Agenda.

→ Interface
→ Abstract Class
→ Static keyword.

⇒ Class. ⇒ Blueprint of an Entity

↳ Attributes & behaviours.

⇒ A Concept based up only on behaviour.

⇒ Anything that can eat, walk & run is an Animal

behaviours.

⇒ interfaces. ⇒ Blueprint of behaviours.

```
interface Animal {

    void eat();
    void walk();
    void run();

}
```

Method declaration.

2

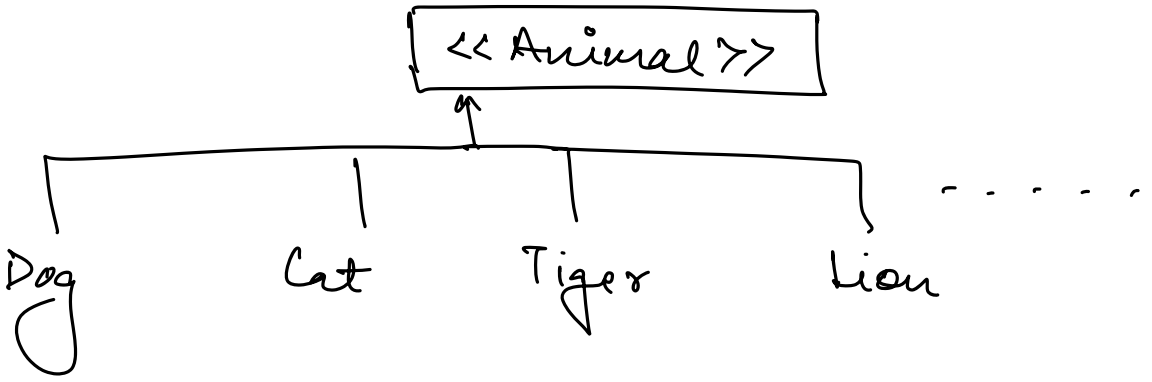
```
Class Dog implements Animal {

    void eat() {
        print("Dog is Eating");
    }
    void walk() {
        print("Dog is walking");
    }
    void run() {
        print("Dog is running");
    }
}
```

⇒ A class which implements an interface, is forced by the compiler to implement all of its method.

⇒ `Animal a = new Animal();` ✗

⇒ We can't create the object of interface as it is incomplete.

⇒ `Animal a = new Dog()` ✓




```
Animal a = new Dog()
           new Cat()
           :
           :
```

⇒ Stack : A LIFO datastructure with 2 madatory functions

→ push(x)

→ pop()

⇒ Multiple ways to implement Stack.

```
interface Stack {

    Void push(int x);
    Void pop();

}
```

```
Class ArrayBasedStack implements Stack {
    int a[10];
    int top = -1;

    void push(x) {
        top++;
        a[top] = x;
    }
    void pop() {
        ==
    }
    int size() {
        ==
    }
    bool isEmpty() {
        ==
    }
}
```

=> Compiler forces us to provide the implementation of all the behaviours in a class that is implementing it.

Class LLBasedStack implements Stack {

Node head

void push(x) {

||LL

}

void pop() {

}

}

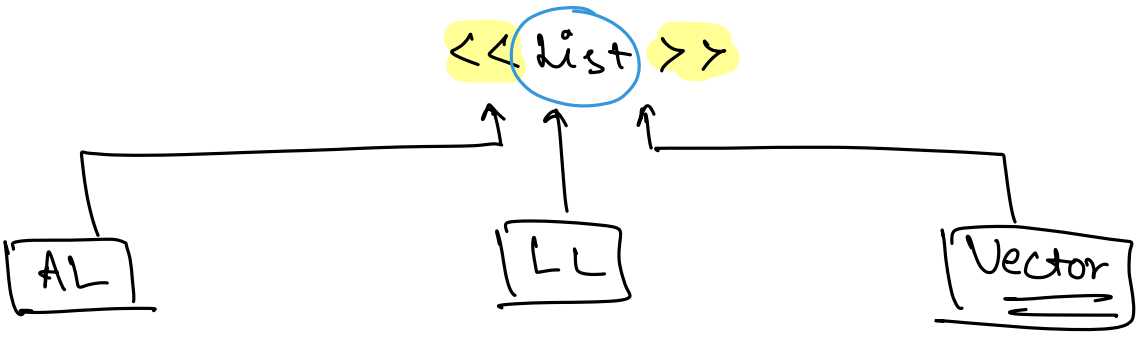
⇒ Stack<Int> st = new ~~ArrayBasedStack();~~

LLBasedStack();

st.push(x)

⇒ List<Int> list = new ~~ArrayList<>();~~

LinkedList<>();



interface List {

add(—);

remove();

get();

}

AL implements List {

// Dynamic Array to implement list methods.

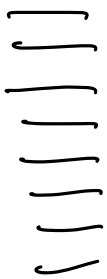
}

⇒ AL<Int> al = new ~~AL<>~~();

LL<>()

X

⇒ List<int> l = new ~~AL<>()~~ LL<>();



⇒ PHONEPE.

```
Class PhonePe {
    YB yb = new YB();

    SendMoney(A,B,x){
        yb.transfer(—);
    }

    CheckBalance(—){
        yb.checkBalance(-)
    }
}
```

```
Class YesBank {

    Void checkBalance(){

    }

    Void transfer(A,B,x){   // from to amount

    }

    Void registerAcc(=){

    }
}
```

⇒ YB ⇒ ICICI

Class PhonePe {

~~YB yb = new YB();~~

ICICI ic = new ICICI();

SendMoney(A, B, x){

ic ~~yb~~.transfer(~~A,B,x~~)(x, A, B)

}

CheckBalance(—){

ic ~~yb~~.~~checkBalance~~(-)
getBalance

}

}

Issues:

→ lot of development effort.

→ lot of testing

⇒ Tight Coupling

Class PhonePe {

~~YB yb = new YB();~~

BankApi bank = new ~~YB();~~

DCICI();

bank.checkBal(—);

bank.transfer(—)

———

———

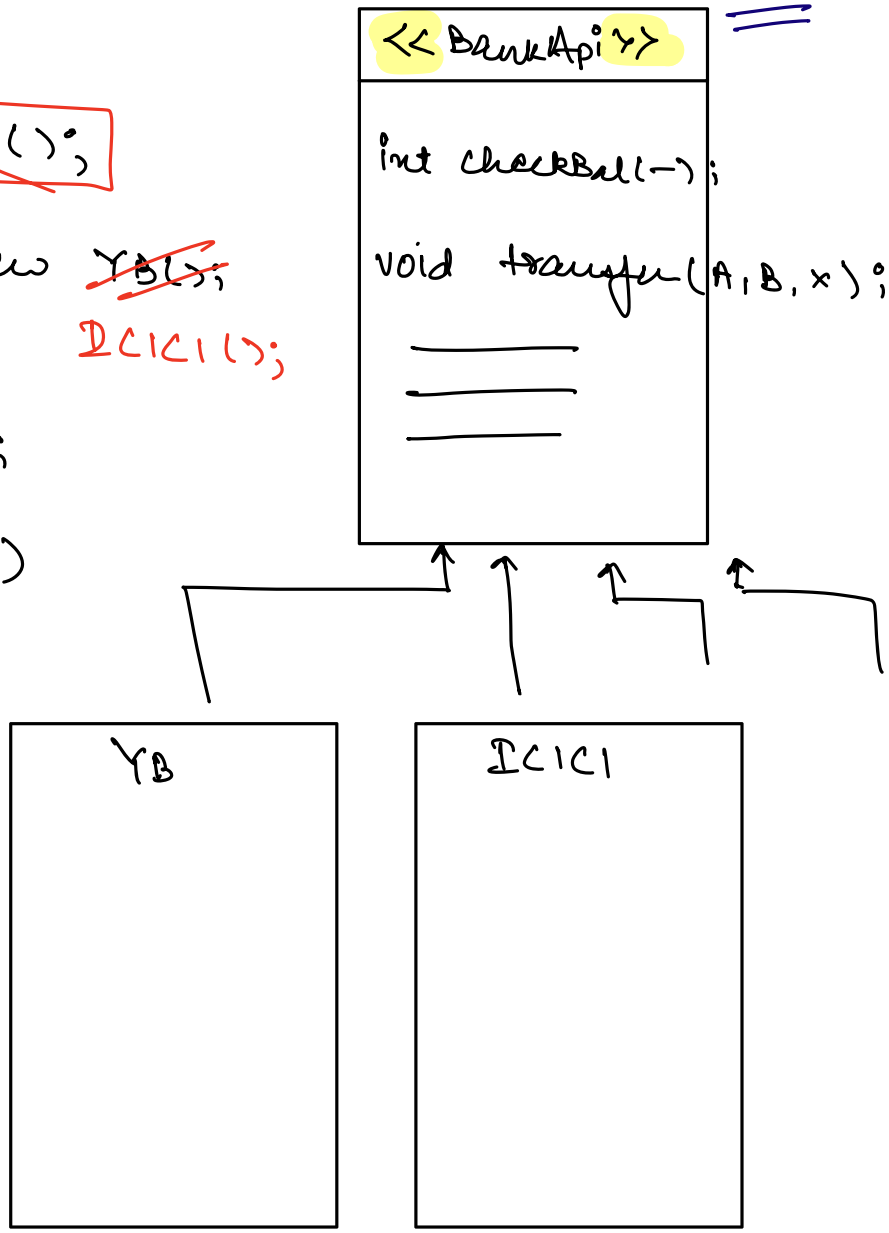
———

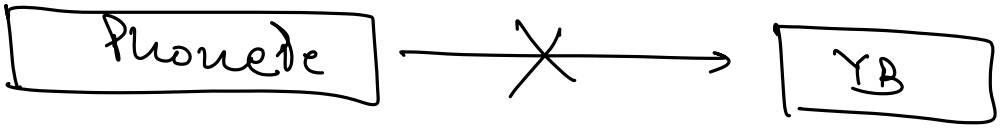
———

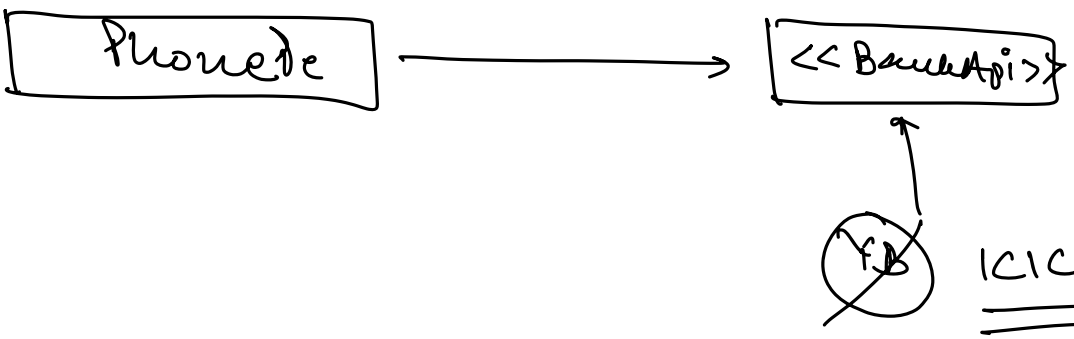
3

⇒ loosely Coupled.

RBI.







Principle : Program to interface, Not to implementation

⇒ Multiple ways of doing something

⇒ INTERFACE.

<<Serializable>>

interface Serializable {

}

ABSTRACT CLASS

⇒ Entity (Can have attrs)

⇒ Can also have behaviours.

⇒ We don't have 100% Clarity on all the behaviours.

```
Abstract Class Animal {
        String name;
        int age;
        void eat(){
            print("Eating");
        }
        abstract void walk();
        abstract void run();
}

Class Dog extends Animal {
        void walk(){
            print("Dog is walking");
        }
        void run(){
            print("Dog is running")
        }
}
```

```
Class Cat extends Animal {

    Void eat() {
        print("Cat is Eating");
    }                                   } Overriding

    void walk() {
        print("Cat is walking");
    }

    void run() {
        print("Cat is running")
    }

}
```

⇒ Q. Can we create an object of Abstract class?

⇒ No

⇒ Abstract Class is also incomplete.

⇒

```
Abstract Class User {
    ---------
    ---------
    ---------
    ---------
    No abstract method
}
```

No one should be allowed to create User Object.

Interface ≡ Abstract class with No attrs & No non-abstract methods.

⇒ STATIC.

```
class Client {
    public static void main(){
```

public → Access Modifier

void ← return type

main() ← Name.

≡

⇒ Client.main()

⇒ We call a method on an object.

⇒ Static → Attr
→ Method
→ Class. ⇒ Builder. Design Pattern.

St1.changeSp(+10)

St2.changeSp(-10)

⇒ Static Method : We don't need an object of a class to call the static method.

Belongs to a Class.

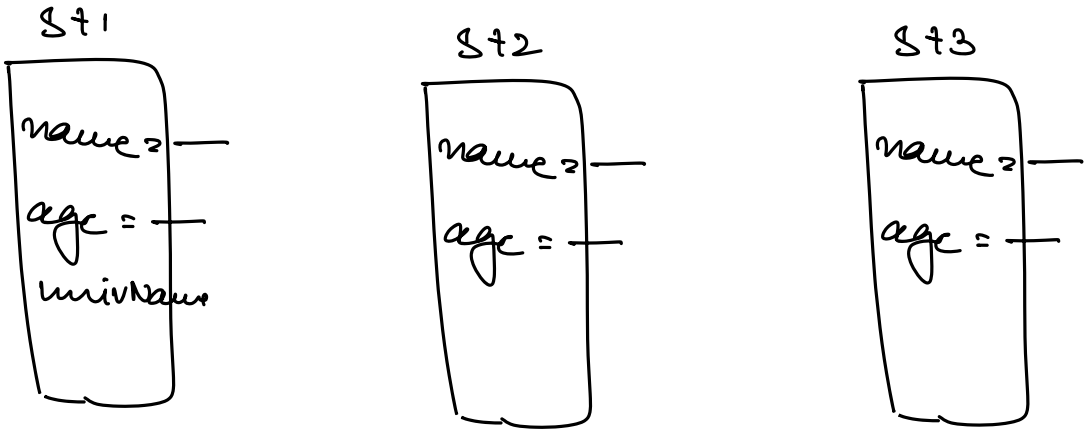
⇒
```
Class Student {
    String name;
    int age;
    static String univName = "Scaler";
}
```

final

Belongs to a Class.

```
Student st1 = new Student();
Student st2 = new Student();
Student st3 = new Student();
```

univName → Scaler



⇒ Objects are created runtime.

⇒ Static fields are initialized at startup time.

⇒ Can Static method access non static attrs?

NO.

Static method → load time

non static attrs → runtime.

⇒ Static methods can only access static attrs.

⇒ Lot of static members/attrs will increase the load time of our Application

——— * ———