCP

v1 → sparrow, pigeon, crow

→ peacock



makeSound() {

if (type == "sparrow")
makeSparrowSound();

else if (type == "pigeon")

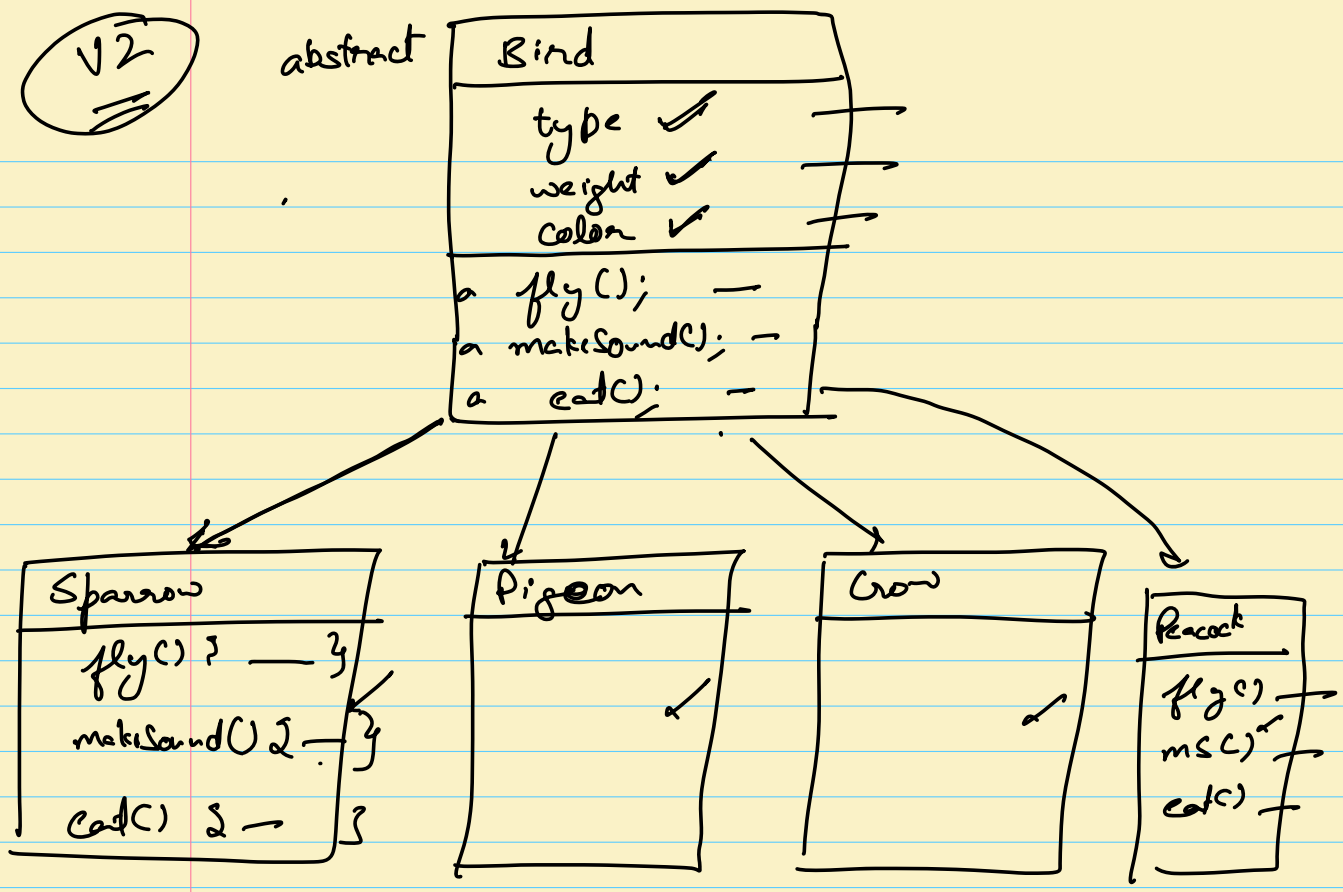
else if (type == "crow")

else if (type == "peacock")

OCP?

makeSound is violating OCP. To add the peacock's functionality old code had to be modified.

V2

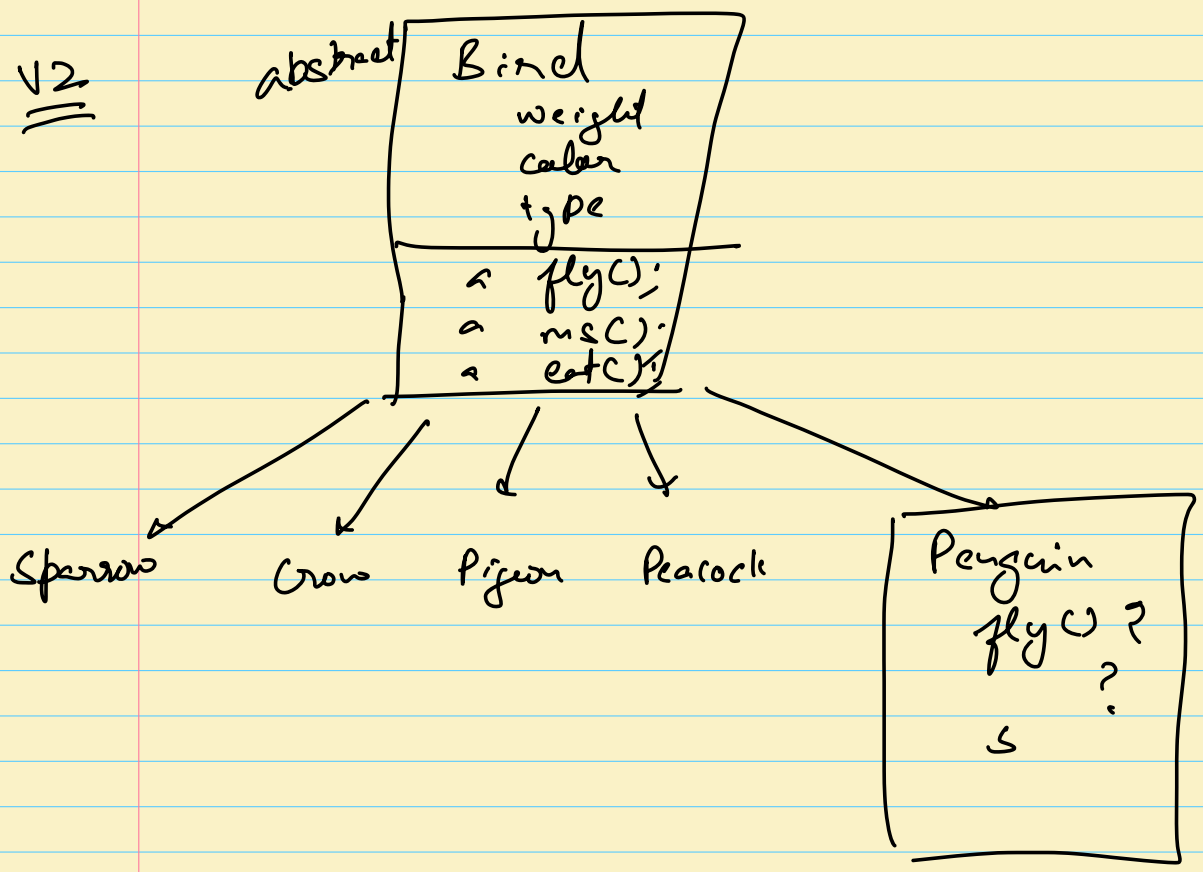


V2 follows OCP :) to add a new functionality we need not change old code.

V2 has improved SRP
Diff classes have their own makeSound now.

New Requirement ⟶ How about Penguins?

V2



fly in Penguin

1. Write nothing
2. Throw an Exception [Not implemented]

Guideline ⟶ [Don't advertise what you can't do.]

→ Such fns. [fly with nobody]
will surprise the client [other developer]

UberEats → Paytm API

UberEats
PayTM pobj
checkoutOrder() {
if (pobj.status != failure) {
orderprocess();
→ success

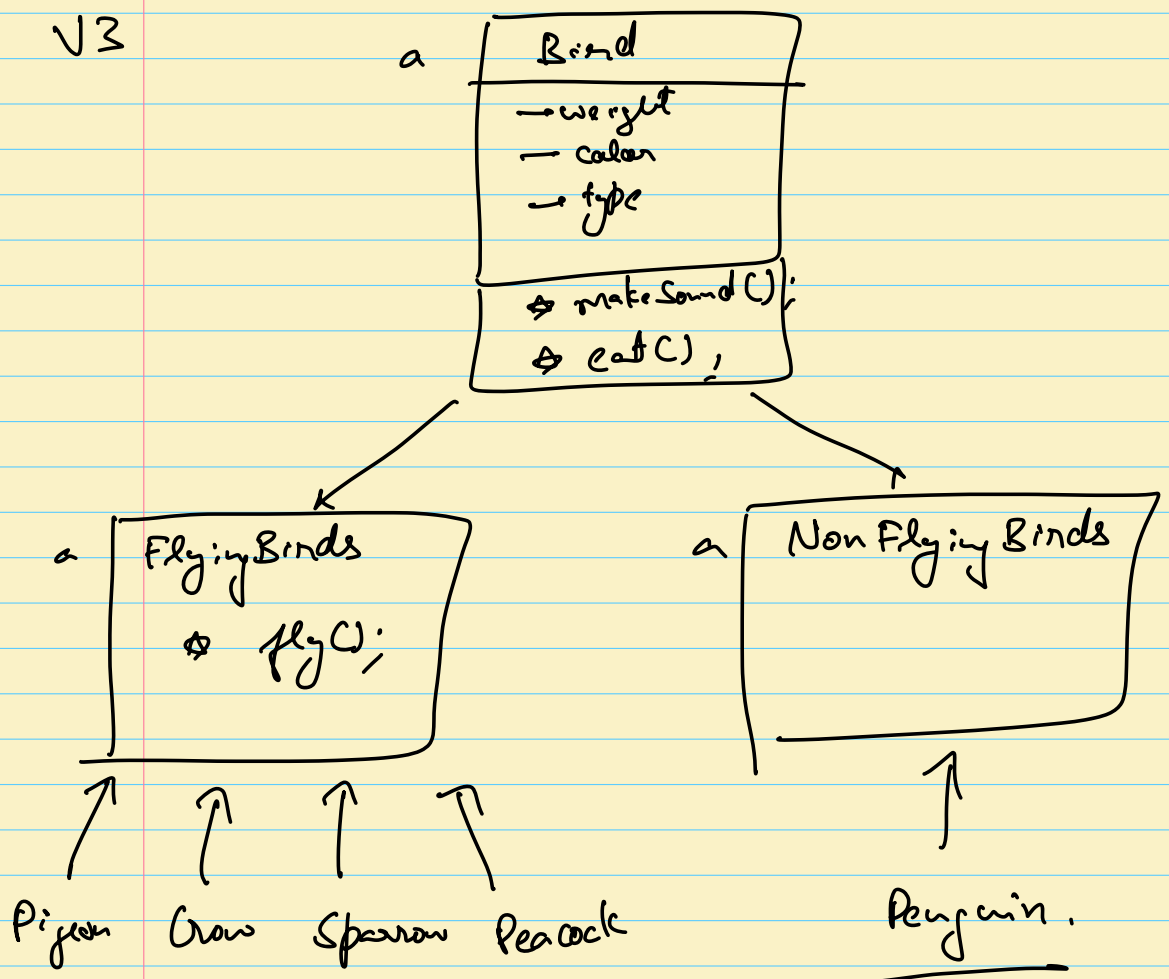
PayTM
Status → Success
failure
Error on Bank Gateway

PayTM Api surprised UberEats Api

Ideal Soln ⟶ If an entity can't do an operation, it should not advertise it

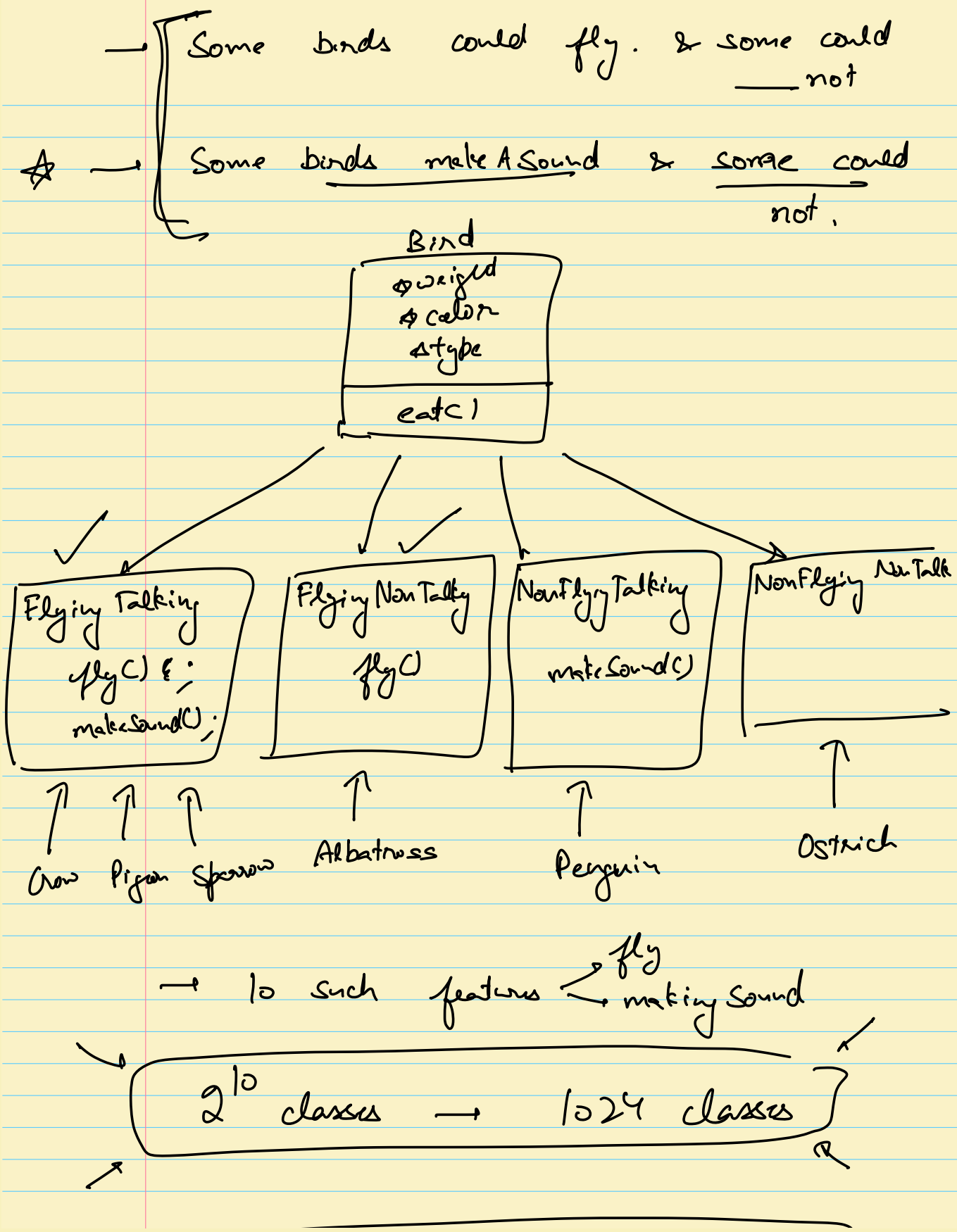
---

V3



Problems in this design

1. Class Explosion


Some birds could fly. & some could not
Some birds makeASound & some could not.
Bird
weight
color
type
eat()
FlyingTalking
fly();
makeSound();
FlyingNonTalky
fly()
NonFlyingTalking
makeSound()
NonFlyingNonTalk
Crow
Pigeon
Sparrow
Albatross
Penguin
Ostrich
10 such features
fly
making Sound
2^10 classes → 1024 classes

→ List of all Flying Birds ?

List <?>

List <Flying Talking>

List <Flying NonTalking>

List <Birds>

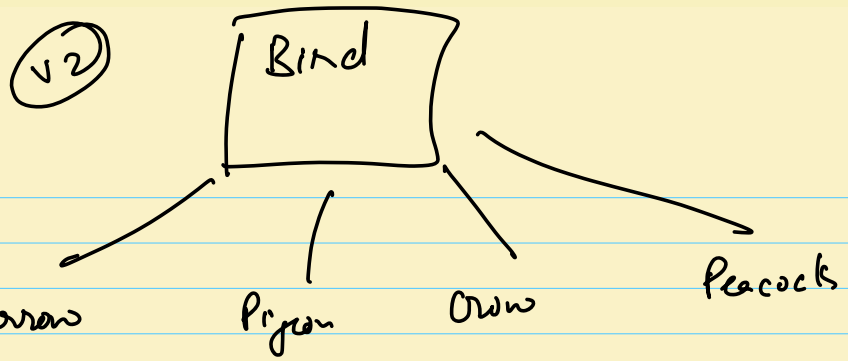
Next class → SOLID

Summary

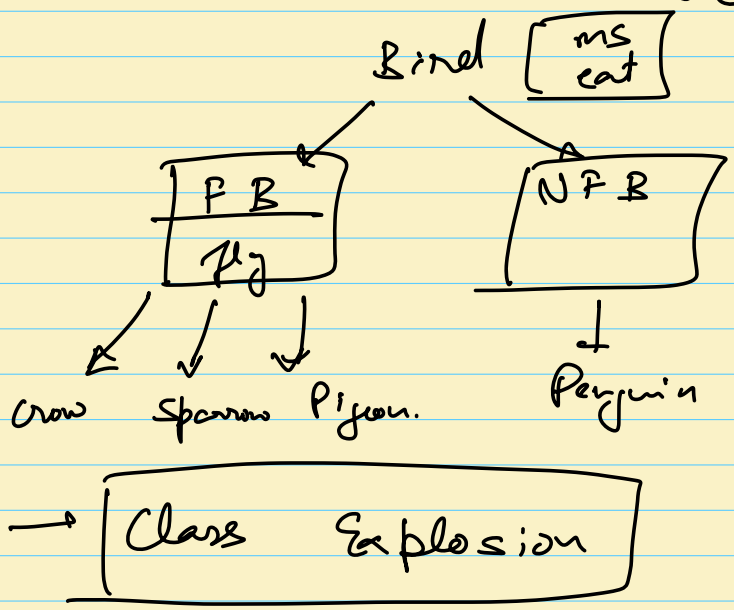
1. SRP → V0 [Monster Methods]
   ↓
   V1 [Bird
   └ ms()
   if
   elif
   elif
   elif

2. OCP [Peacock] → elif

V2



3. How to support Non flying birds?



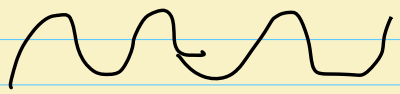
→ Class Explosion

1. Good Evening
2. Lecture begins at 9:10 pm
3. Introduction to Design Patterns.

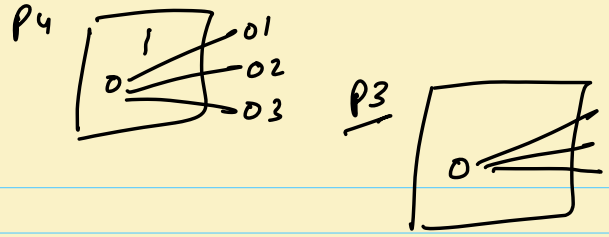
Agenda

1. Introduction to Design Patterns
2. Categories of Design Patterns.
3. Creational Design Patterns
   - Singleton Design Pattern.

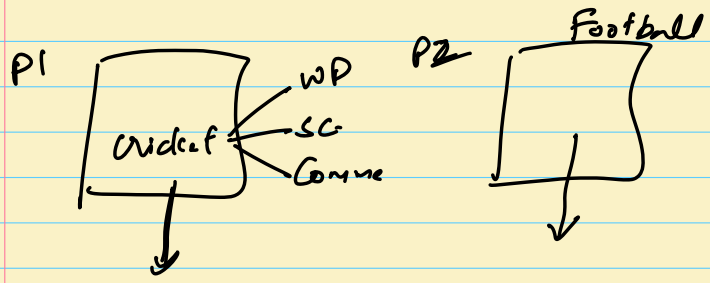
What are Design Patterns?

Pattern → 
- Something repititive

Design → Design of maintainable ... software systems.



Design Pattern



* Well established solutions to common design problem

→ 4 authors : GoF — Gang of Four

→ 23 Design Problems

~ 10 Design Patterns

→ Common patterns in codebase

→ " " asked in interviews.

Solid Principles → 23 problems*

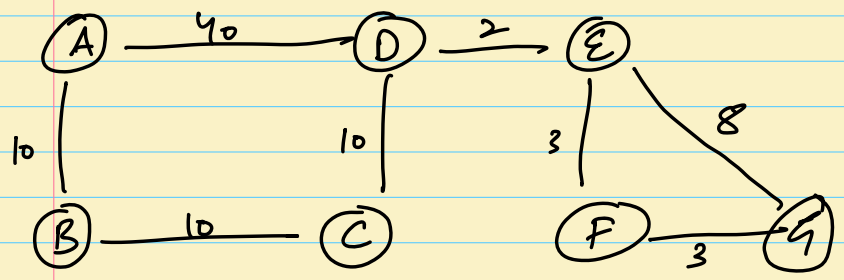
↓ Guidelines

Problem + Solution

Design Pattern.

Why should we learn Design Patterns?

1. Shared Vocabulary? [To save time]



Shortest Path [in kms] A to G

Djikstra → Heap / PQ

Hashmap

30 minutes [60 minutes]

* How to design a class which allows only one object to be created?

* Singleton

2. Interview Questions?

Category of Design Patterns?

- ★ 10 - 15 minutes
- ★ Solution [to me]

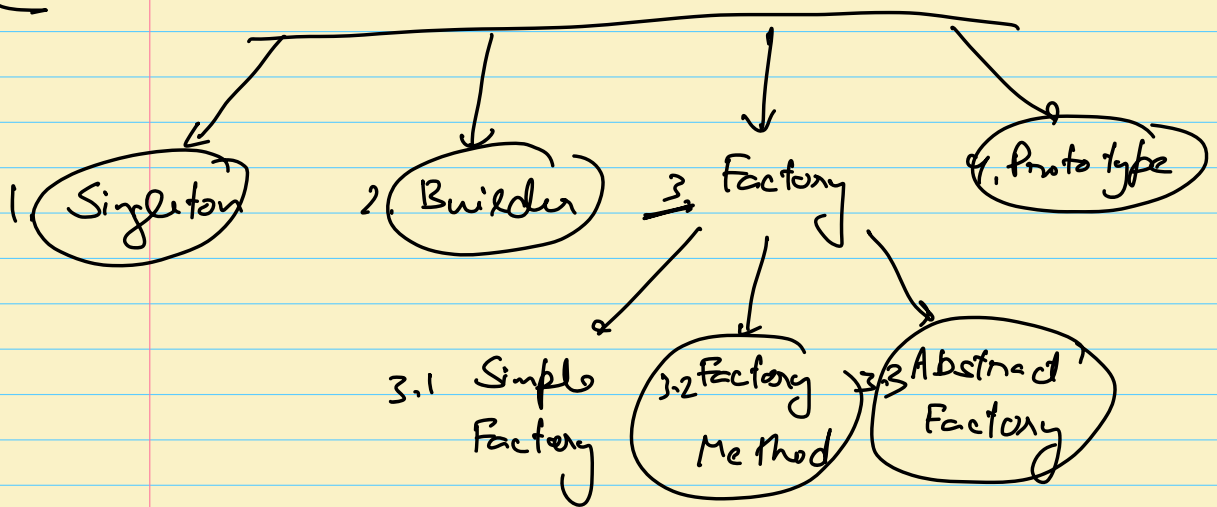
→ Common problems W.E Solutions in OOD?

Life Cycle of objects → Created

1. Creational Design Patterns?

   How an object can be created?

   How many objects can be created?

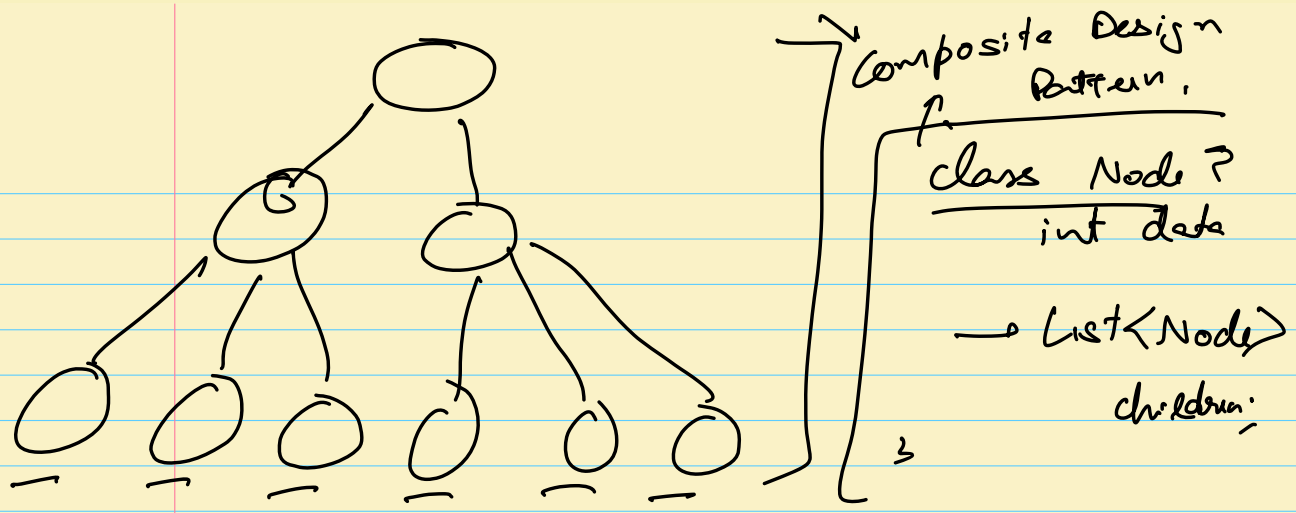


new keyword.

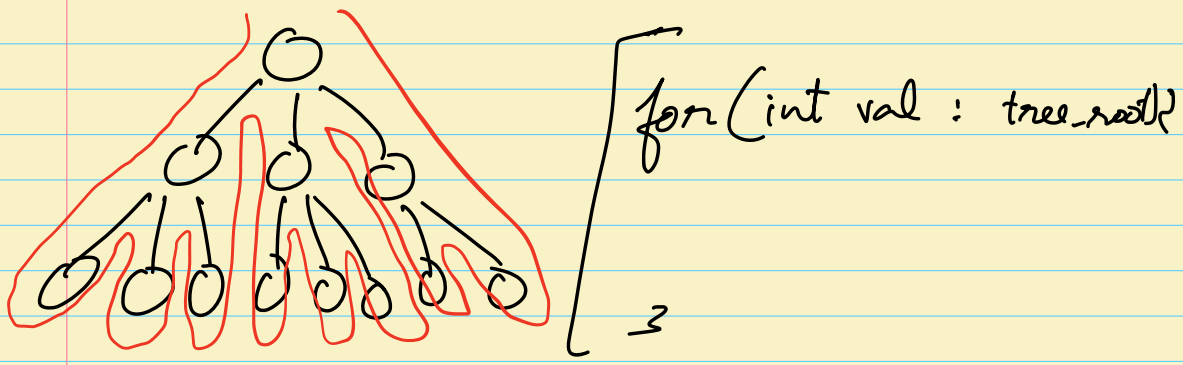
2. Structural Design Patterns.

   → How will one class be composed

   → Attributes



3. Behavioural Design Patterns.
   - To add a behaviour



★ Iterator Design Pattern.

Creational Design Pattern
- Singleton Design Pattern.

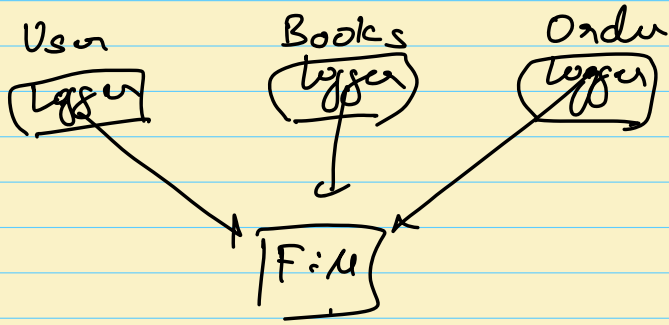
Singleton Design Pattern. [Problem Solution]

- Defn.
- Why? [Problem Statement]
- How? [Implementation]
- Pros
- Cons
- Code

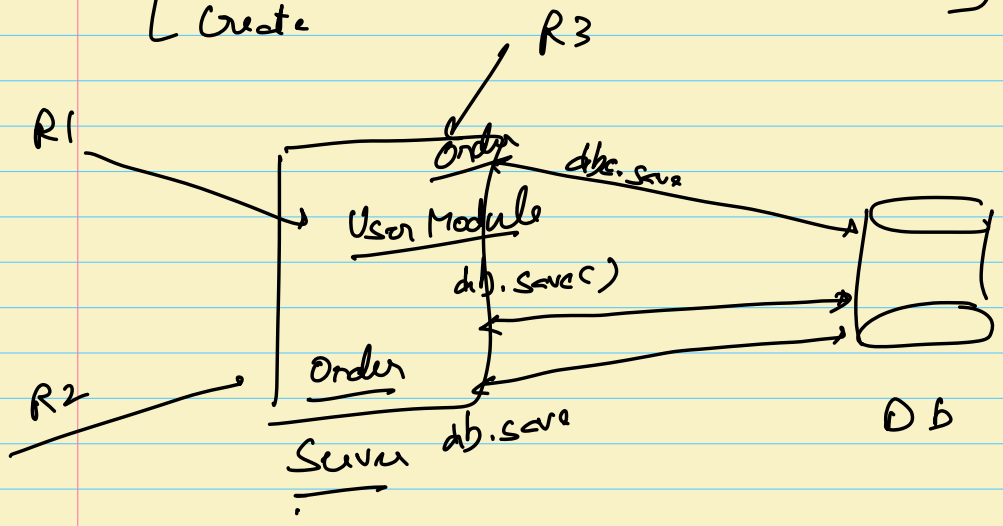
Definition → [Allows you to create a class for which only one object can be created.]

Why? → 1. Shared resources.

e.g. Logger. → only one object.



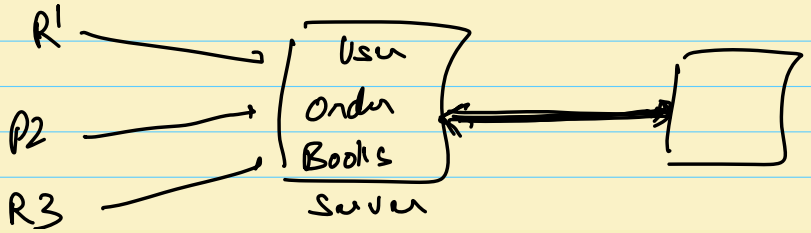
2. [If an object is slow to Create]



* Creating a db connection is Costly
  - db connection → TCP connection.

- class DbConnection {
  - String url;
  - String userName;
  - String pwd.
  - List<TCPConnection> con }

3



☆ A costly shared resource.

↓

Single object to be created.

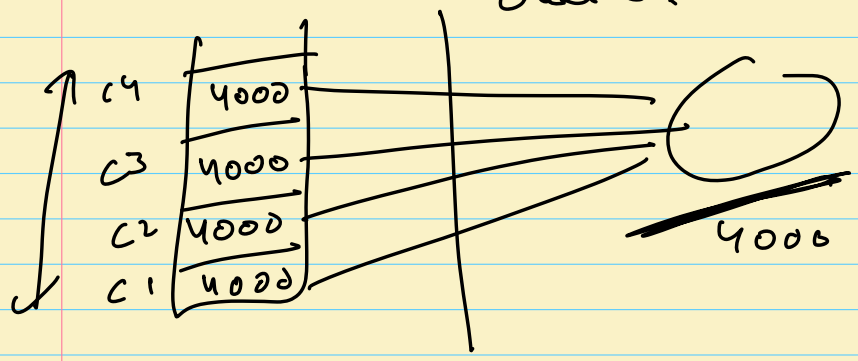
How to implement Singleton?

Version 1

```
class DbConnection {
    pvt. String url;
    pvt. String userName;
    pvt  "     pwd;
    "    TCPConnection conn;

    public DbConnection() {

    }
}
```

```
DbConnection c1 = new DbConnection();
"            c2 = new DbConnection();
```

Problem 1 → multiple instances getting created.



version 1 has a public constructor.

---

version 2

```
class DbConnection {
    // same as version 1

    // removed ctor.
}

DbConnection c1 = new DbConnection()
// c2 = new DbConnection();
```

Default ctor

Problem 2 → ∵ of default constructor, still we are able to create multiple instances

---

Version 3 → Make the ctor private

```
class DbConnection {
    // same as version 1
    private DbConnection() {
    }
}
```

```
DbConnection c1 = new DbConnection();
//            c2 = //          // ();
```

Problem 3 → Not able to create instances at all.

Version 4

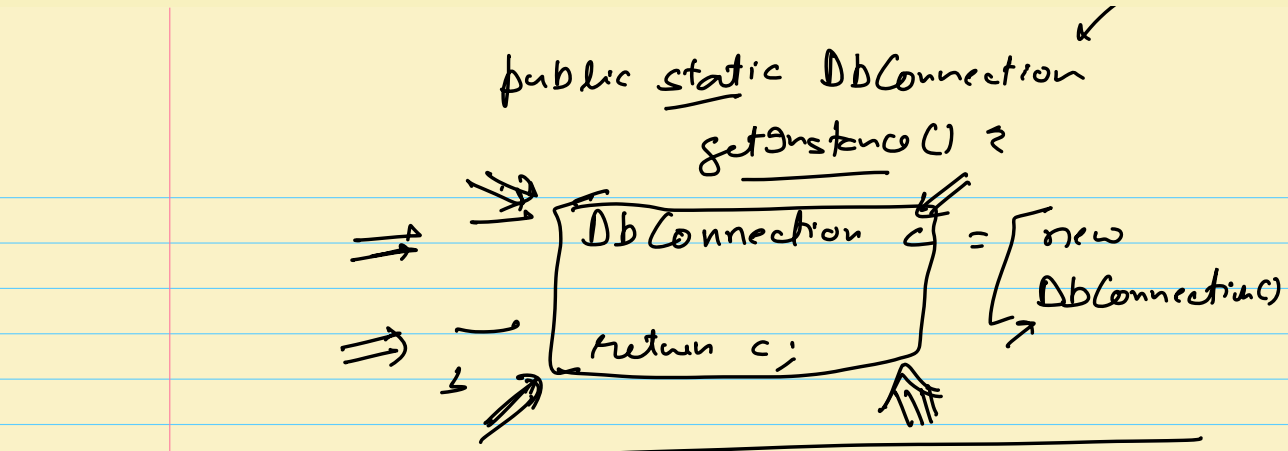
```
class DbConnection {
    // same as version 1
    private DbConnection() {}

    public static DbConnection getInstance() {
        DbConnection c = new DbConnection();
        return c;
    }
}
```

```
DbConnection cl = ?.getInstance();
```

Problem → Not able to call getInstance without having an instance.

Version 5

```
class DbConnection {
    // same as version 1
    private DbConnection() {}
}
```



DbConnection c1 = DbConnection.getInstance()

—— c2 = " . " ();

Problems → multiple instances getting created again

Version 6

✓ → local variable ✓

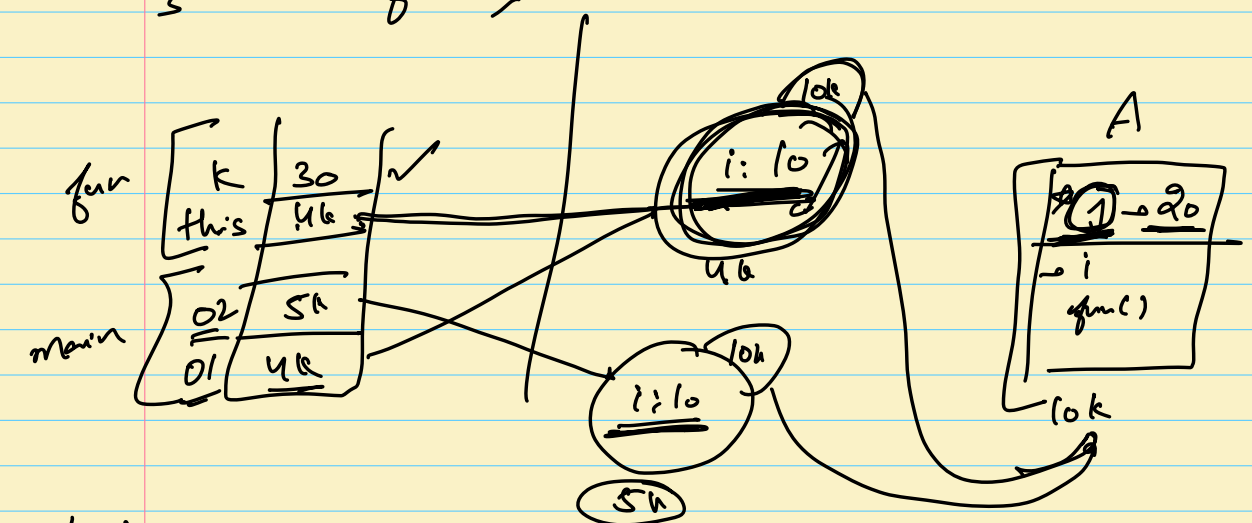
✓ → instance level data member ✓

✓ → class " " " ✓

↓

static

```
class A {
    int i = 10 ✓
    static int j = 20 ✓

    void fun() {
        int i = 30; ✓        2 ✓
        sout(i) ✓
        sout(j) ✓
        sout(F) ✓
    }
}

main() {
    A o1 = new A() ✓
    A o2 = new A() ✓
    o1.fun(); ✓
    o2.fun();
}
```



o1.i
o2.i

o1.j A.j
o2.j

Version 6

```
class DbConnection {
    // same as version 1
    private DbConnection() {}

    private static DbConnection inst = null;
    public static DbConnection getInstance() {
        if (inst == null) {
            inst = new DbConnection();
        }
        return inst;
    }
}
```

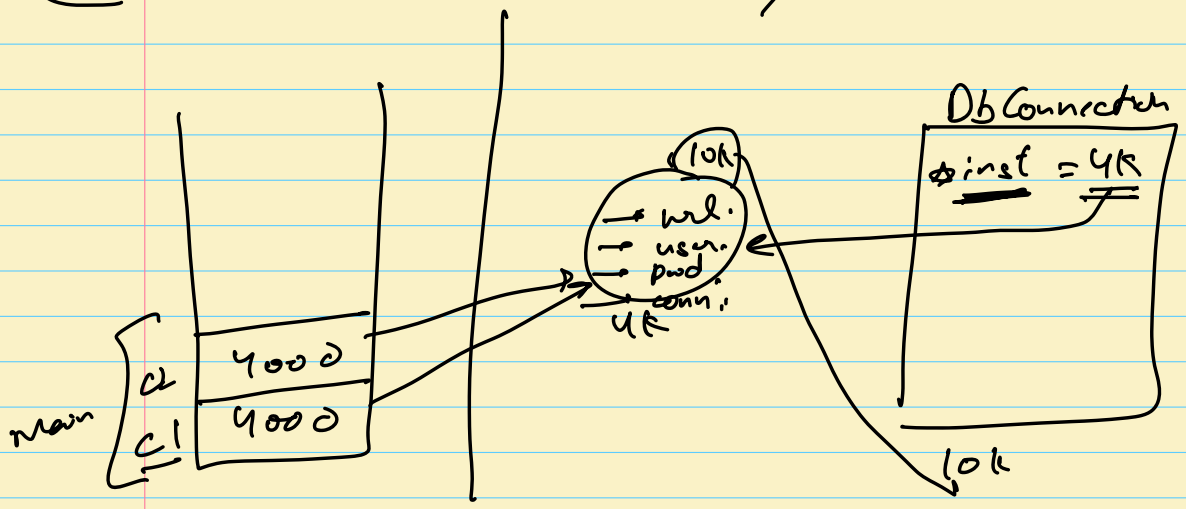
1. Can static methods access non-static data members? No

2. Can non-static methods access static data members?

main

DbConnection c1 = DbConnection.getInstance(); (4000)

// DbConnection c2 = DbConnection.getInstance(); (4000)



Version 6 has nearly solved the problem

or

completely?

* Is there a situation where version 6 fails?

On multi-threaded application version 6 will fail?

Break → 10:36 to 10:46

- How multi-threading can break the code?
- How to solve?

t1 → UserService [Runnable]

run() ?

DBconn c1 = DBC.getC()

t2 → BookService [Runnable]

run() ?

Dbcon c2 = DBC.getC()

class DbConnection ?

// variables

// private cton

pvt. static DbConnection inst = null

public static ... getInstance()?

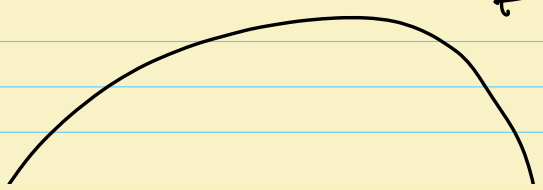
L1 → if (inst == null) ?

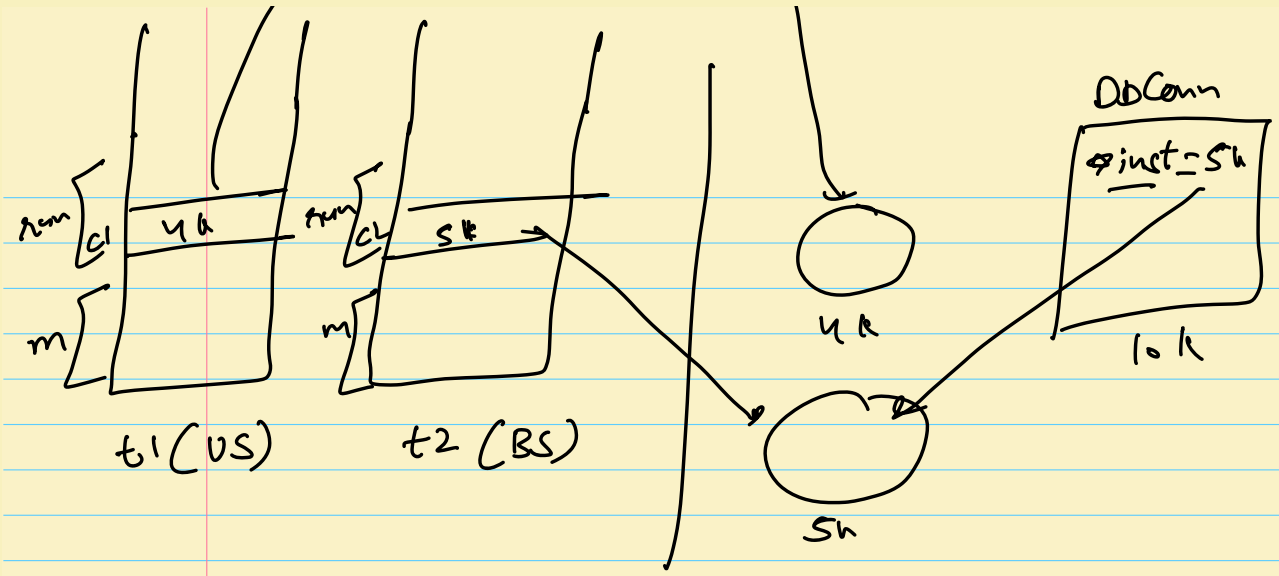
L2 → inst = new DBConn ();

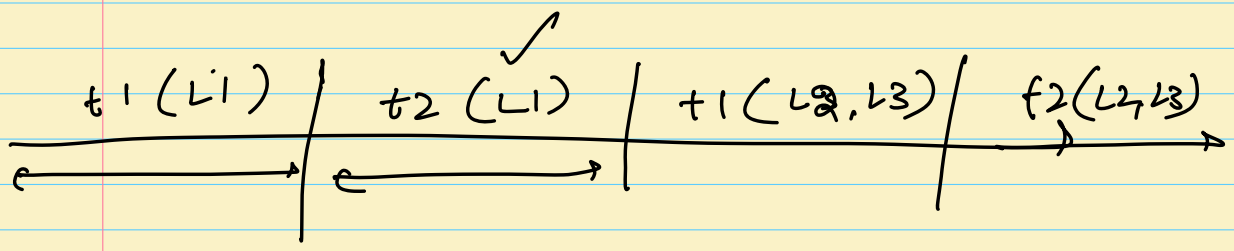
L3 → return inst;

}

t1(L1) / t2(L1) / t1(L2) / t2(L2)







☆ Thread Switching is controlled by OS.

→ [Problem of Synchronization]

- A shared data
- A critical section
- Race condition

☆ Solution of Synchronization

- lock
- synchronized

Lock or synchronized

- Allow only 1 thread in the critical section

☆ [If 1 thread is in CS, the other thread trying to enter CS blocks]

Version 7

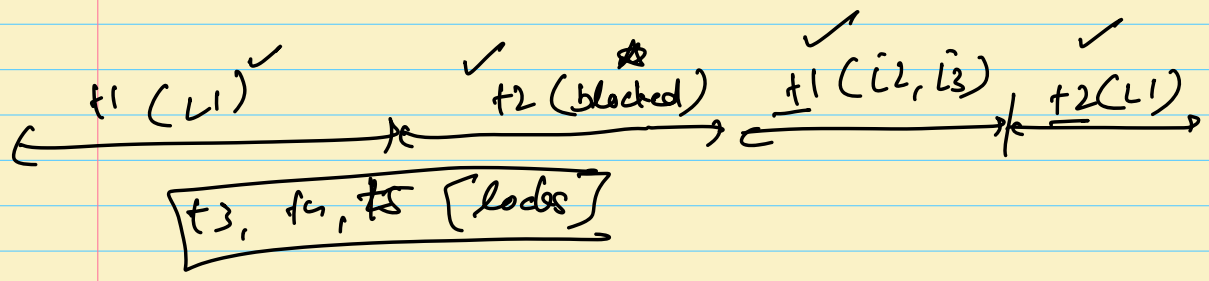
- Everything remain same as Version 6
- → Just add synchronized keyword to getInstance

7.1

class DbConnection {

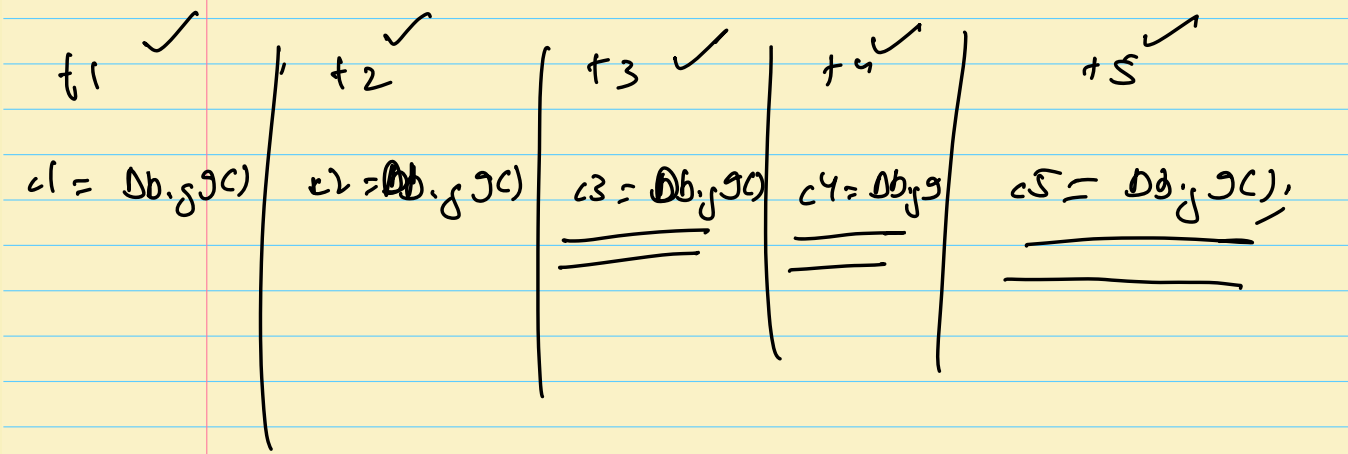
// same as version 6

```
public synchronized static getIns() {
    L1  if (inst == null)
    L2      inst = new DbCon();
    L3  return inst;
}
```



version 7.1 → A good solution, works in concurrent environment also.

Problem 7.1 → { synchronized keywords adds performance bottleneck



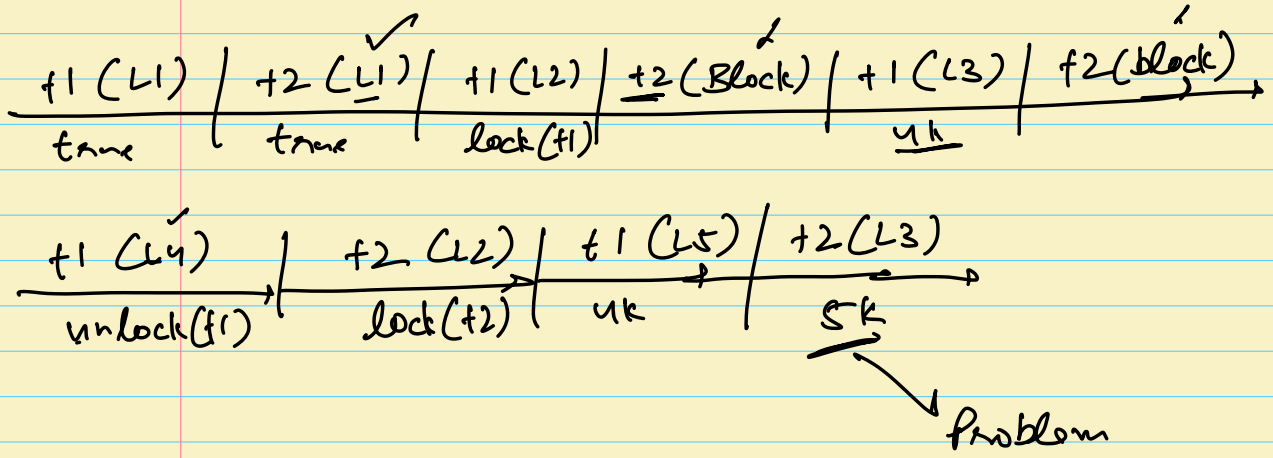
If inst is initialized, after that should we still keep threads blocking while entering the getInstance()

★ → Once inst is initialized, even if multiple threads enter the method there is no harm, infact will be better for performance.

version 7.2

```
class DbConnection {
    // version 6 variables
    // version 6 ctor
    // " " 6 inst

    public static DbConn getInstance() {
L1      if (inst == null) {
L2          lock.acquire()
C3  ✓           inst = new DbConn();
L4          lock.release()
        }
→ L5    return inst;
    }
```

t1 | t2 | t3



Problem 7.2 → Not singleton, multiple object

Version 7.3

class DbConnection {

// variable, ctor, inst same as version 6

public static DbConnection getInstance() {

L1 — if (inst == null) {

L2 — lock.acquire();

L3 — if (inst == null)

L4 — inst = new DbConn();

L5 — lock.release();

}

L6 — return inst;

}

}

t1 t2 t3 t4

t1(L1)	t2(L1)	t1(L2)	t2(block)	t1(L3)	t2(block)
true	true	lock(t1)	L2	true	

t1(L4)	t2(block)	t1(L5)	t2(L2)	t1(L6)	
UK		lock(☆)	lock(t2)	UK	

t2(L3)	Any thread	t2(L4 L5)	Any thread	t2(L6)
false		lock(☆)		UK

t3(L1)	t4(L1)	t3(L6)	t4(L6)	
false	false	UK	UK	

☆ After the instance is created, no thread even came across a lock.

Version 7.3 vs Version 7.1

Version 7.3	Version 7.1
☆ Threads deal with locks only till instance is not created	Even after instance is created, all threads acquire & release a lock

- Once the instance is created, in version 7.3, the future threads never acquire or releasing a lock.
  - Does everything as version 7.1
  - & has better

- version 7.3 is Dual check locking

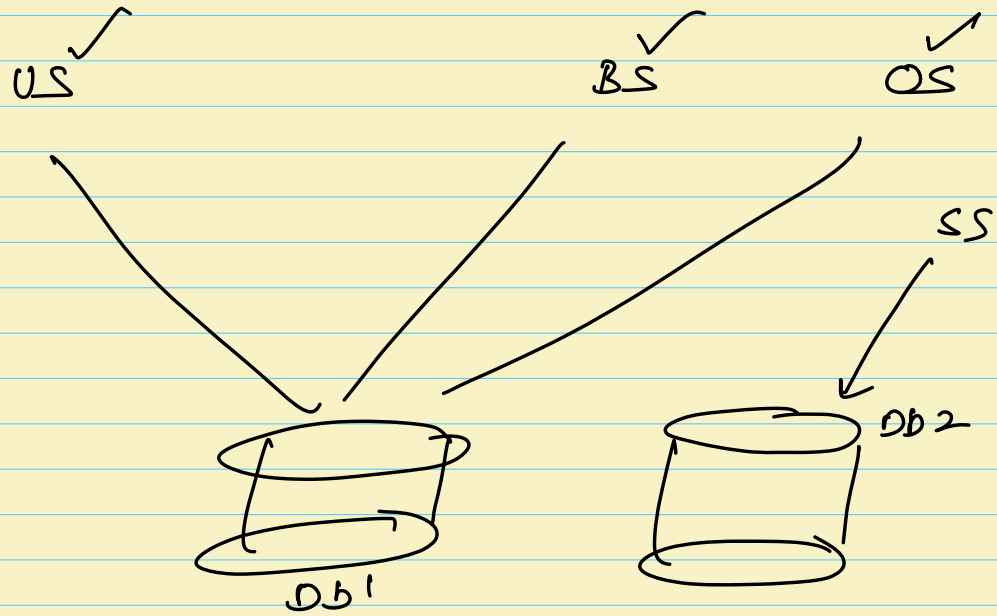
- Pros → Resource saving when there is a costly object creation for a shared resource.

- Cons → If the properties can mutate.

Use Singleton when the objects are immutable.

DbConnection ?

String url = ☆



- Code
- Interview Questions

1. Good Evening

2. We begin at 9:10 pm

3. Topic – UML Diagrams

Agenda

1. UML Diagrams
    1. Use Case Diagram
    2. Class Diagrams

→ Skills builds coding which are important

↳ [We can not write entire code ourselves]

↓

Collaboration is important btw different stakeholders

VV

→ What is important for effective collaboration?

└ Comprehensible Communication

└ Non ambiguous

└ Fast/ Effective understanding

Communication

1. Words → Emails
→ Documents
→ Zoom meetings

* Ambiguity especially for technical topics

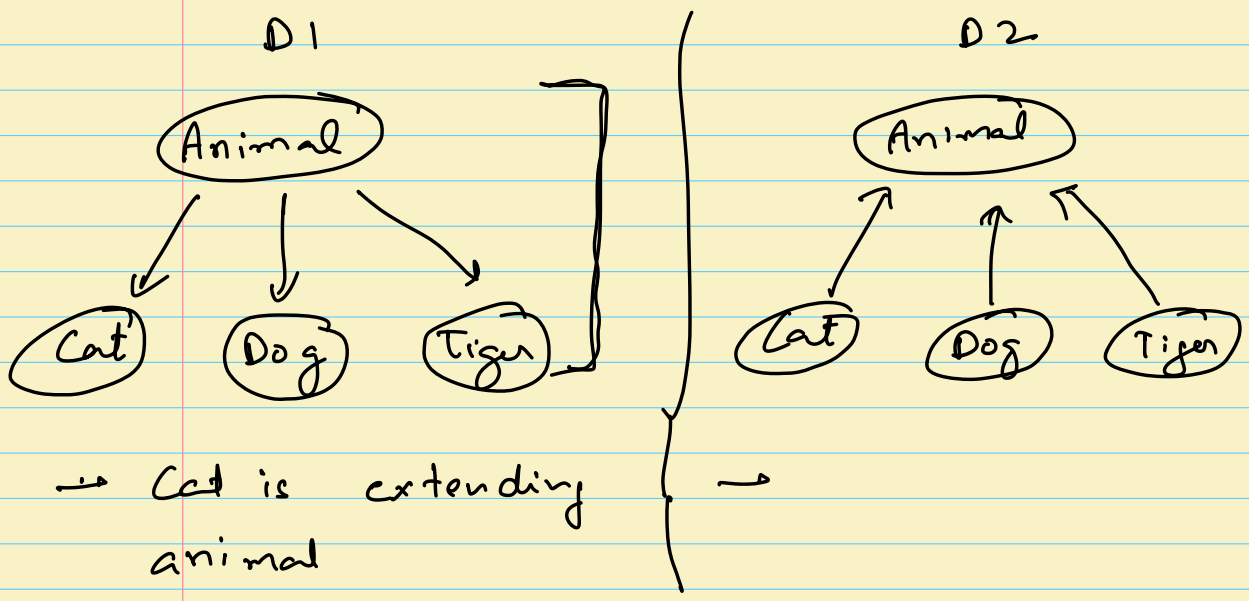
* Slow comprehension

2. Diagrams or Images

→ Faster to comprehend ✓

"A picture is worth a thousand words"

→ Images can also be ambiguous



→ Cat is extending animal →

* In absence of standardisation images could be ambigous.

→ Solution is standardising

UML → Unified Modelling Language

* Standardizes representation of diff SW concepts via diagrams.

Categories of UML Diagrams

1. Structural UML : How codebase is structured?
2. Behavioural UML : How a system works?

Structural UML	Behavioural UML
1. Class Diagram	1. Use Case Diagram
2. Object Diagram	2. Activity Diagram
3. Package Diagram	3. Sequence Diagram
4. Component Diagram	

USE CASE Diagram [Behavioural Diagrams]

→ Use case? ↙

- Mock Interviews → is a use case for Students ↑ in Scaler System →

★ Feature or a functionality = Use Case

★ Who is going to use the feature? = Actor

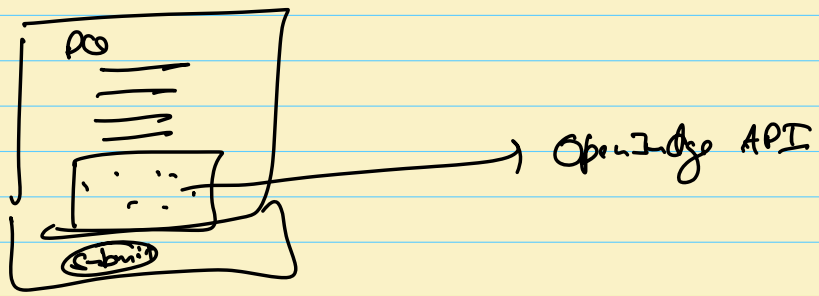
TAs, Students, Instructors, Mentor → Actors

joinClass, mockInterview, createHelpRequest → Use cases.

5 key words

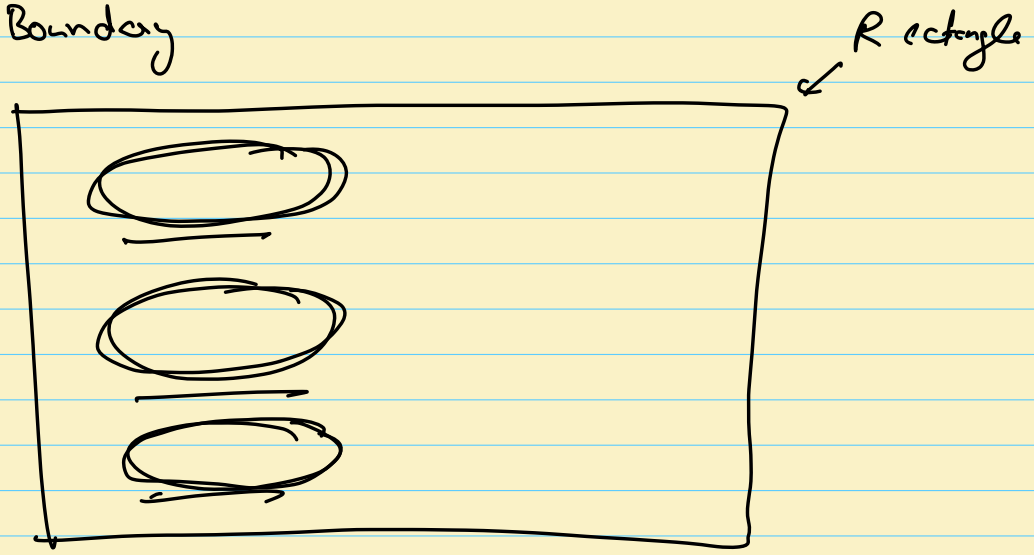
1. System Boundary

- Scala might be using RazorPay or Stripe
- Google Auth for signin
- Zoom Apis for video classes
- Open Judge Api for dashboard.



1. → System Boundary confines the scope of SW system

→ All use cases implemented by the SW are shown internally in System Boundary

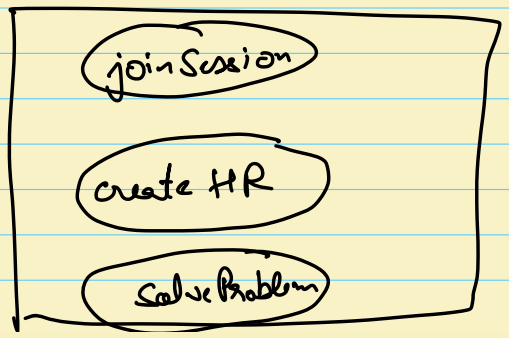


2. Use Case

→ Feature or functionality

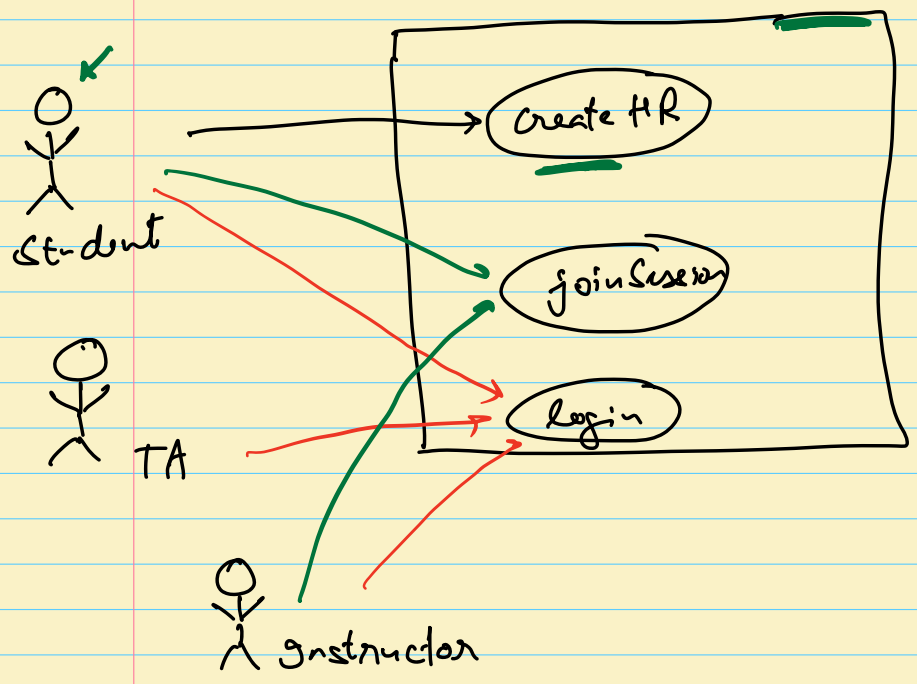
→ Verb

→ Ovals with text inside system boundary



3. Actor

1. Entity who uses a particular use case
2. Noun
3. Stick Diagrams outside system boundary
4. Arrows from actor to use cases to represent relations



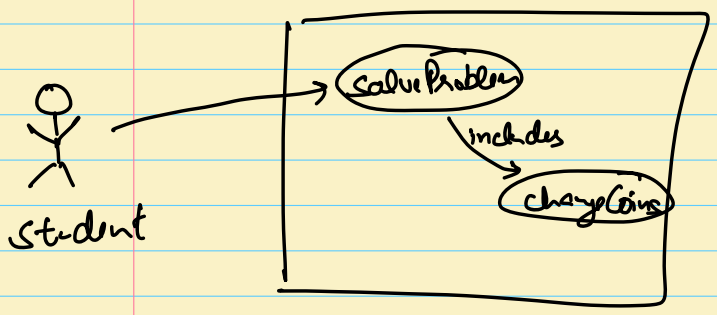
4. Includes

1. Use case is a method

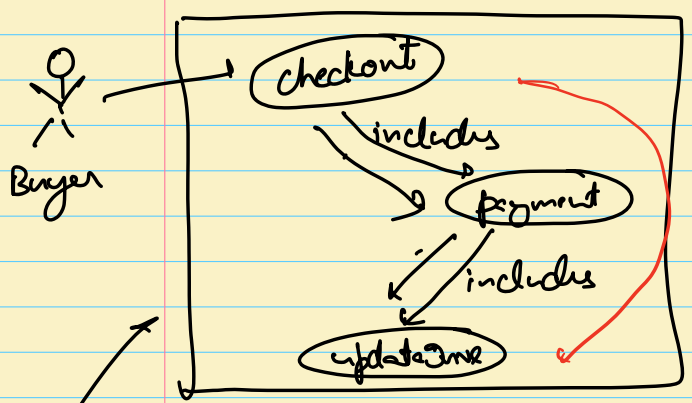
solveProblem() {

changeCoins();

}



Ecommerce



checkout() {

checkout() {

payment();

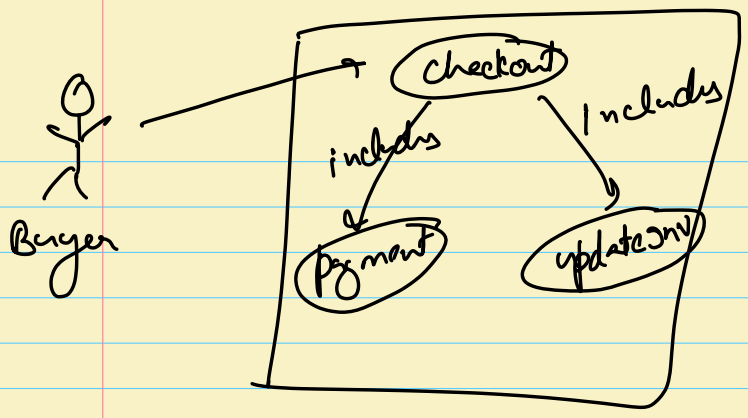
}

payment() {

updateInv();

}

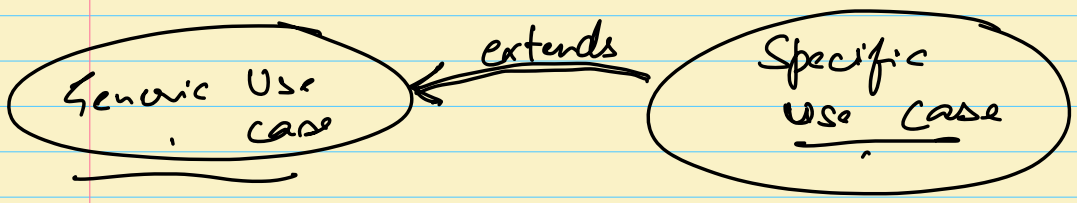
}

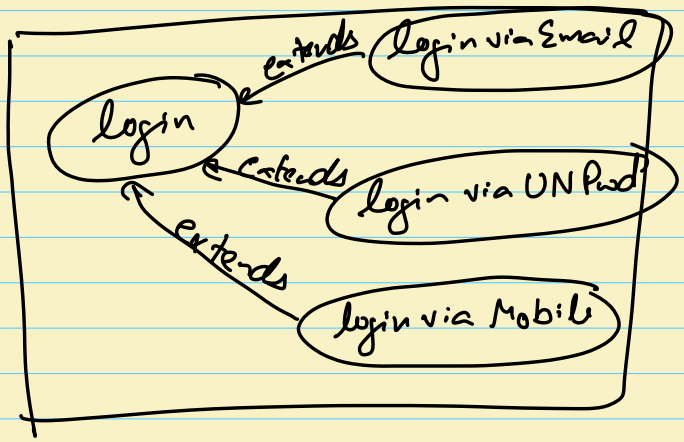


5. **Extends**

* For different variations of a use case

e.g. login → emails
login → username - pwd
login → mobile phone using OTP





Project Managers, Business Analysts, Architect

→ Classwork Assignment

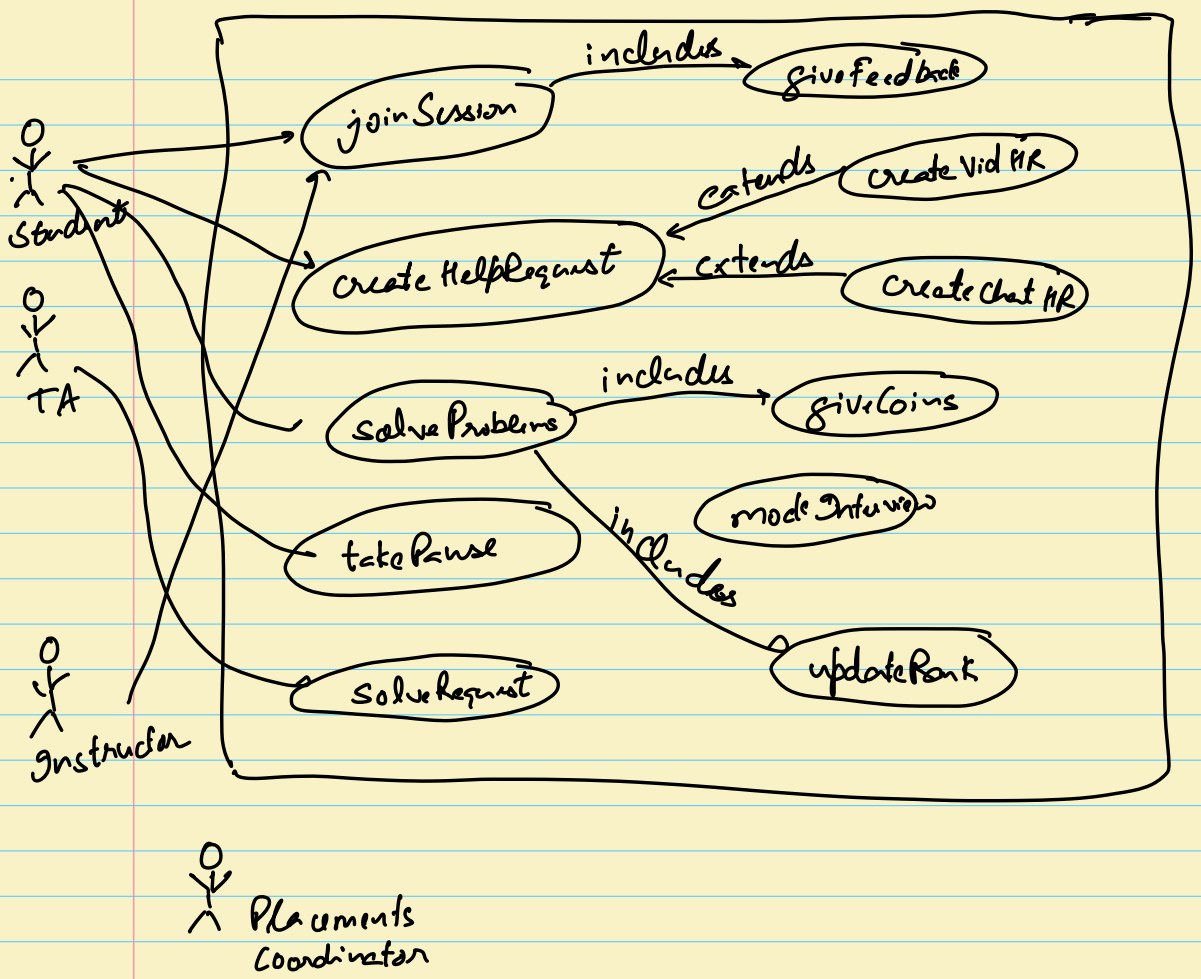
Take Scalar as a story

1. Make a use case diagram
2. At-least 5 use cases ✓
3. " 3 actors
4. " 1 include
5. " 1 extends

Break = 9:54 to 10:04

Classwork Asgn.

→ Discussion
→ Class Diagram



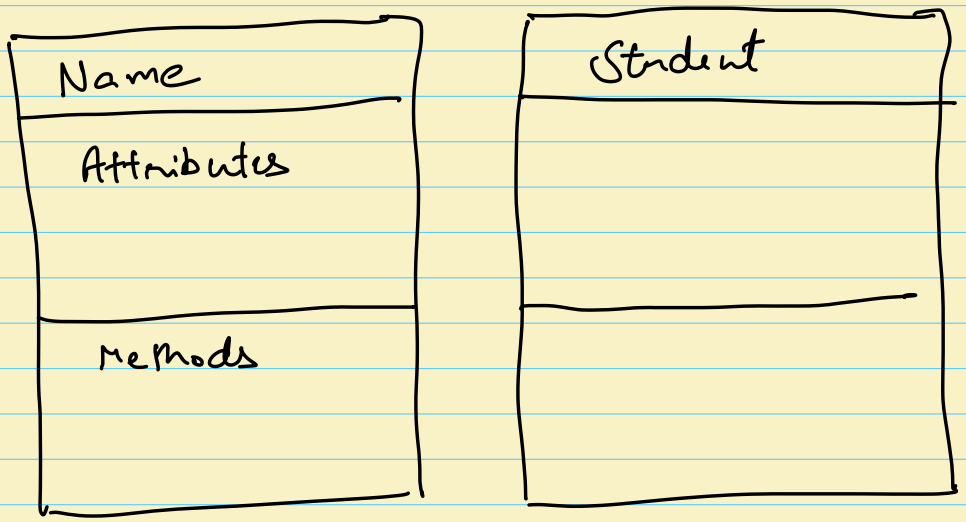
Class Diagram [Structural UML]

1. Representing entities in SW system
   - ✡ Classes
   - ✡ Interfaces
   - ✡ Enums
   - ✡ Abstract classes

2. Relations

- Who implements what?
- Who extends what?
- What is an attribute of what?

Class: Via a rectangle with 3 sections



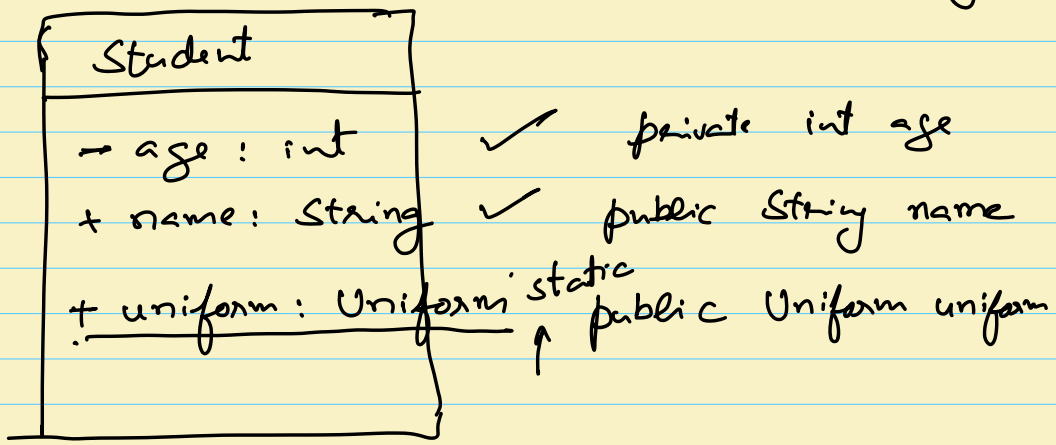
Attributes

```
class Student {

private int age;

}
```

[Access Modifier] [data type] [name]

- private
+ public
# protected
~ default

In the diagram = [AM] [name] : [Data Type]



★ Represent static by underline

★ " final by caps

??

Verify

→ Methods

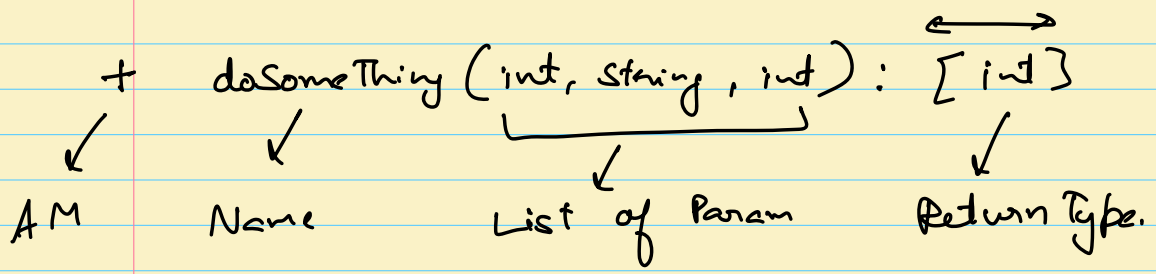
class Student {

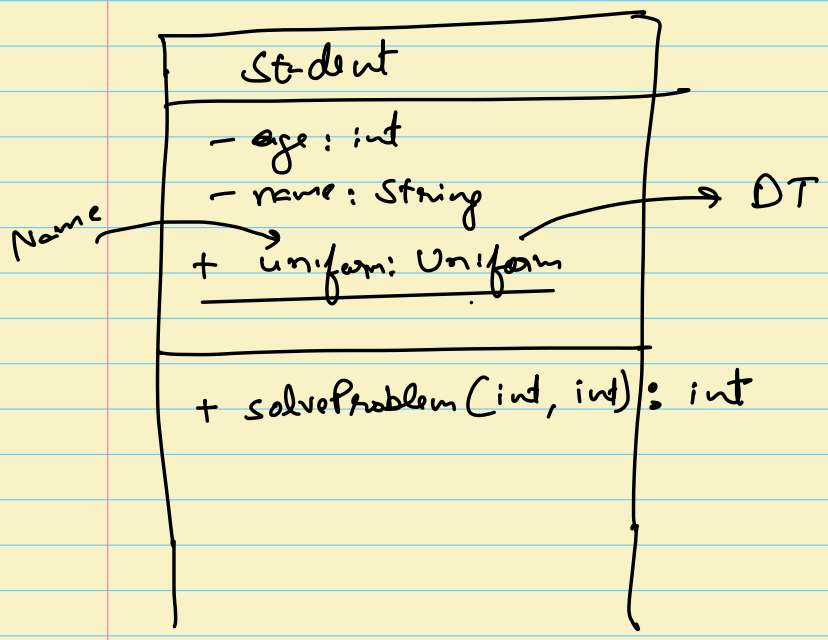
public int solveProblem (int qid, int q[]){

}

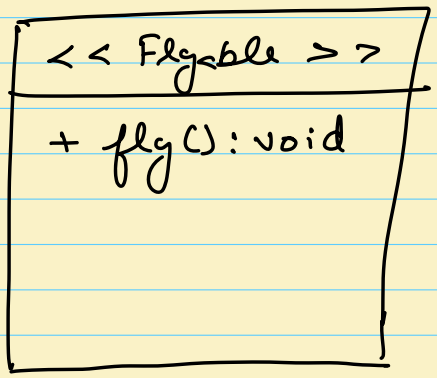
[Access Modifier] [Return Type] [Name] [list of params]

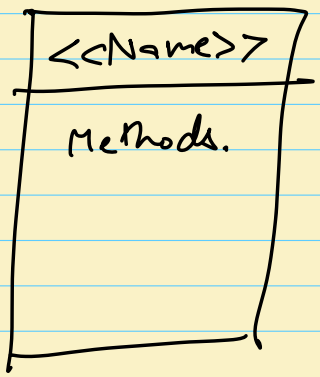
On a diagram





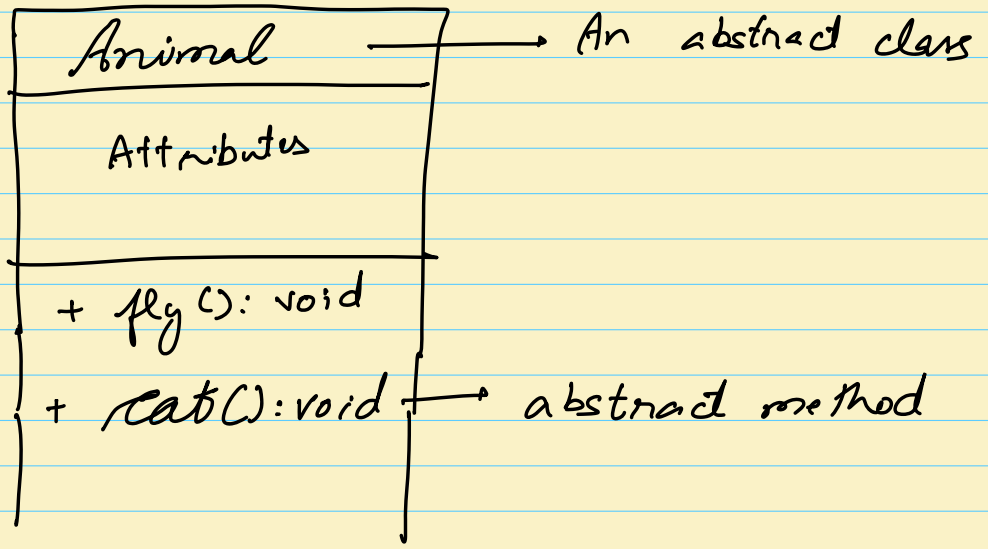
Interfaces





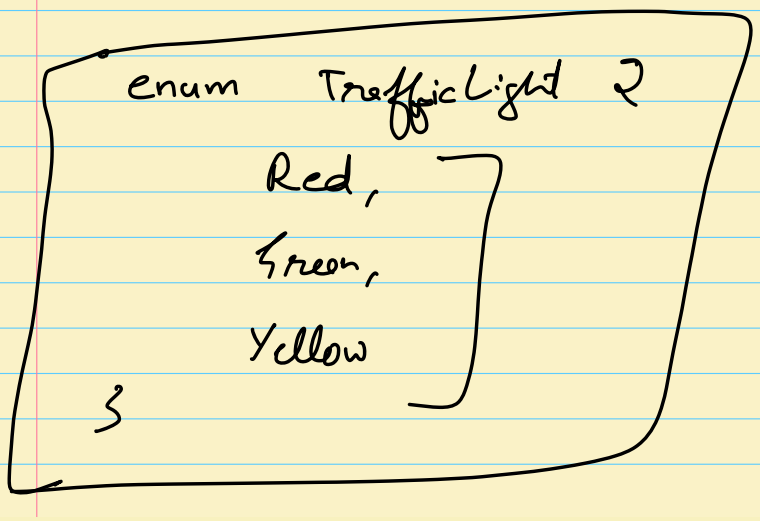
Abstract Classes

- abstract fns. → Italics
- Non-abstract fns. → Normal

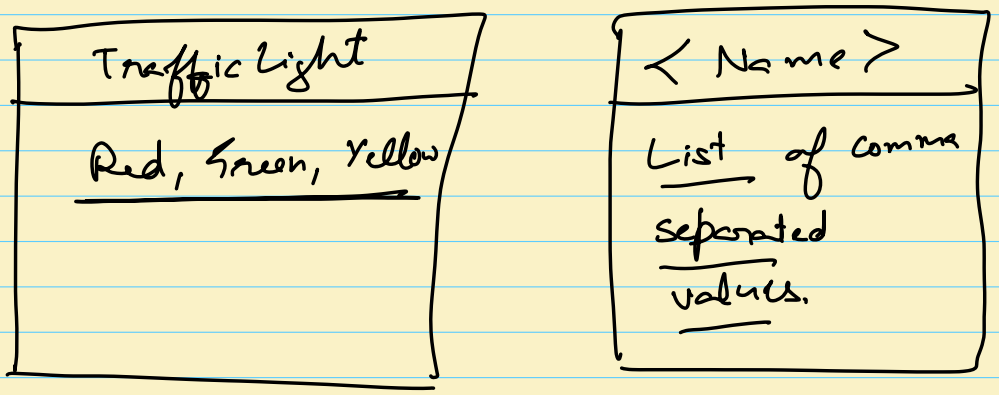


Enums : Fixed set of constant values.

- Days of weeks
- Months of Year



→ TrafficLight tl = TrafficLight.Red;

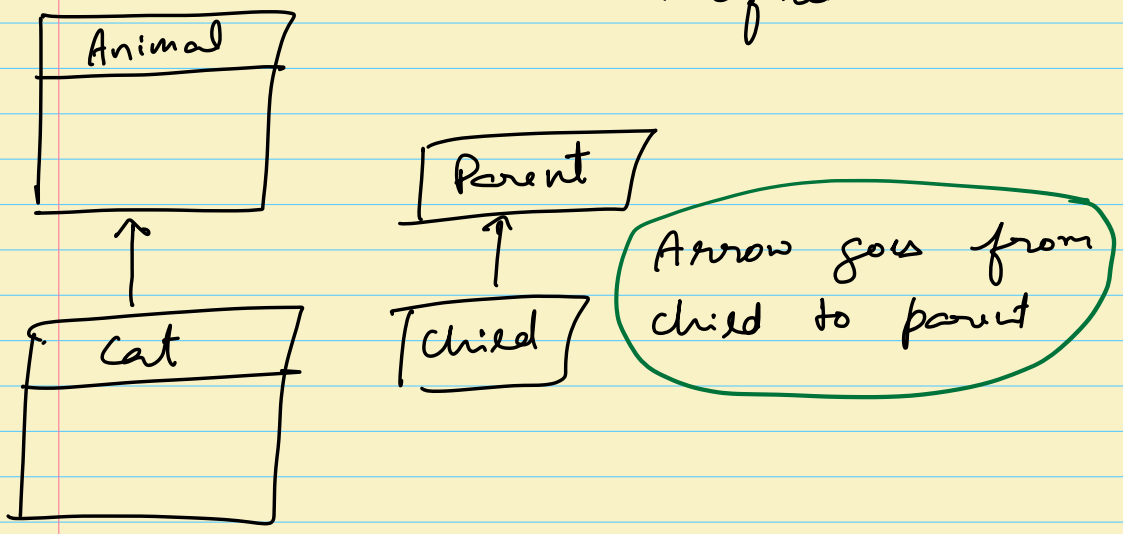


Relations

└ 1. IS-A relation : Inheritance.

1. Extends : Parent-Child

2. Implements : Class implements an interface



2. Has-A relation = Association

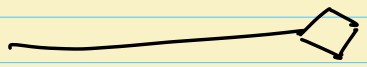
class Car ?

Engine eng

class Batch ?

List <Student> students;

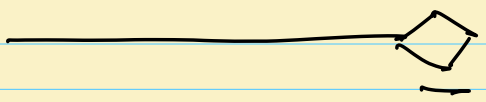
Container contains contained



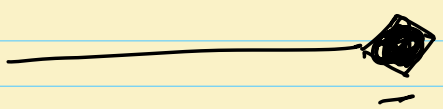
1. Arrow with diamond head
2. Direction is container to contained.

Has-A [Association)

1. Aggregation [Weak Association)



2. Composition [Strong Association)



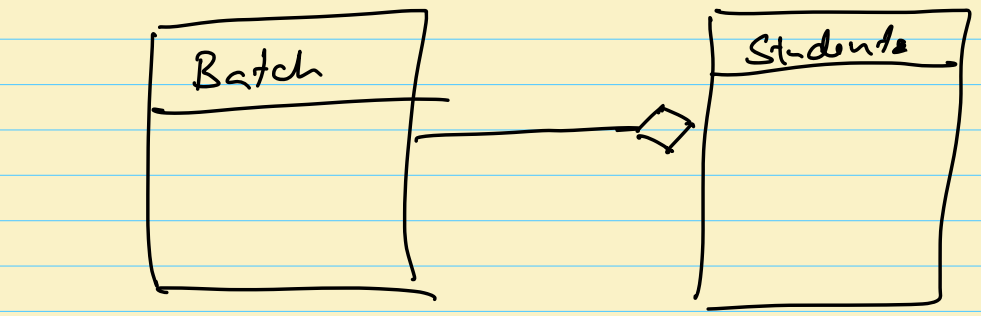
Aggregation [Weak Association]

→ Contained can exist independently of container

class Batch {

✓ List<Students> list

}

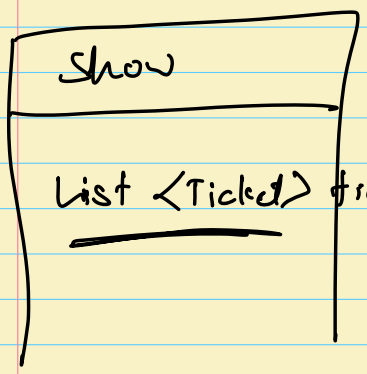


☆ Students can exist without a batch.

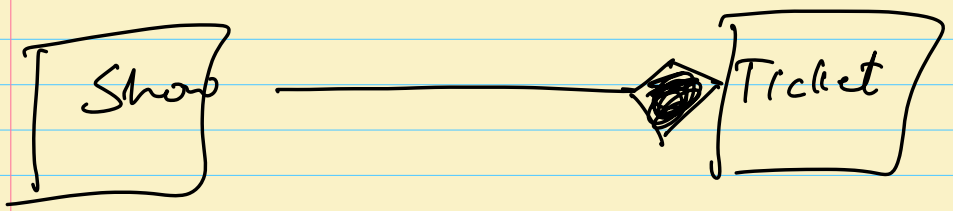
↳ Weak Association.

Composition [Strong Association]

→ when contained has no meaning without container.



* Tickets has no meaning without the show. [Strong Association]



Summary

1. Collaborate
2. Communicate
3. Images are better than words
4. ∞ faster comprehension.

5. But images can be ambigous without standardisation.

6. UML

HeadFirst Design Patterns

↳ Read it like a story book

Blackwasp.co.uk

---

7. Catgories
   - Structural
     1. Class Diag ✓
     2. Object "
     3. Pkg "
     4. Comp "
   - Behavioural (15 min)
     - Use Case ✓
     - Activity (15 min)
     - Sequence (15 min)

8. Use case Diag
   - System Boundary
   - Use Case
   - Actor
   - Includes
   - Excludes

<< >>

9. Class Diagram
   - Class, Abstract class, Interface, Enum
     - Italics for abstract
     - underline for static