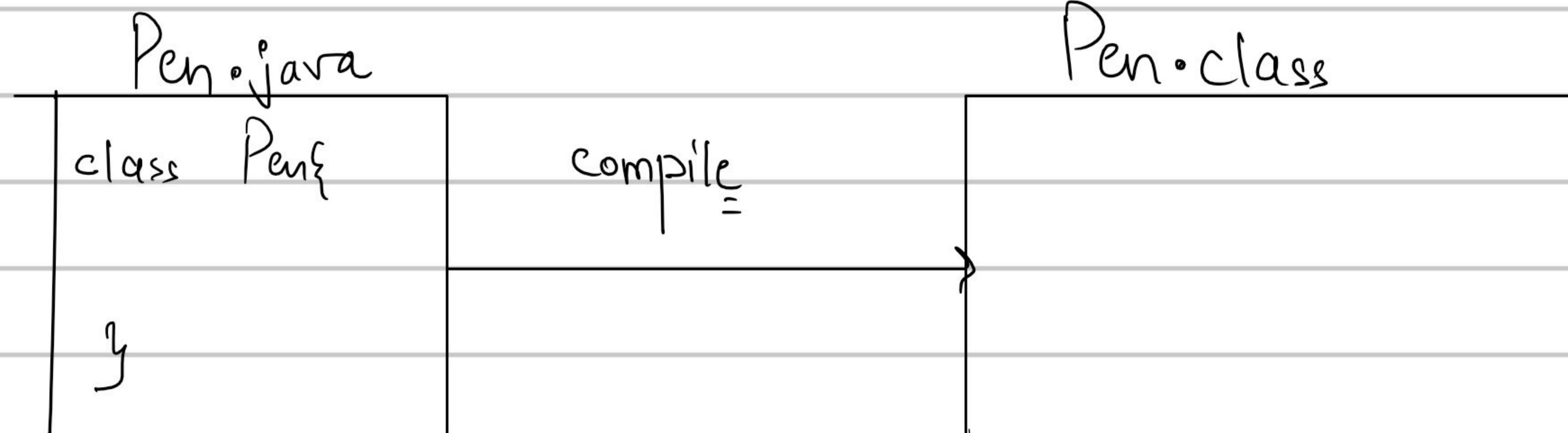


Everything in Java
is an object.

Java

object oriented
programming language.



Components of Java Environment.

JDK → Java Development Kit

JRE → Java Runtime Environment

JVM → Java Virtual Machine

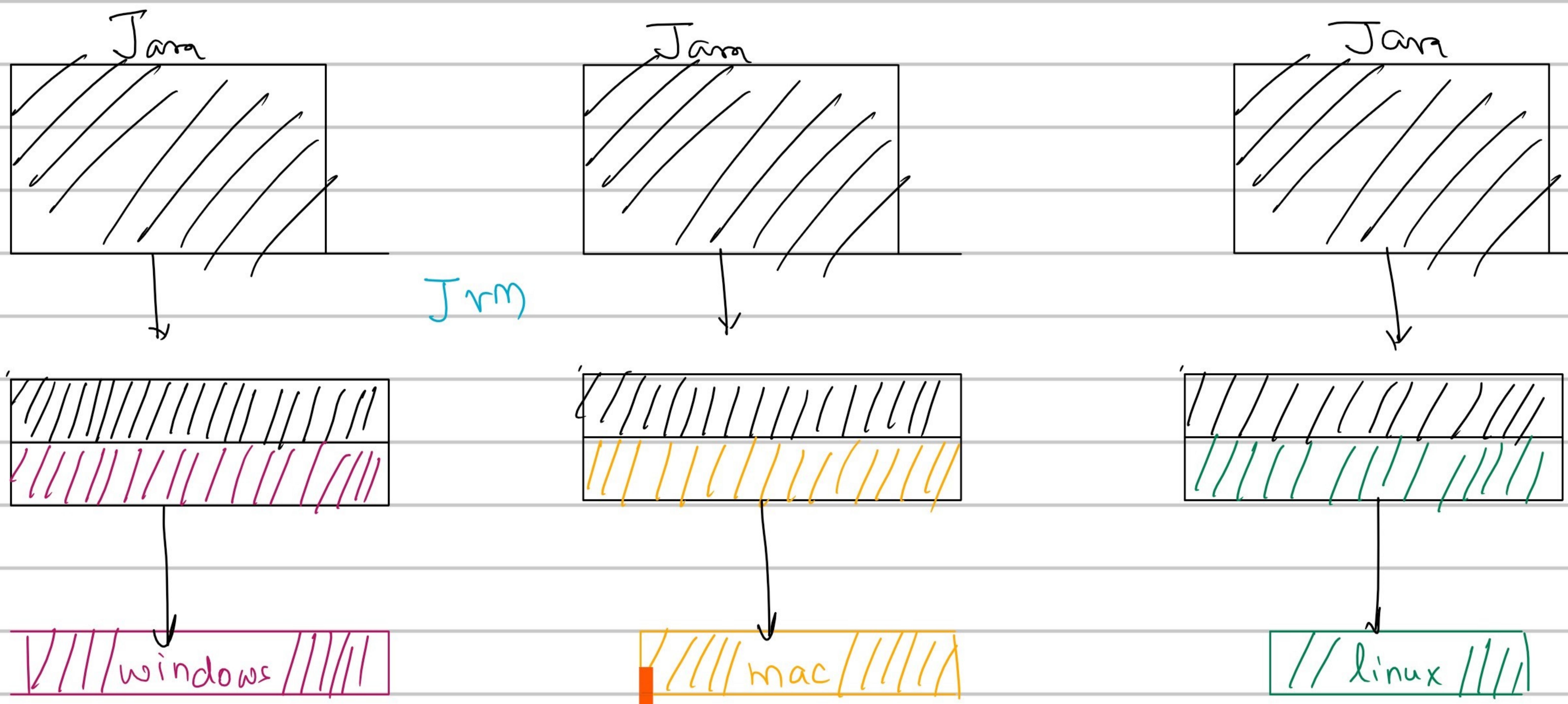
virtual machine
↓

a machine
that runs
inside another
machine.

JDK → Java Development Kit

Java Runtime Environment

JVM



Java is Platform INDEPENDENT
because
JVM is Platform DEPENDENT!!

Simple HelloWorld program

HelloWorld.java

Name of the class

public class HelloWorld {

 public static void main (String [] args) {

 System.out.println ("Hello World");

where
application
get
started

}

Class / Interface

→ One Two Three Four

Methods / Variables

}

Pascal Casing

Camel Casing → oneTwoThreeFour

Java supports 8 data types

Name	value it holds	bytes	range
byte	number	1	± 127
boolean	true/false	2	true or false
char	'a', 'b'	2	0 to 65535
short	number	2	± 32767
int	number	4	± 2147483647
long	large number	8	± 9223372036854775807
float	real	4	3.4×10^{38} to 1.1×10^{-45}
double	real	8	1.7×10^{308} to 1.9×10^{-324}

Control statements

Operators

Loops

Arrays

Arrays in Java are objects who has a container inside it which can hold elements of a similar type.

type → primitive data type / class

DECLARE

```
type [ ] nameOfArray;
```

```
int [ ] iArray;
```

```
Pen [ ] pArray;
```

CONSTRUCT

```
nameOfArray = new type [size];
```

```
iArray = new int [6]
```

```
pArray = new Pen [3];
```

INITIALIZE

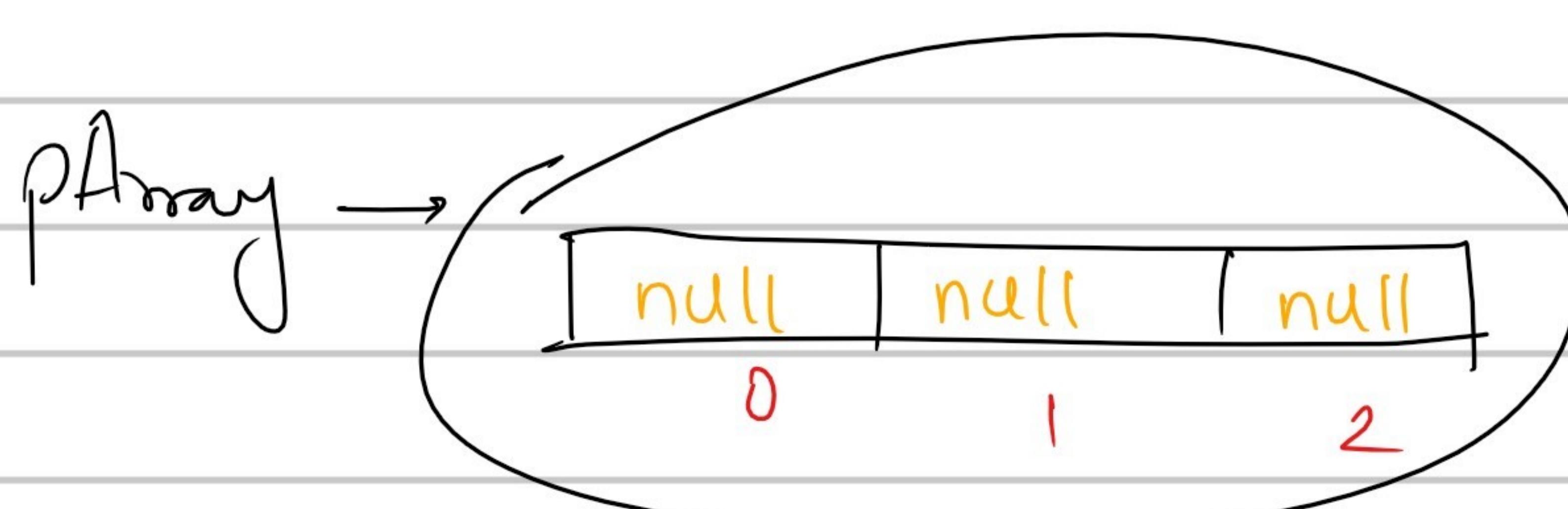
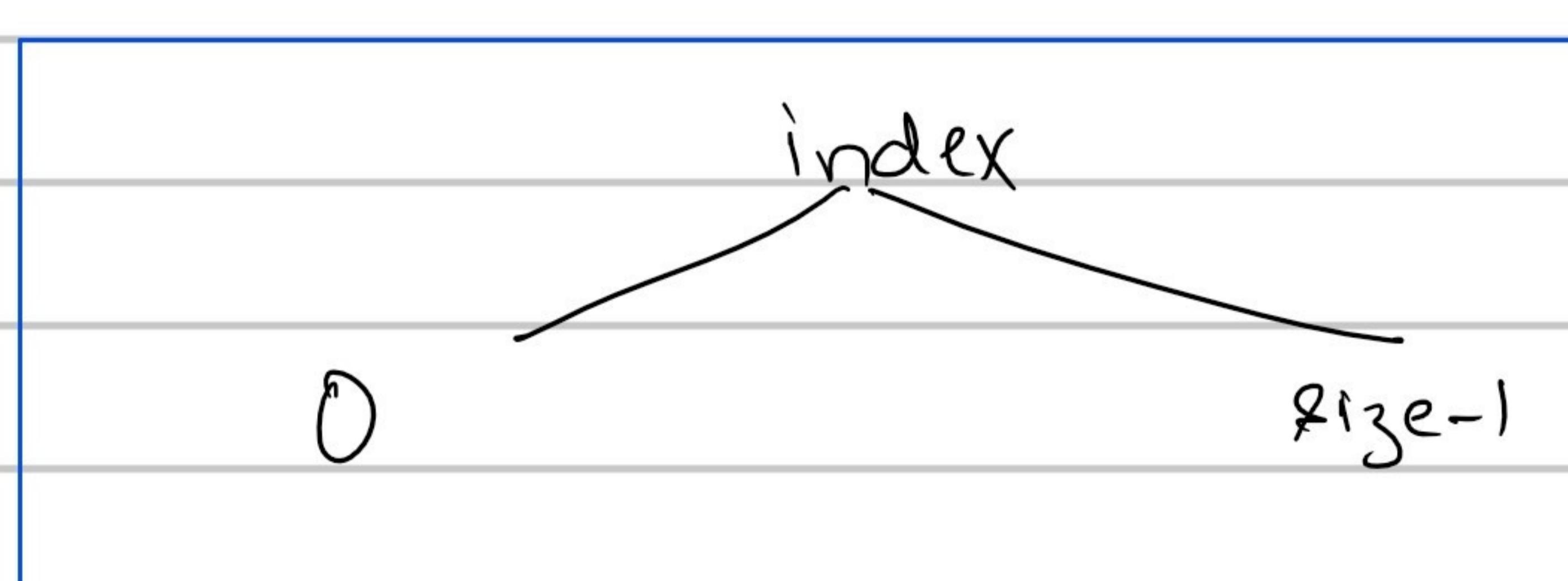
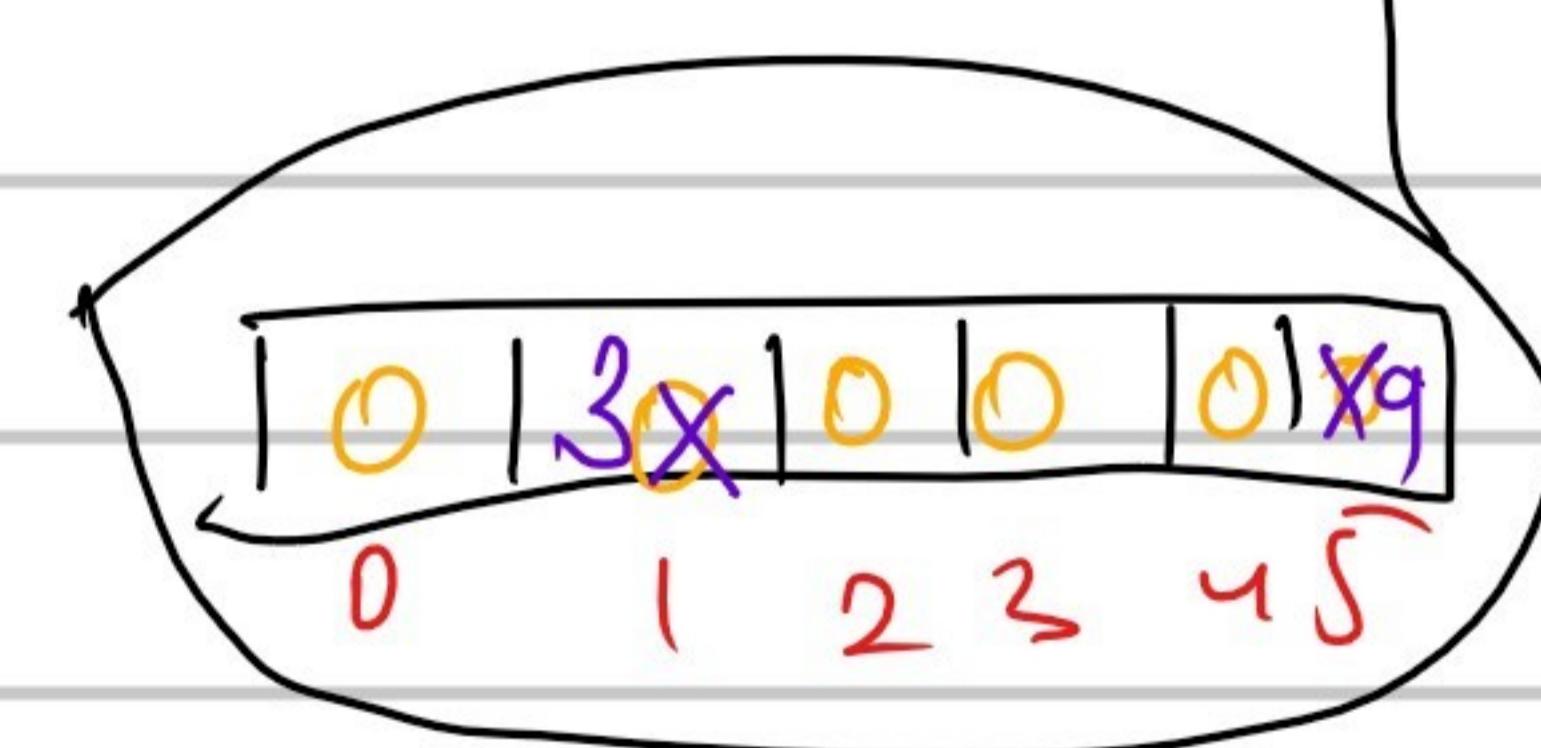
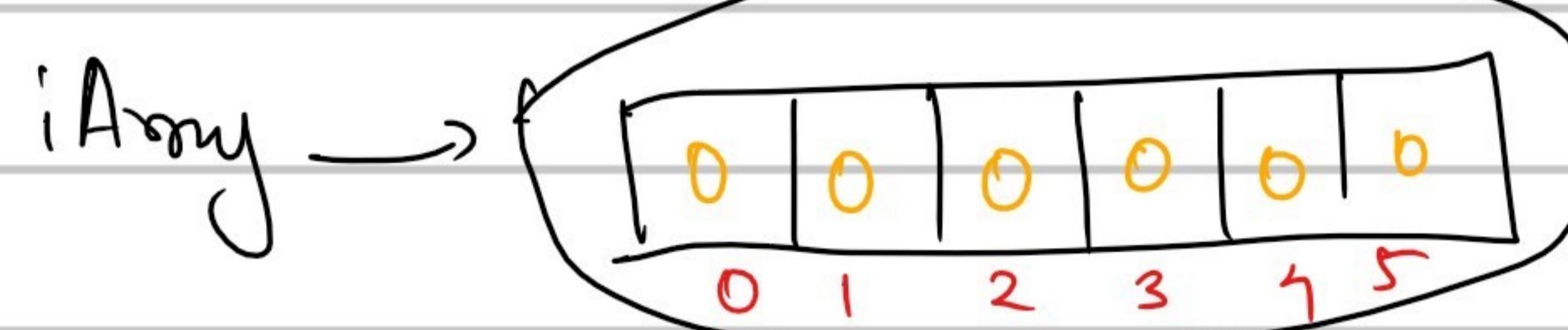
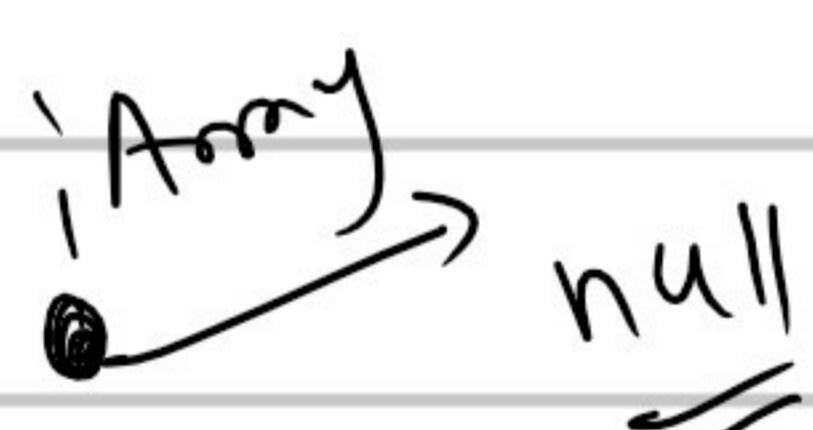
```
nameOfArray [index] = value;
```

```
iArray [1] = 3;
```

```
iArray [5] = 9;
```

```
iArray [7] = 20;
```

Exception



Length of array.

int len = iArray.length;

Initialized

nameOfArray[index] = someValue

Use

int x = nameOfArray[index]

s — (nameOfArray[index])

* Software Objects *

→ Object Oriented Language

→ Objects which replicates real world entity.

→ tangible or intangible.

ability to be measured

Pen

Car

Oxygen level
Air Pressure

→ Anything which has magnitude can be considered as a software object

All the software objects have 2 basic characteristics.

① What it has/have? →

STATE

primitive data
Objects with Has A relationship.

② What it does? →

BEHAVIOR

methods

Constructional

Operational

* Class *

Classification

→ a **template** or a **blueprint** which is used to describe an object.

→ Class is the place where working of an object is defined.

→ An object will support only those things which are defined inside it's class

NOTHING LESS

NOTHING MORE

EAGLE cannot SWIM

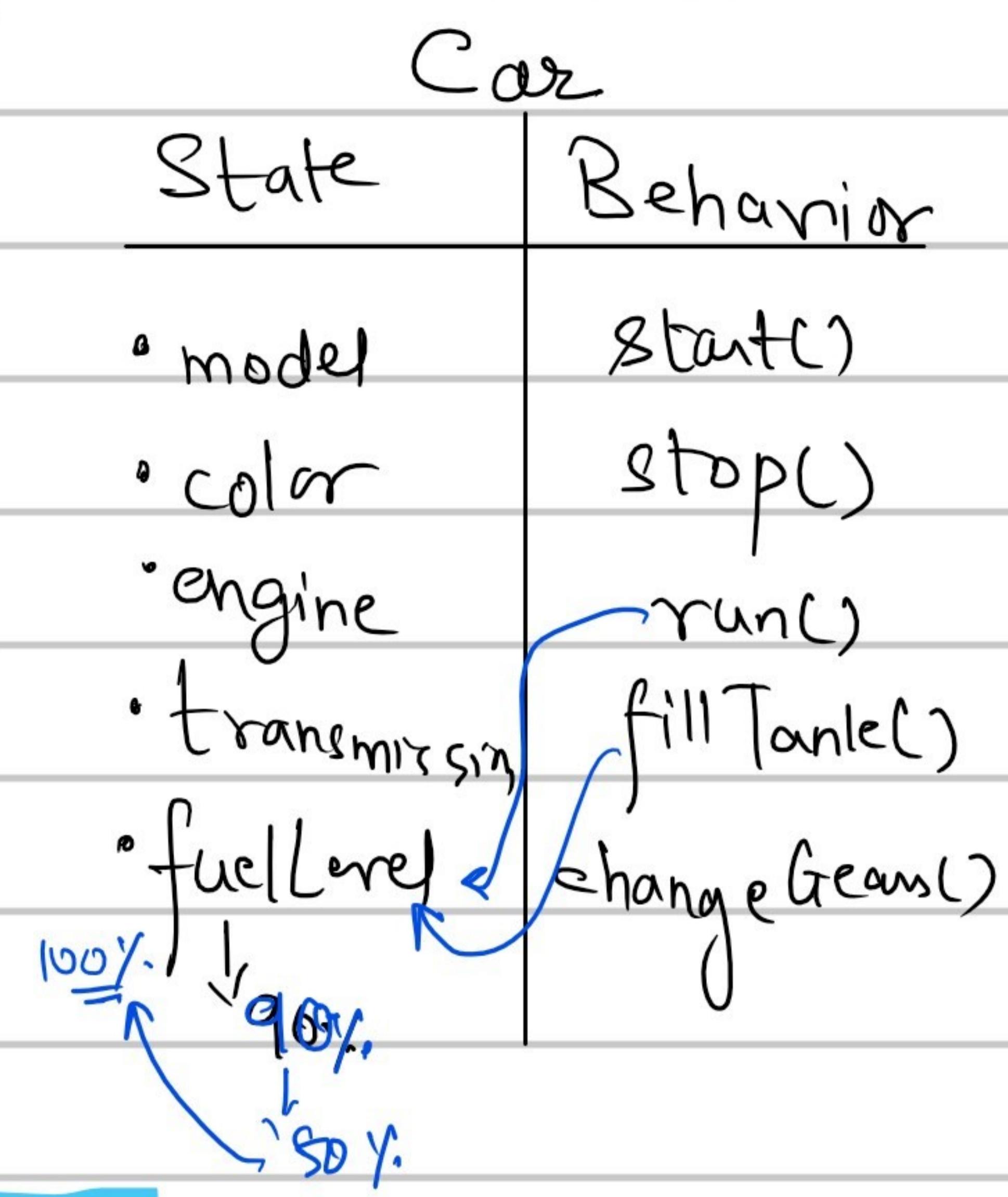
DOLPHINS cannot FLY

* State & Behavior are connected to each other *

Behavior acts on the State
of an object & changes it
from one point to another.

run() acts on fuelLevel of Car;

changes the fuelLevel from
90% to 50%.



E-commerce

- * ~~entity~~
- * product
- * Category
- * customer
- * orders
- * seller
- * order_details

Class, Interface → Pascal Casing → OneTwoThreeFour

Methods, Variables → Camel Casing → oneTwoThreeFour

class NameOfClass
{ variables;

| methods

Constructor

block

{ inner class

}

class Pen
{ int inkLevel = 90; ✓

public void write() {
 =

}

Object

reference of class
example of a class.

ClassName → objectName;

p is an object of Pen

String s;

Pen p;

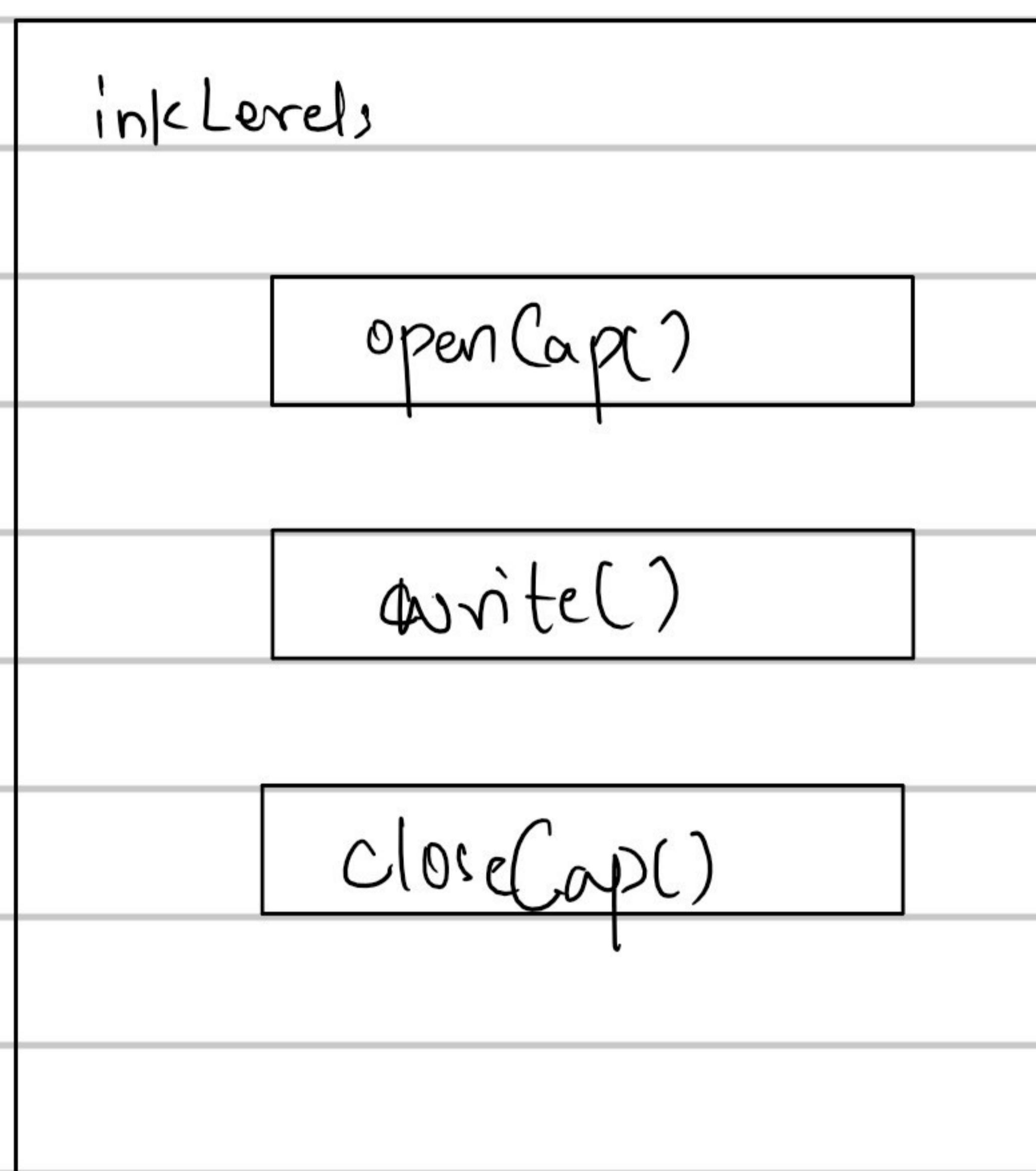
Can c;

c is an
object of Can.

Objects in Java are not operational on declaration; they need to be instantiated.

Instantiation → new + Constructor

Pen



Pen P; object

p.write(); X

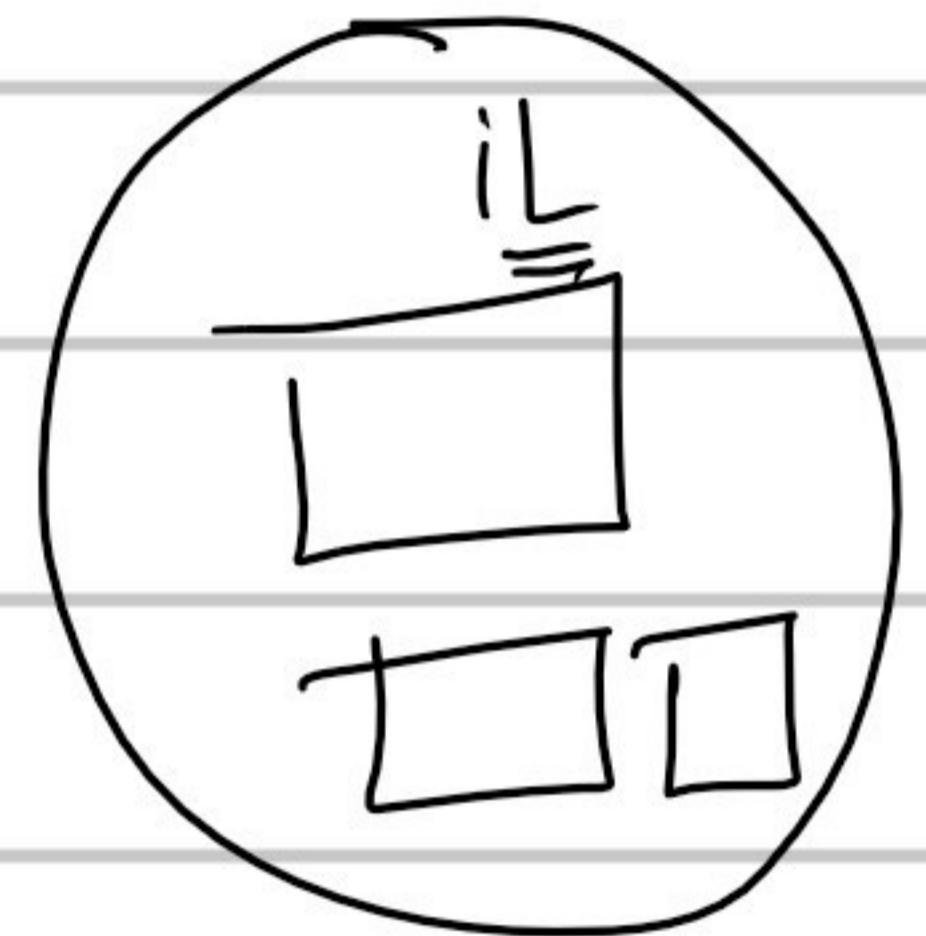
↳ object
is not instantiated

P = new Pen();

instance ↪

Objects are not operational
instances are

Memory Graph



Behavior

methods

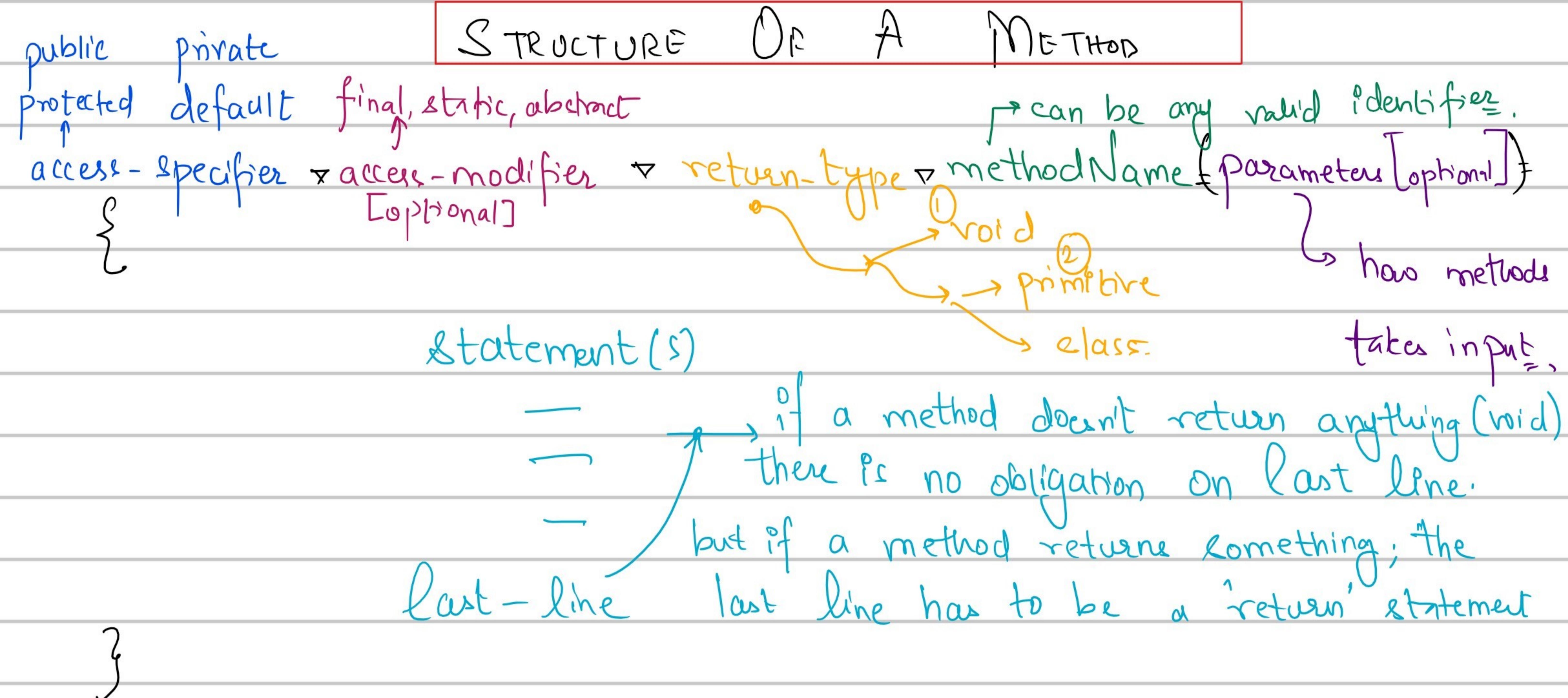
Constructional

Operational

Methods

'Code On Demand'

Collection of statements which are grouped under a single name & performs a specific task.



class MyMethods {

public void display() { } No Parameter No Return Type

No Parameter
Return Type

private int getTaxRate() {
return 18; }

By — ("LCB");

public void add (int a, int b) {
int c = a + b;
{ } (c); }

Parameters
No Return Type

{ }

Parameter
Return Type

public int square (int x) {
return x * x; }

{ }

Invoking the Method

To invoke a method; we need an instance of a class & then make use of • (dot operator)

```
MyMethod obj = new MyMethod();  
obj.display(); int sq = obj.square(5);  
obj.add(1, 2);  
int tax = obj.getTaxRate();
```

Constructor

→ Constructor are the special type of methods.

Constructor

① Name is same as class-name. (even the cases should match)
Pen → Pen() / Pen(x) ✓ match

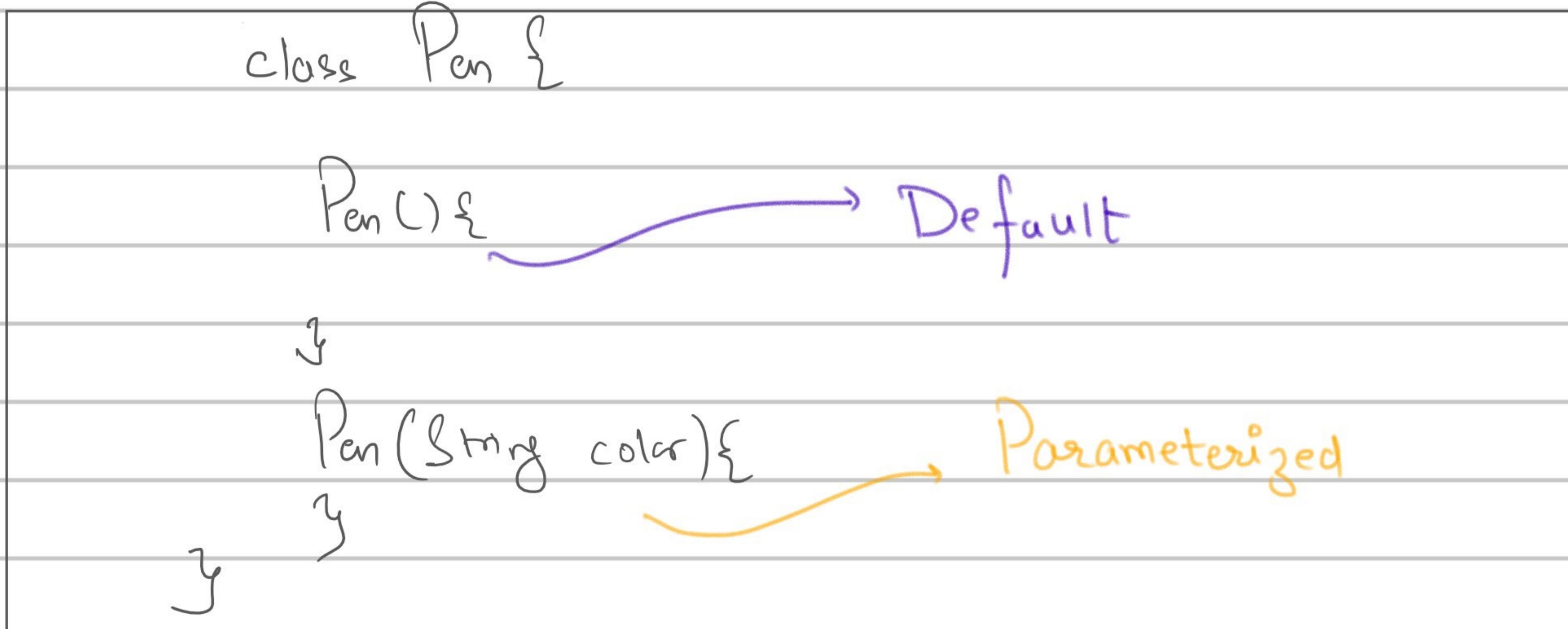
Car → car() / car(x) ✗

② No return type (not even void)

* Constructors are of two types *

① Default

② Parameterized



* Constructors are invoked when an object is instantiated.

Constructor defines the state in which object will be created.

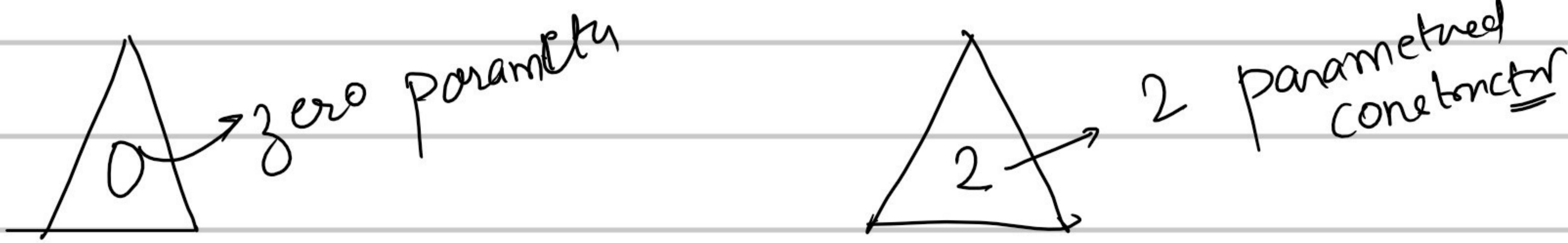
Constructor

Method

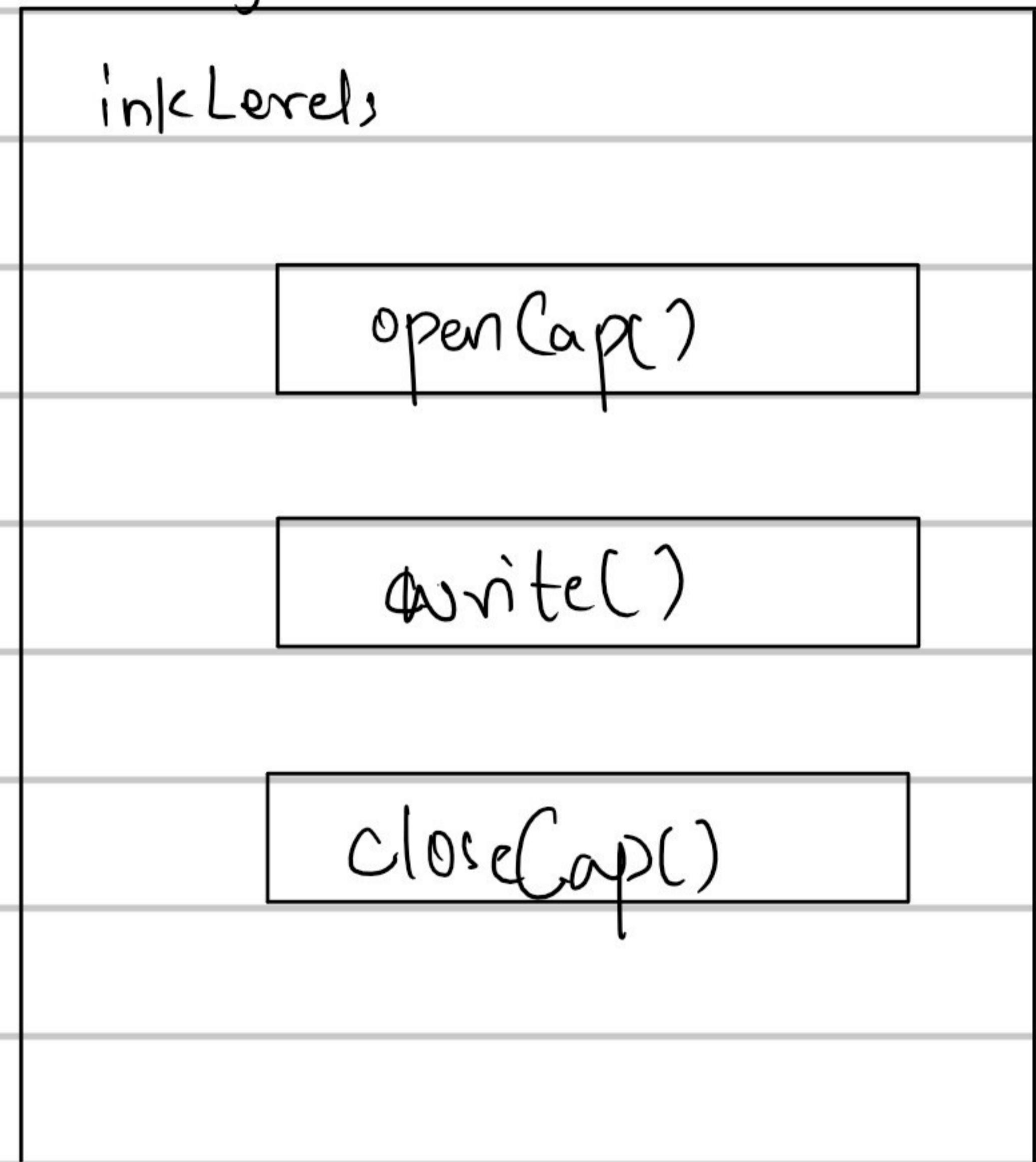
→ invoked while object is getting ready

→ invoked after an object is ready

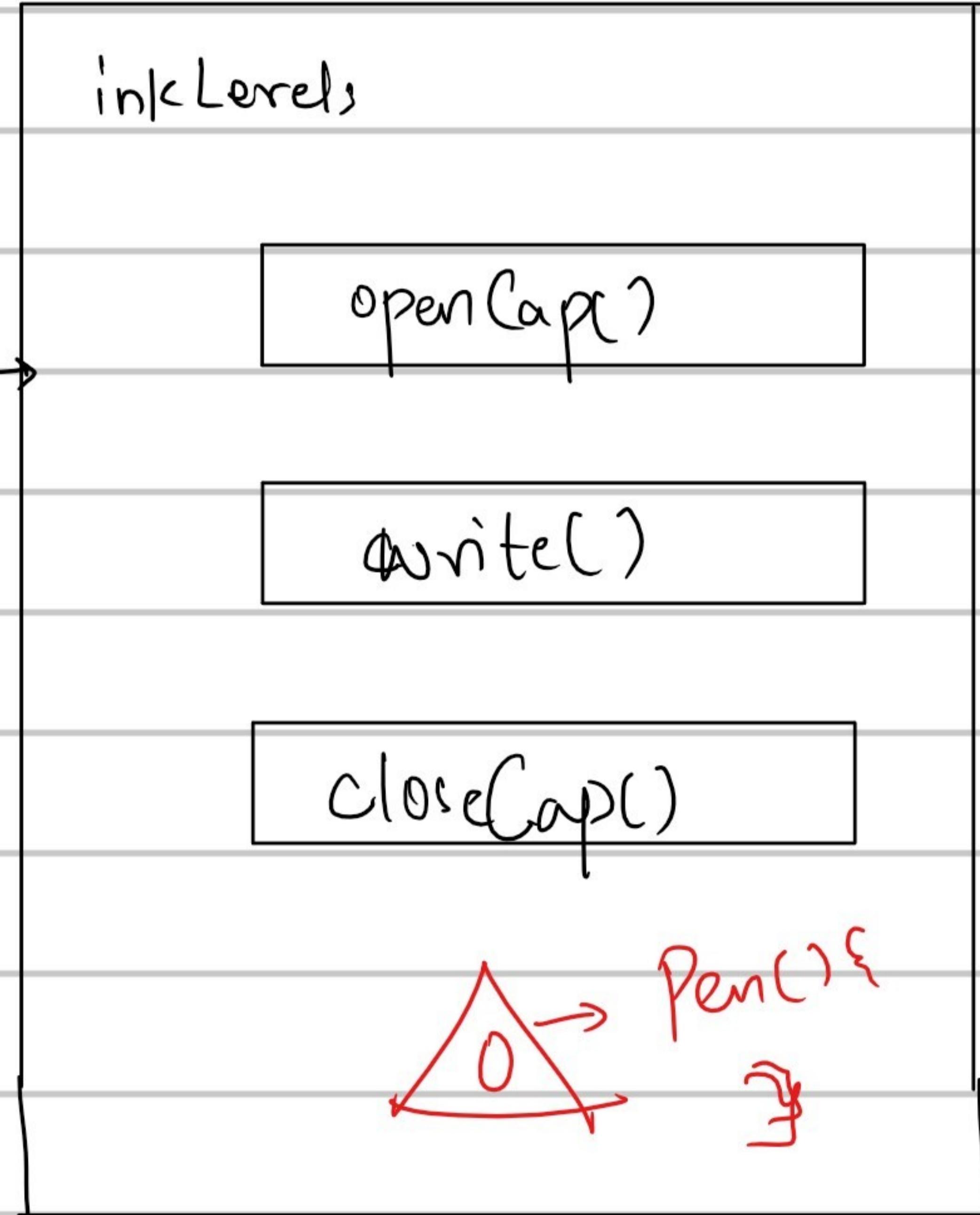
Tom gets ready & go to the
 constructor → Office & work → methods



Pen .jar



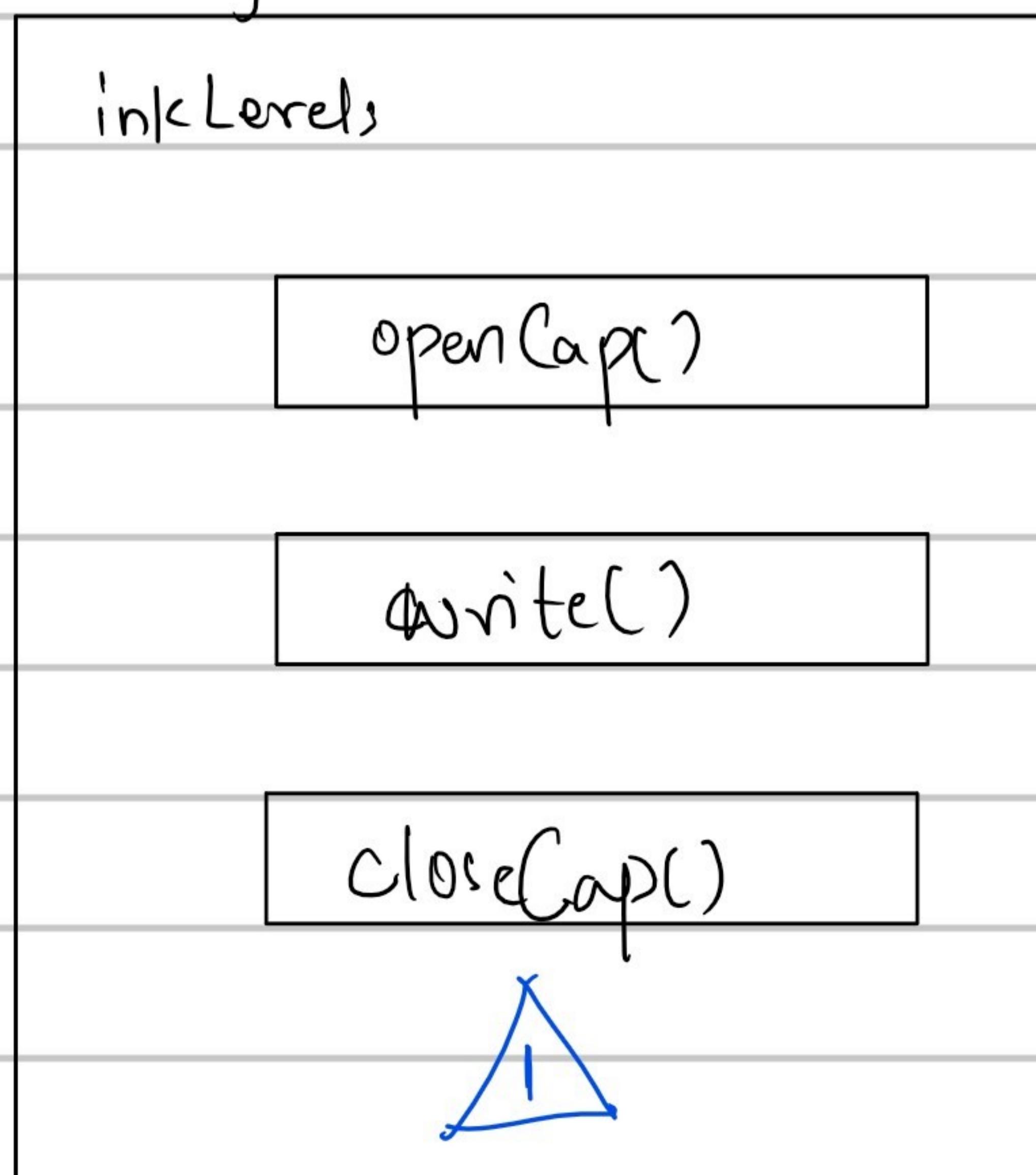
Pen .class



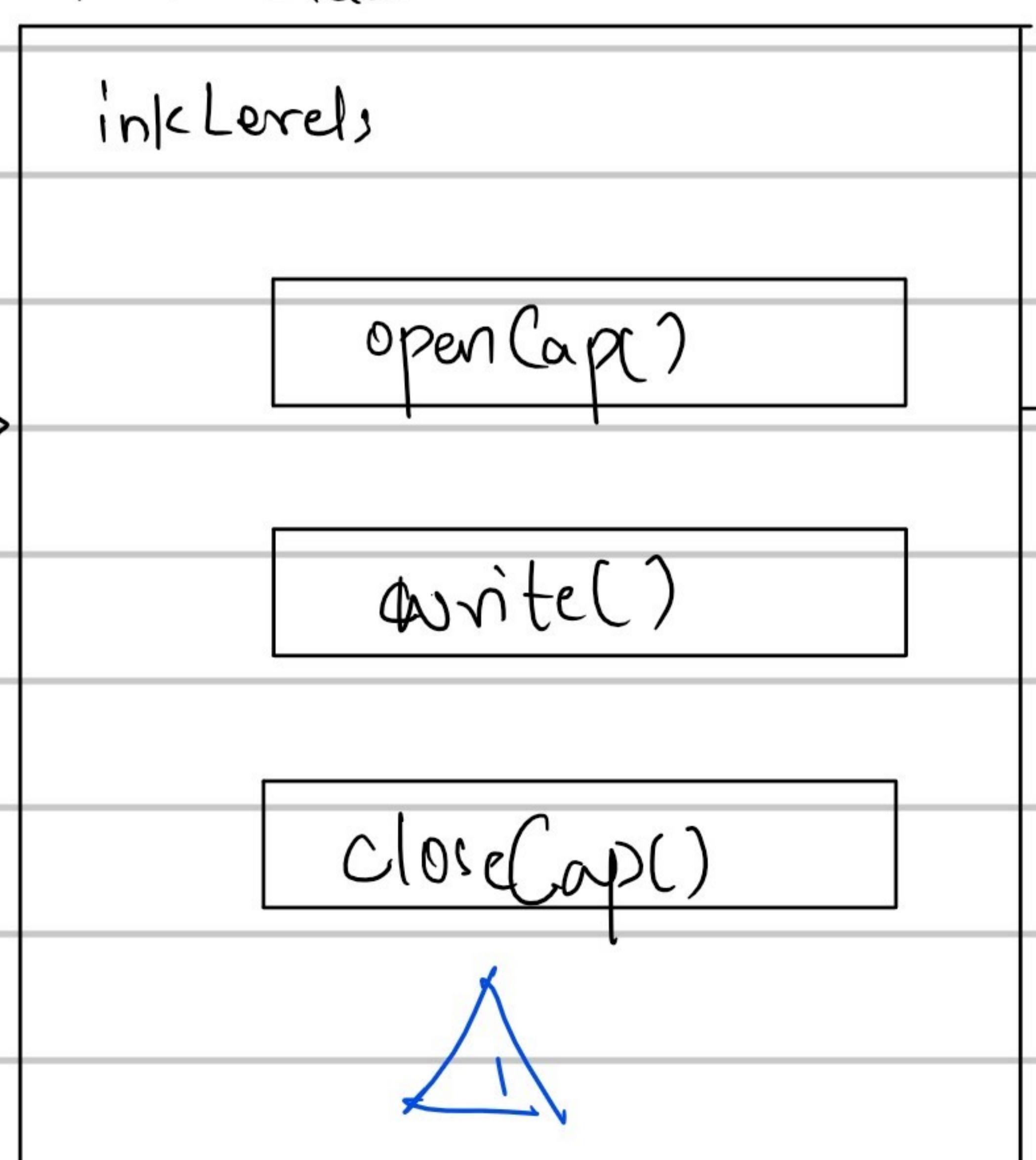
Whenever a .jar file is compiled &
 if it does not have any constructor
 definition inside it, Java Compiler
 provides a free default constructor
 in its .class file

Pen p=new Pen(); ✓
 p.openCap(); ✓
p.fly(); ✗

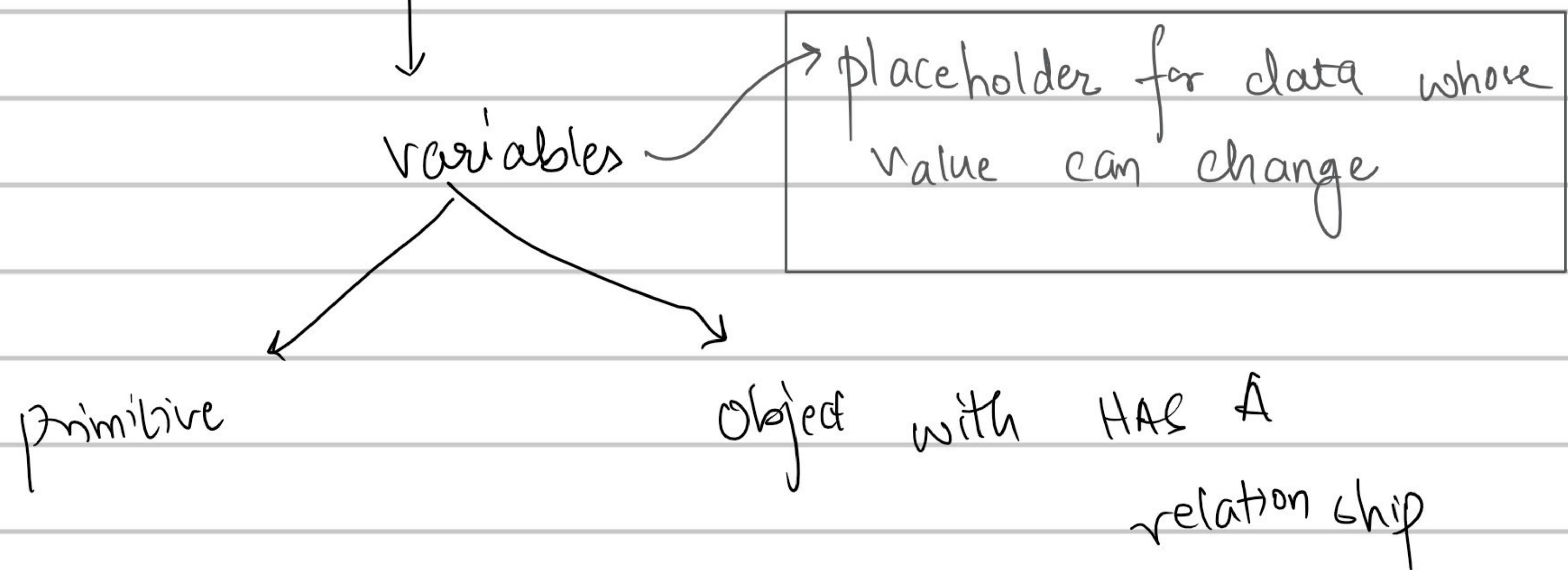
Pen .jar



Pen .class



State



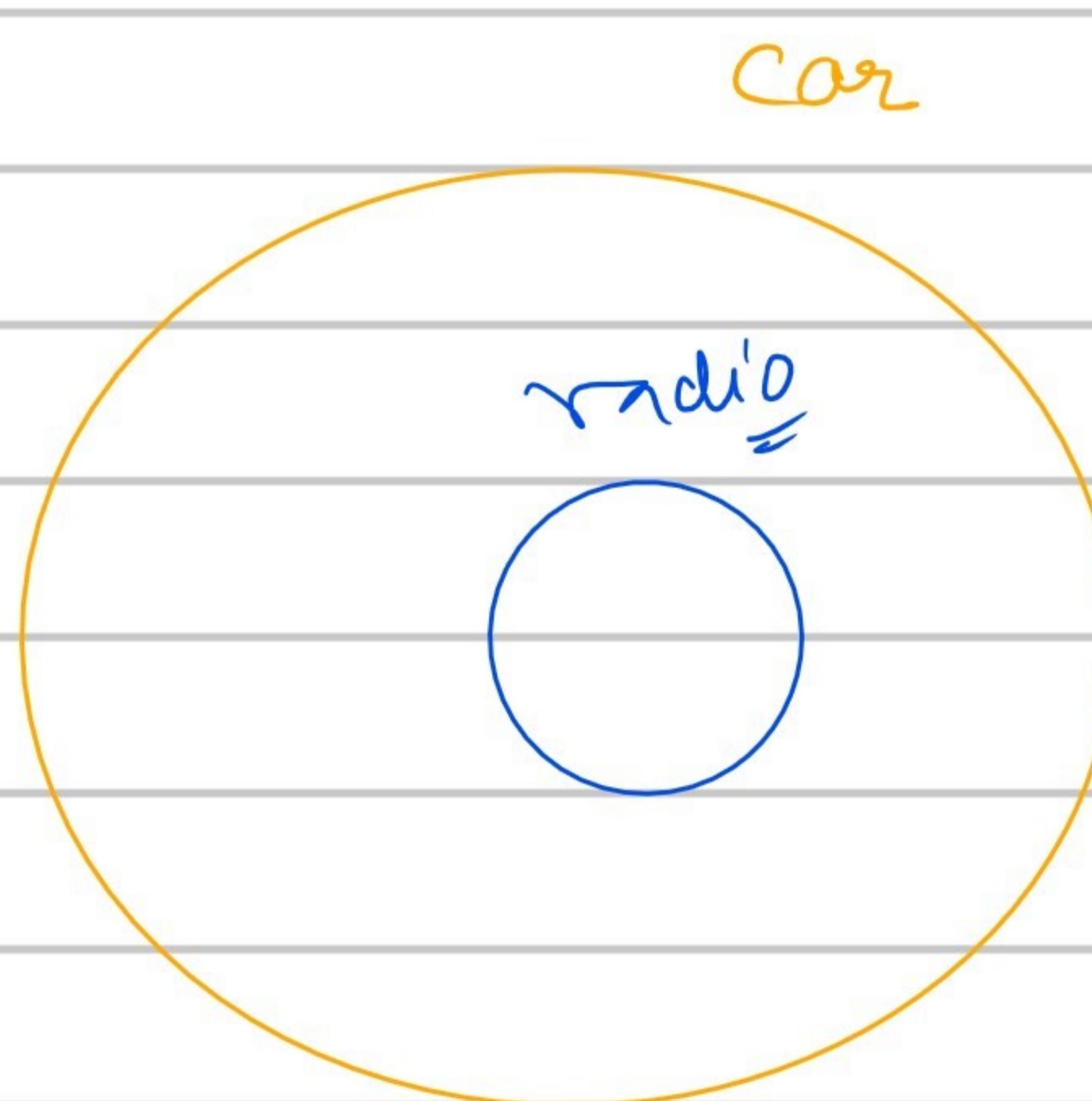
* Primitive Variables

↳ are those variables which are declared using data types

byte boolean char short int long float double

* Object Variables

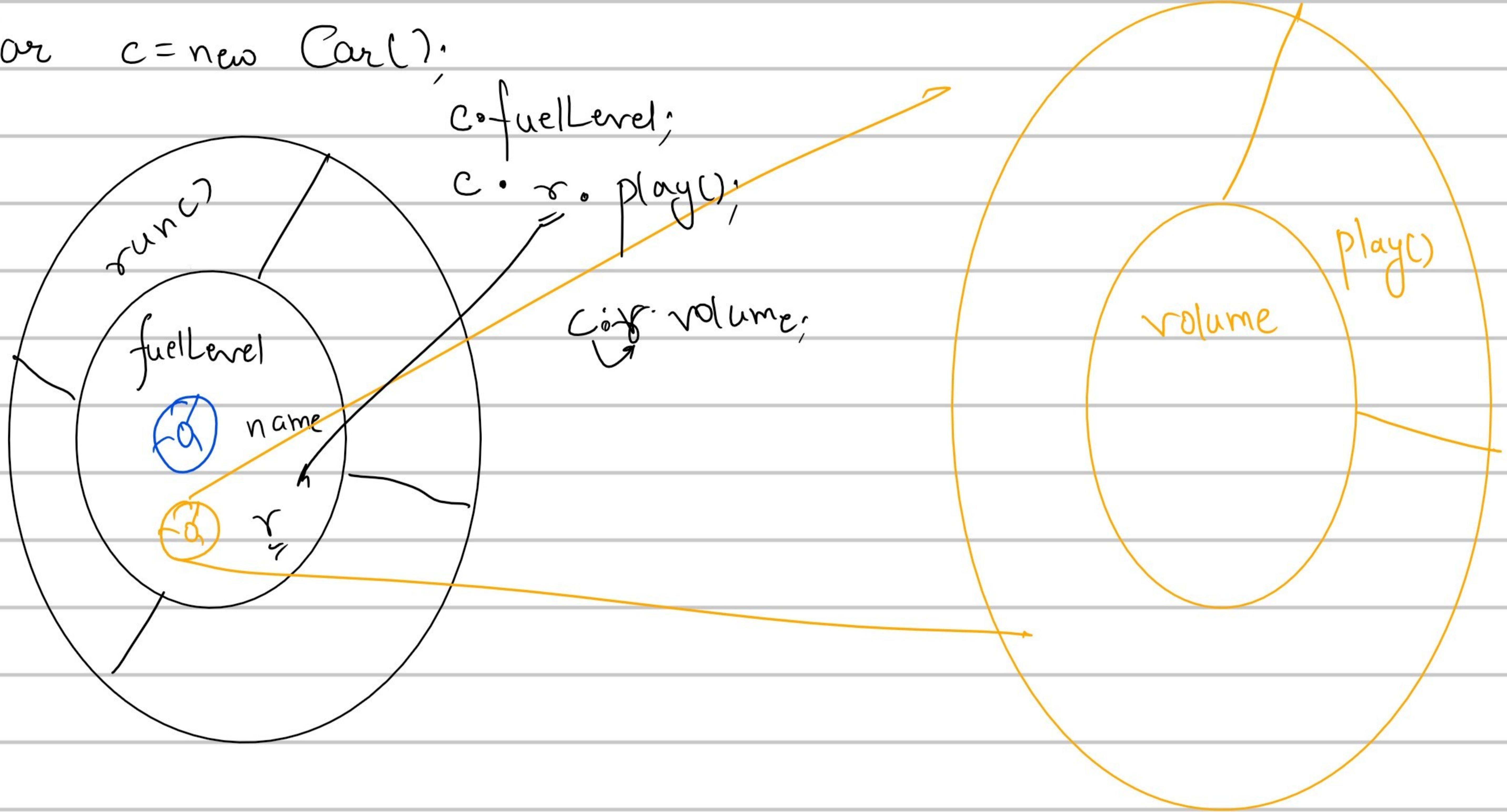
↳ are those variables which are declared using other class objects



```
class Radio {  
    int volume;  
    public void play() {  
        R → "+volume";  
    }  
}
```

```
class Car {  
    String name;  
    int fuelLevel;  
    Radio r = new Radio();  
}  
public void run() {  
}
```

Car c = new Car();

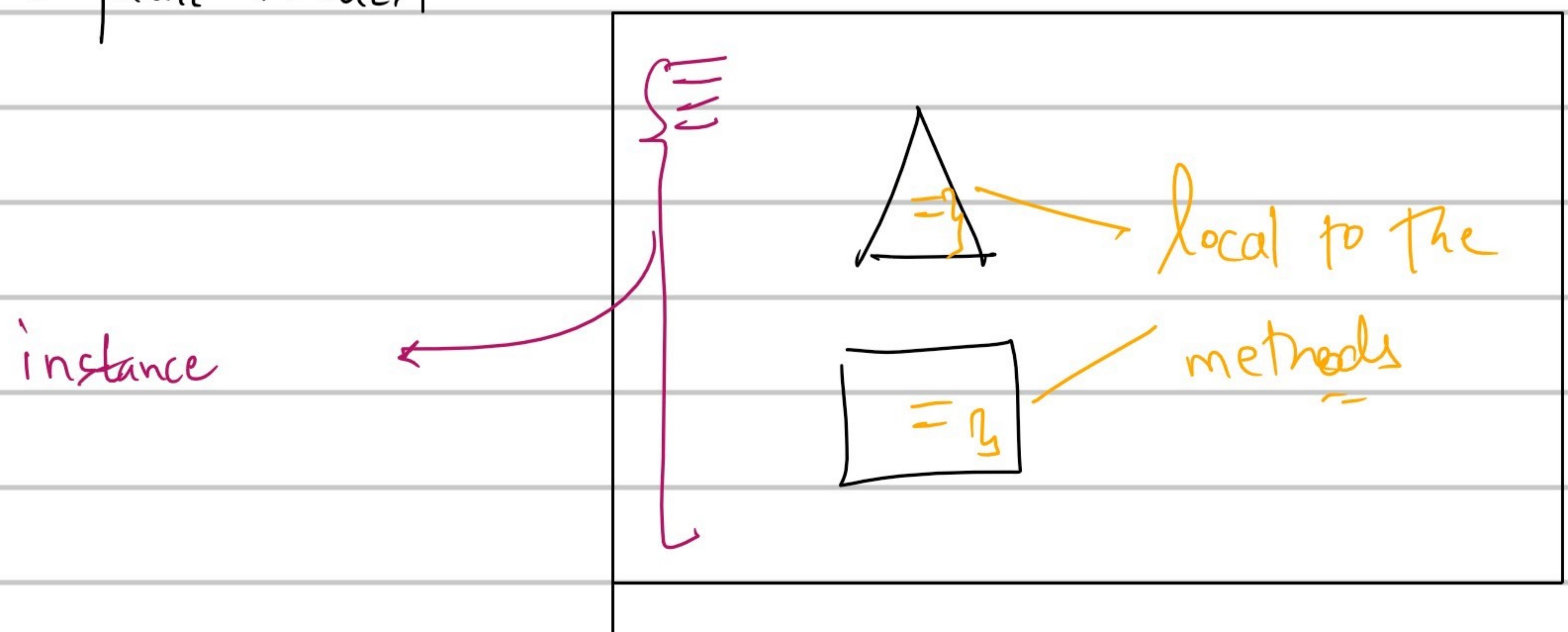


Based on the position of the variable,

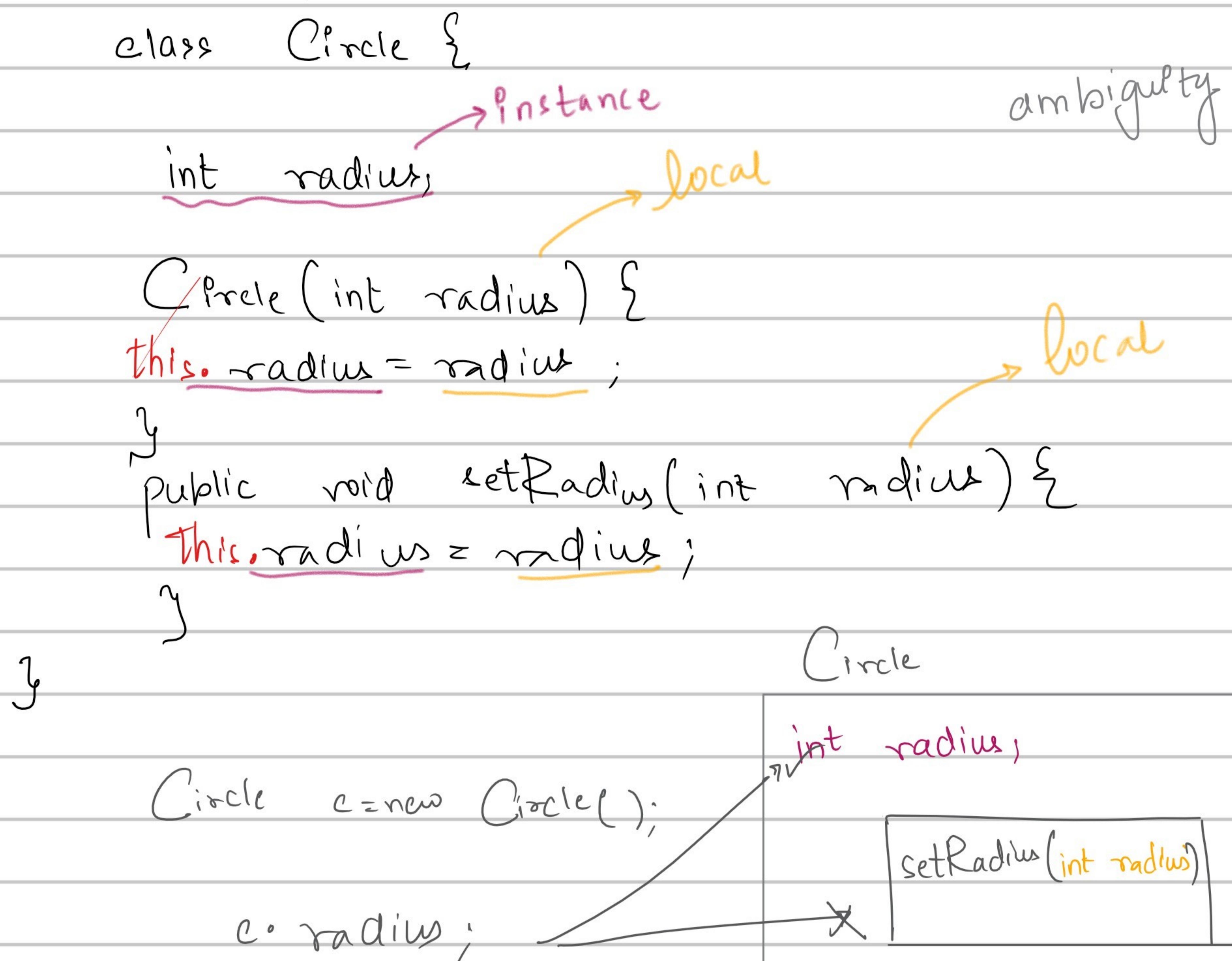
① Instance

② Local

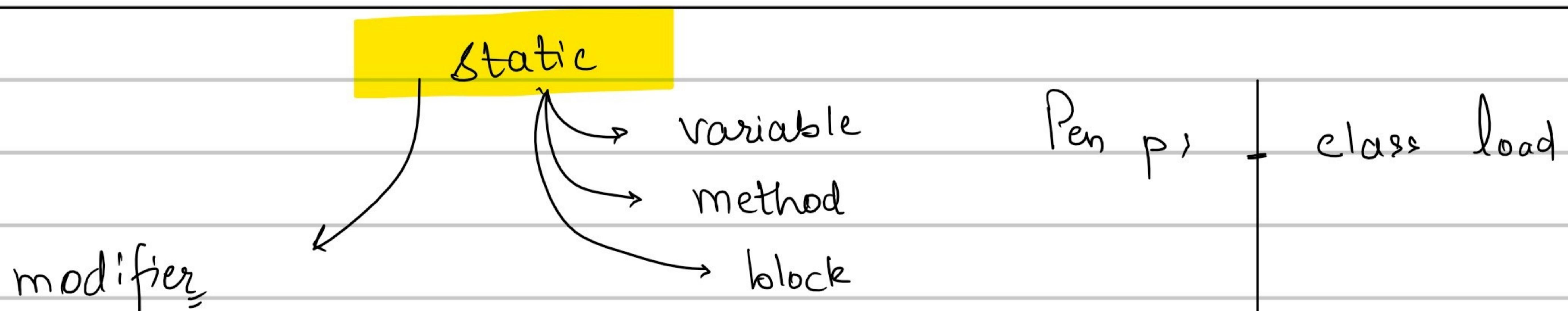
instance	local
→ those variables which are declared outside the constructor and/or methods	those variables which are declared inside the constructor and methods, parameters are local
→ available throughout the class.	→ available in the constructor/method where it is declared
Declared	
→ int i; If a variable is declared Pep p; In instance area & is assigned to its default value.	→ if left declared → will result in error



Local & Instance Variables can have same name



this keyword
points to an instance of
same class



* Anything which is of nature static

p = new Pen();

object load

① Is loaded when the class is loaded.

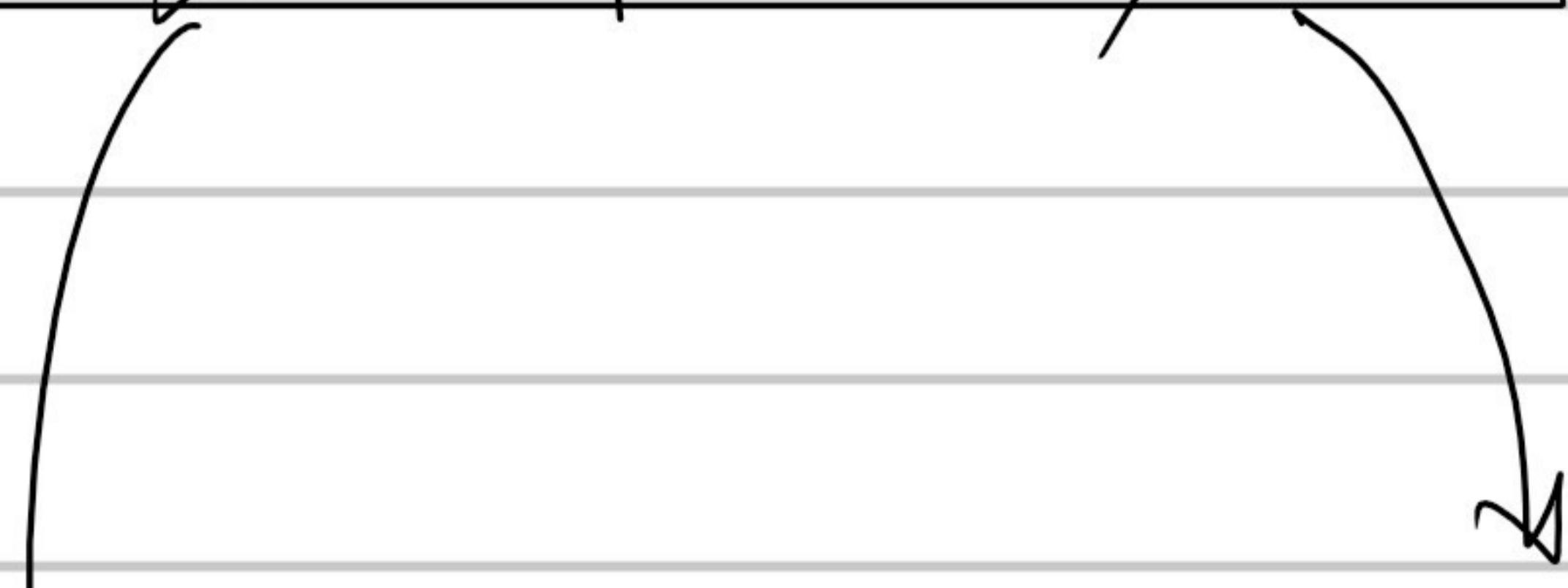
Time

② Have one copy for all the instances

③ Can be accessed directly using the class-name

P_1 | P_2

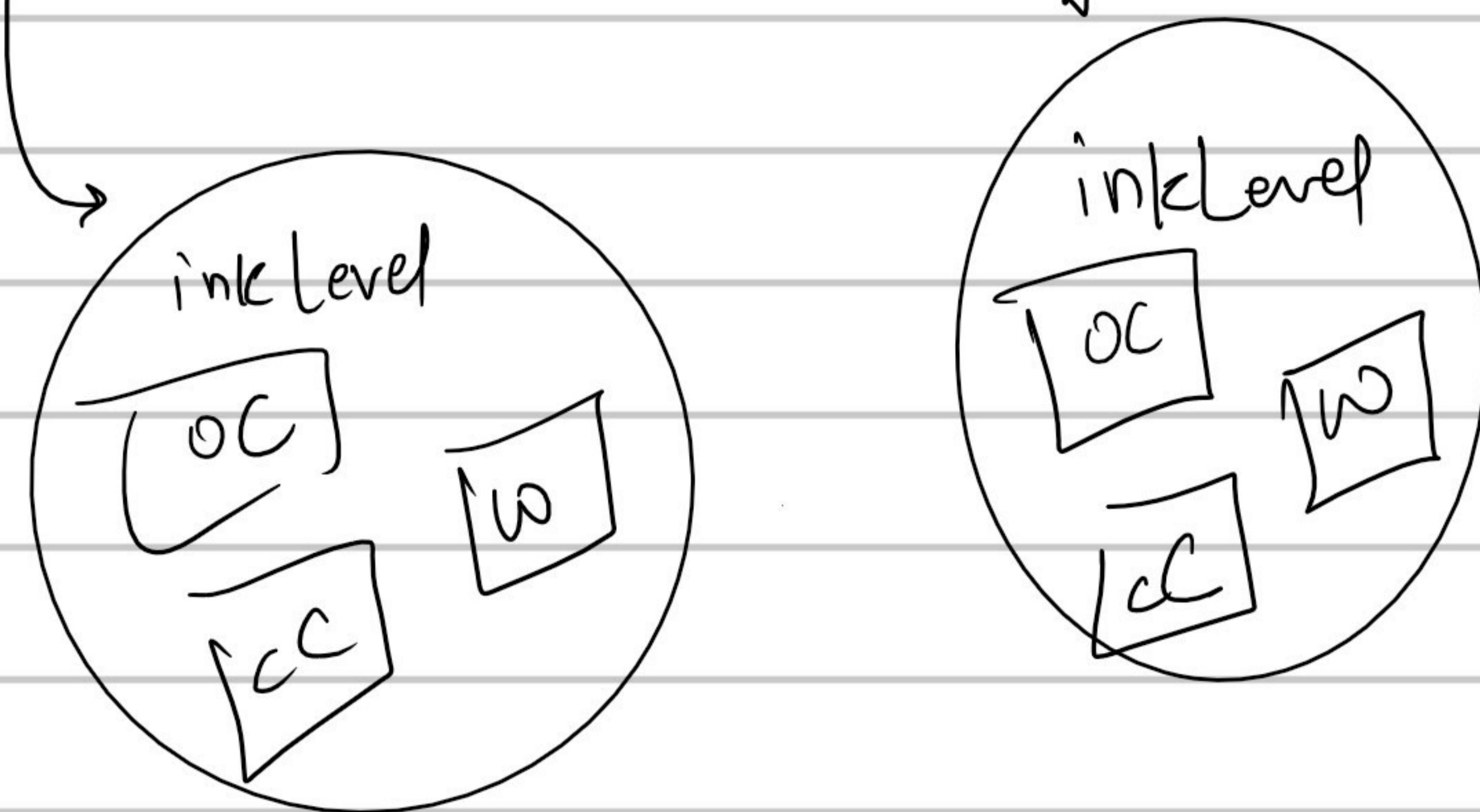
~~null~~ | ~~null~~



Pen $P_1, P_2;$

$P_1 = \text{new Pen();}$

$P_2 = \text{new Pen();}$



class ObjectCounter {

static int count = 0;

.ObjectCount() {

 count ++;

y

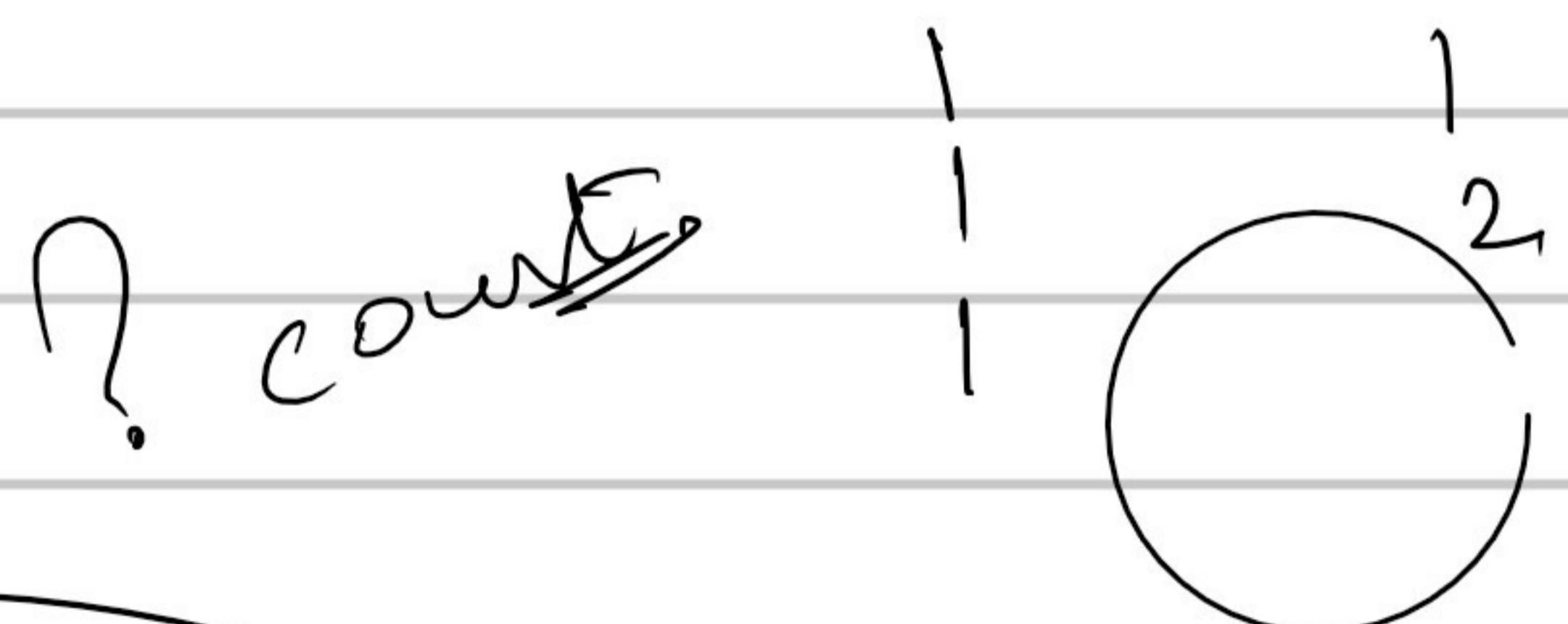
}

ObjectCount $OC_1, OC_2, OC_3,$

$OC_1 = \text{new } \rightarrow ()$

$OC_2 = \text{new } \rightarrow ()$

$OC_3 = \text{new } \rightarrow ()$



count

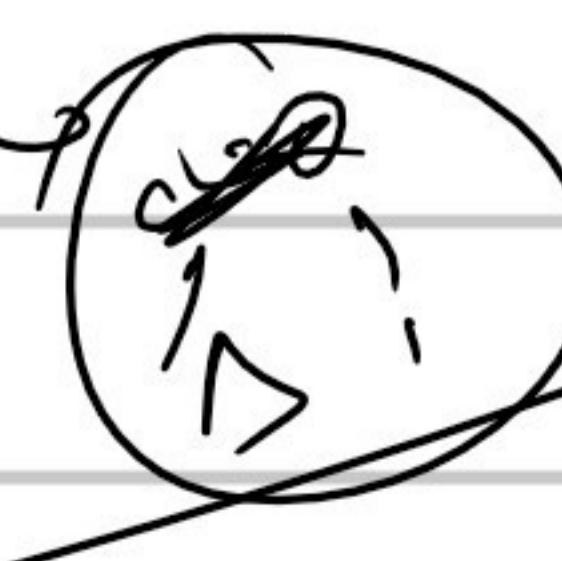
GC³

OC_2

~~null~~

~~null~~

~~null~~

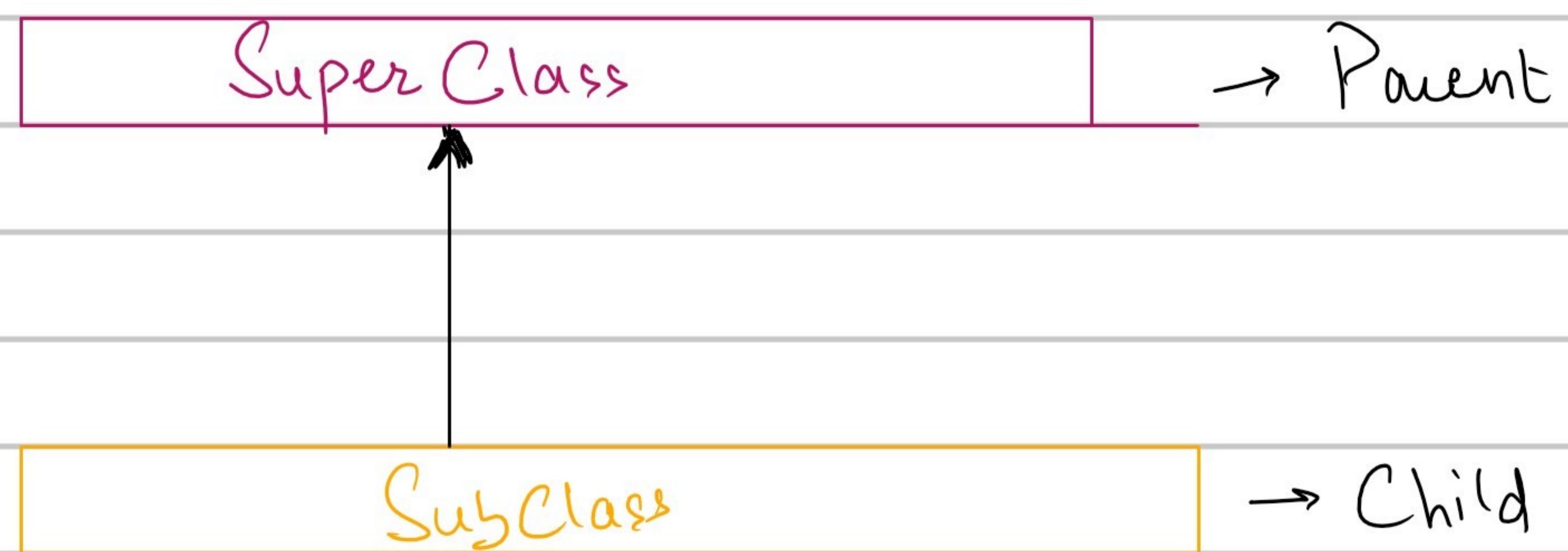


Inheritance

Parent → Super Class

Child → Sub Class

Process through which object of sub-child acquires the state & behaviors of its super class



* How inheritance happens *

extends

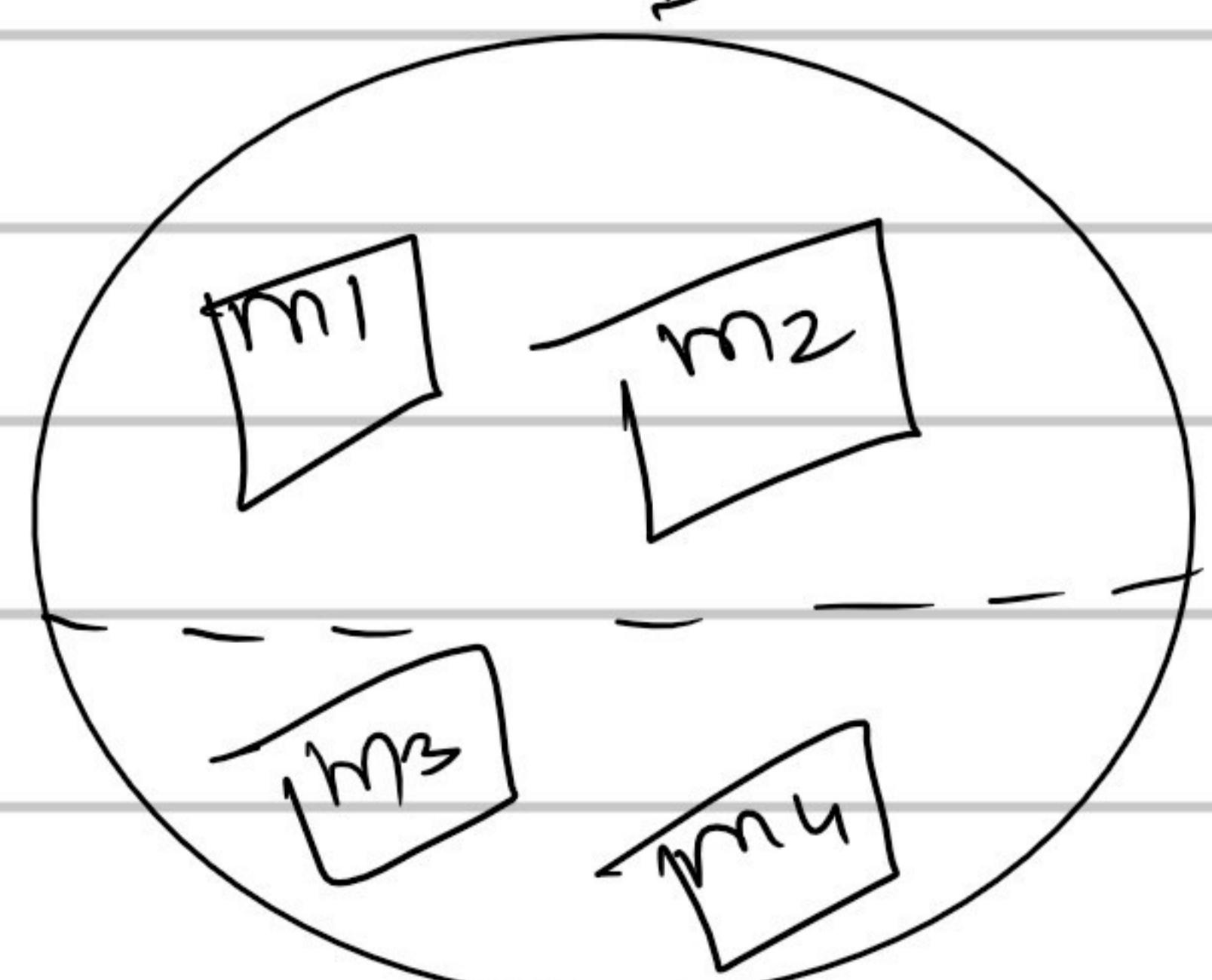
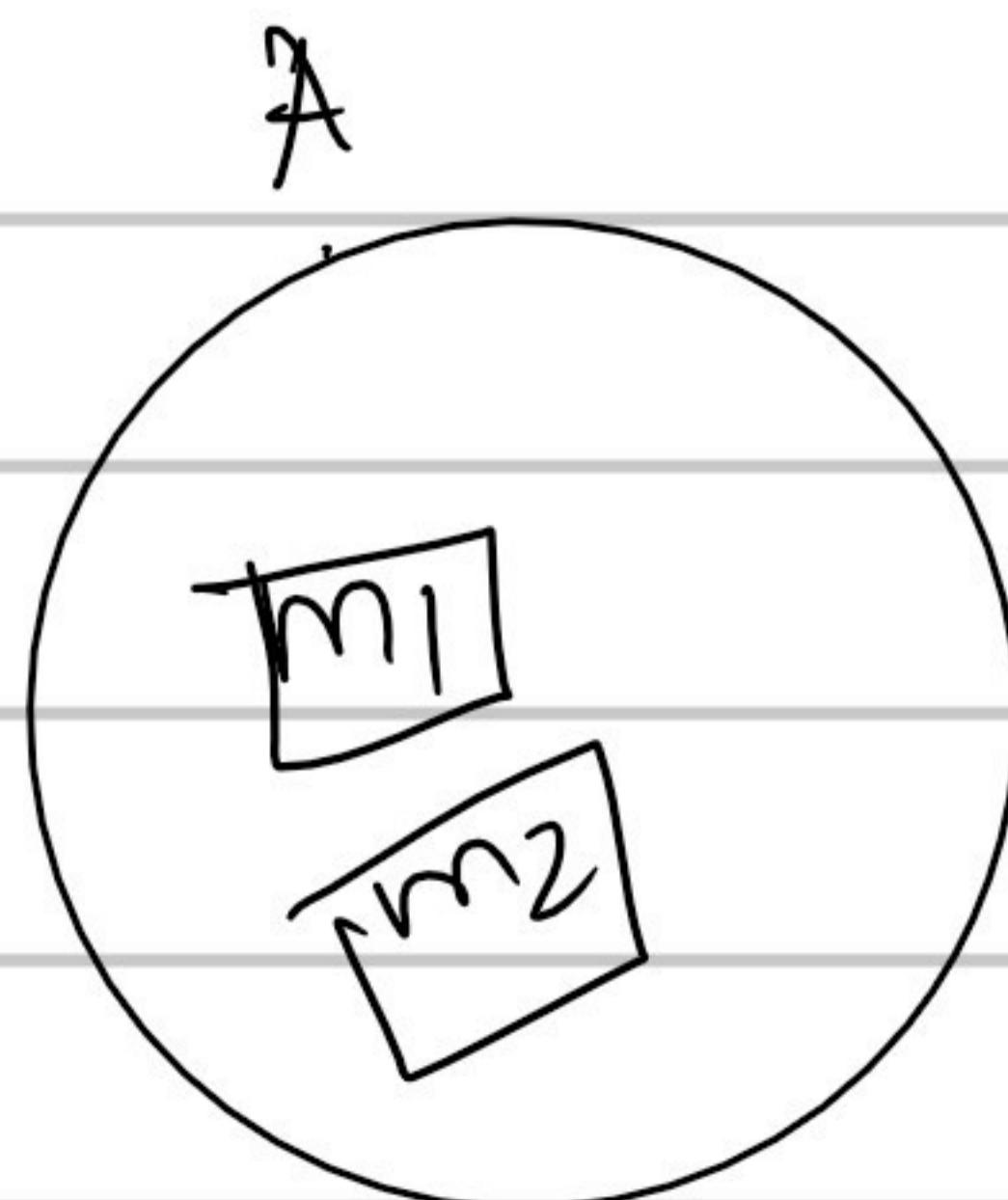
class Parent {

class Child extends Parent
{

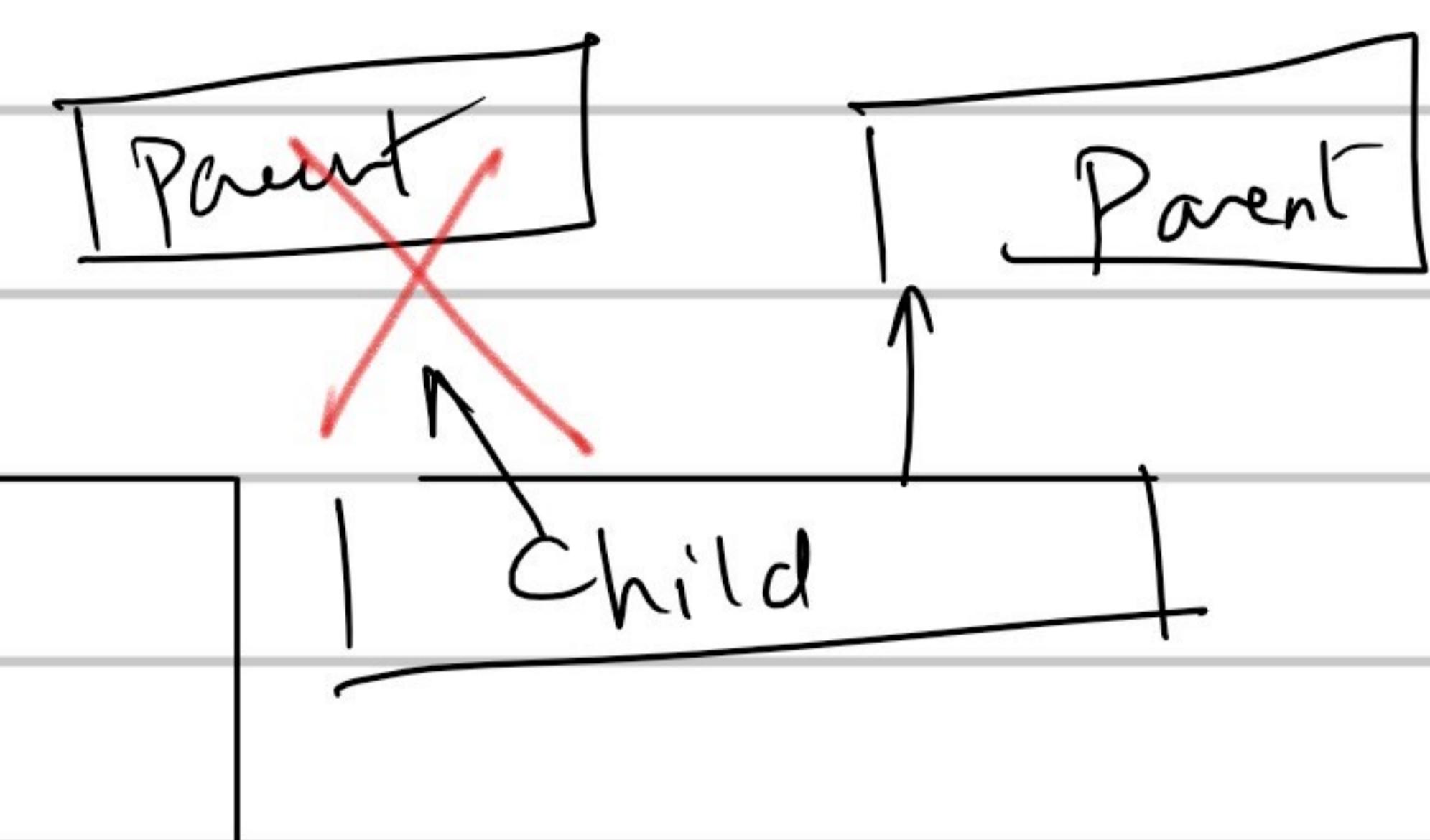
}

}

When inheritance happens all the data from super class comes down to sub-class



* Java does not support multiple inheritance.



→ At any given point in time; a

Java class will only have one direct super-class.

→ Java supports multi-level inheritance



→ Java does not support cyclic inheritance



* A java class atleast & atmost have one super-class

class X {

class Y extends X {

class Z {

Y

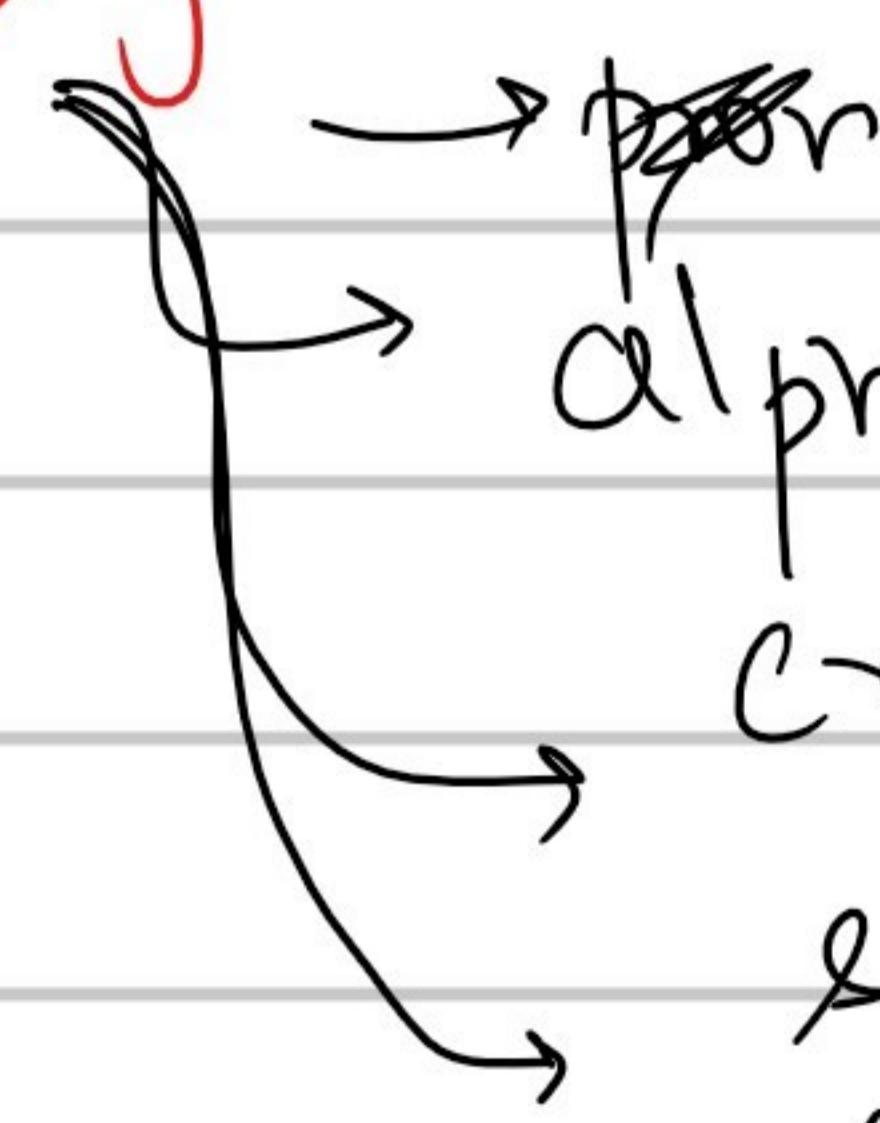
Y

Z

java.lang.Object

java.lang.Object

* Adam of Java *



→ provides us with methods;

* Method Overriding *

When a sub-class & super-class have method with same signature ; we say sub-class have overridden the super class method

↑
name of method ↓
parameters

class Animal {

public void eat() {

 → ("Eat → Animal");

}

}

class Elephant extends Animal {

public void sleep() {

}

public void eat() {

 → ("Eat of Elephant")

}

Animal

eat()

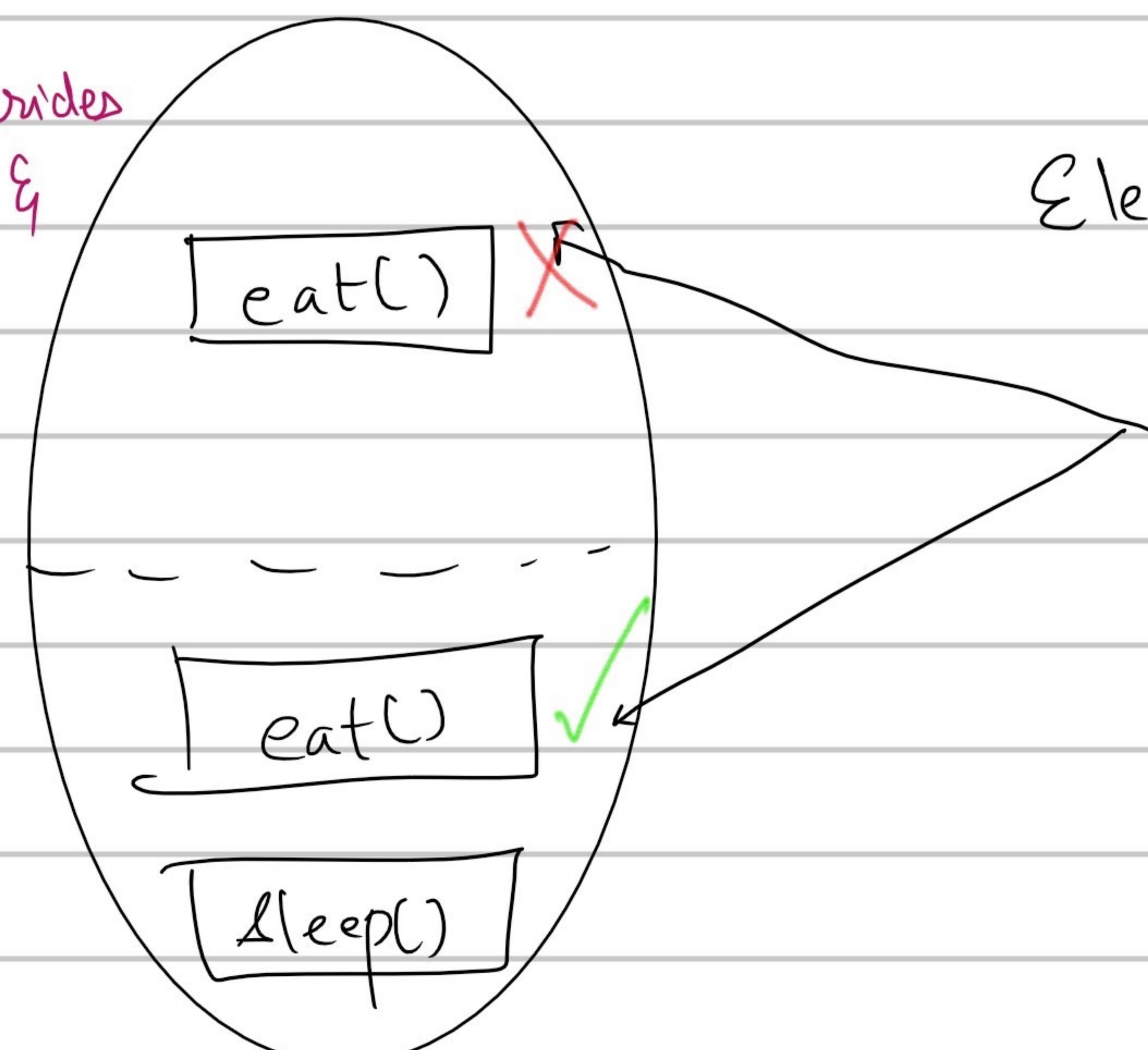
Elephant

eat()

x.eat();

* When a sub-class overrides a super-class method & if that method is invoked using

Sub-class object ; always the sub-class version of the method is invoked



Elephant e=new Elephant();

e.eat();

Super

refers to immediate super class

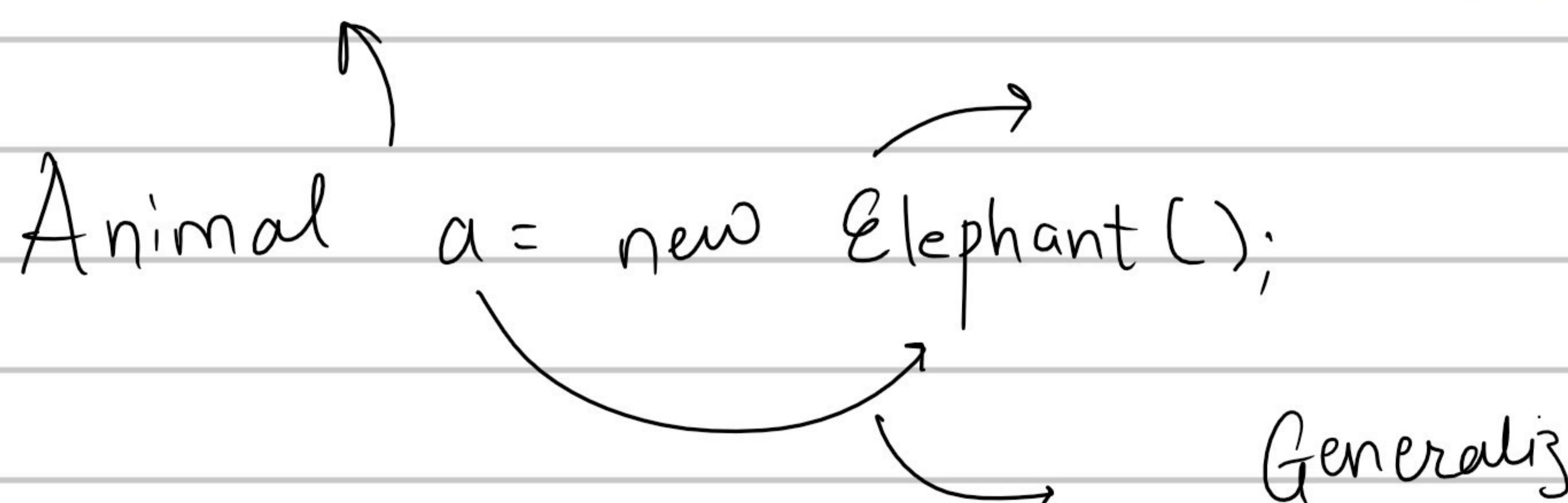
Super keyword can be used for accessing

① Variables in super class

② Methods in super class

③ Constructor in super-class

Object of a super class can be made to point at its sub-class instance.



public void treat(Animal a) {
 someObj.treat(e);

y

a.eat();

When an object of super-class
points at its sub-class instance; it
accesses only overridden methods; other
methods are sliced.

When super-class object pointing to sub-class instance
invokes the overridden method. Always the sub-class
version of method is invoked.

Animal a = new Elephant();



* final keyword *

① variable

② method

③ class

④ object

final variable

```
final int MAX = 100;
```

it's a convention to write final variable
totally in upper-case.

→ Once the value is assigned; it
cannot be
re-assigned

final method

* A method marked as final; cannot be overridden.

```
class A {
```

```
class B extends A {
```

```
public final void m1() {
```

 ↳ ↴

```
public void m1() {
```

 ↳ ↴

X

final class

A class marked as final; cannot be sub-classed

```
final class Y {
```

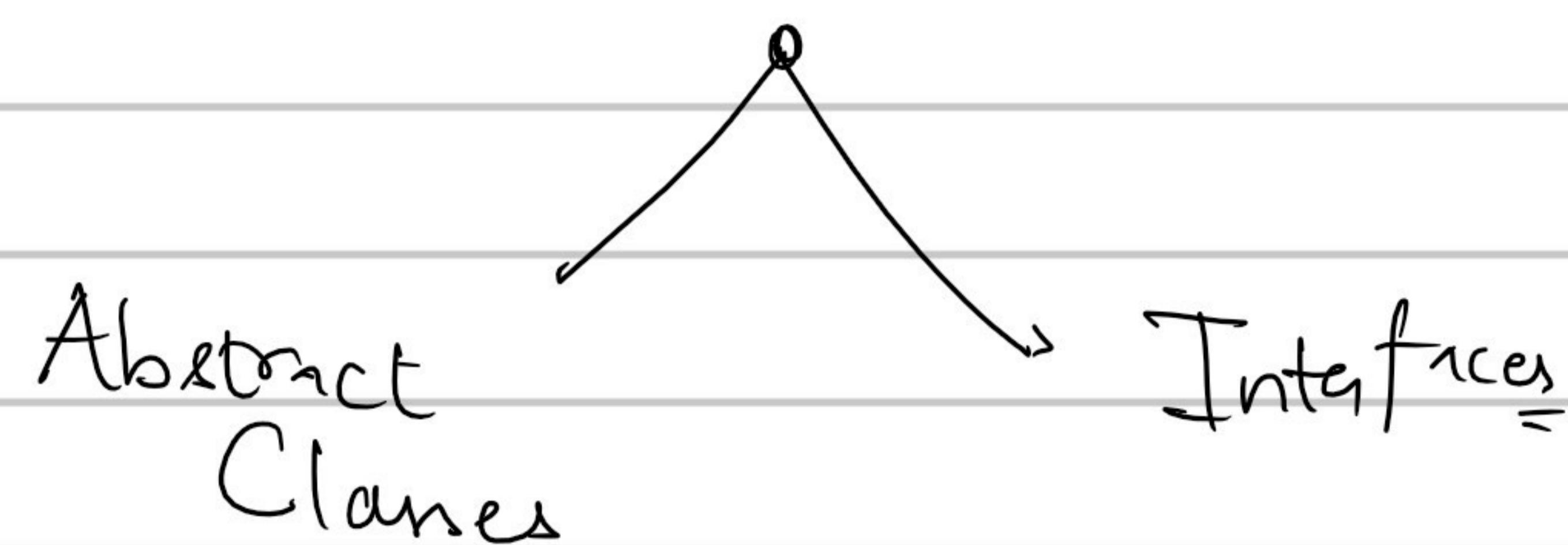
 ↳ ↴

```
class Z extends Y {
```

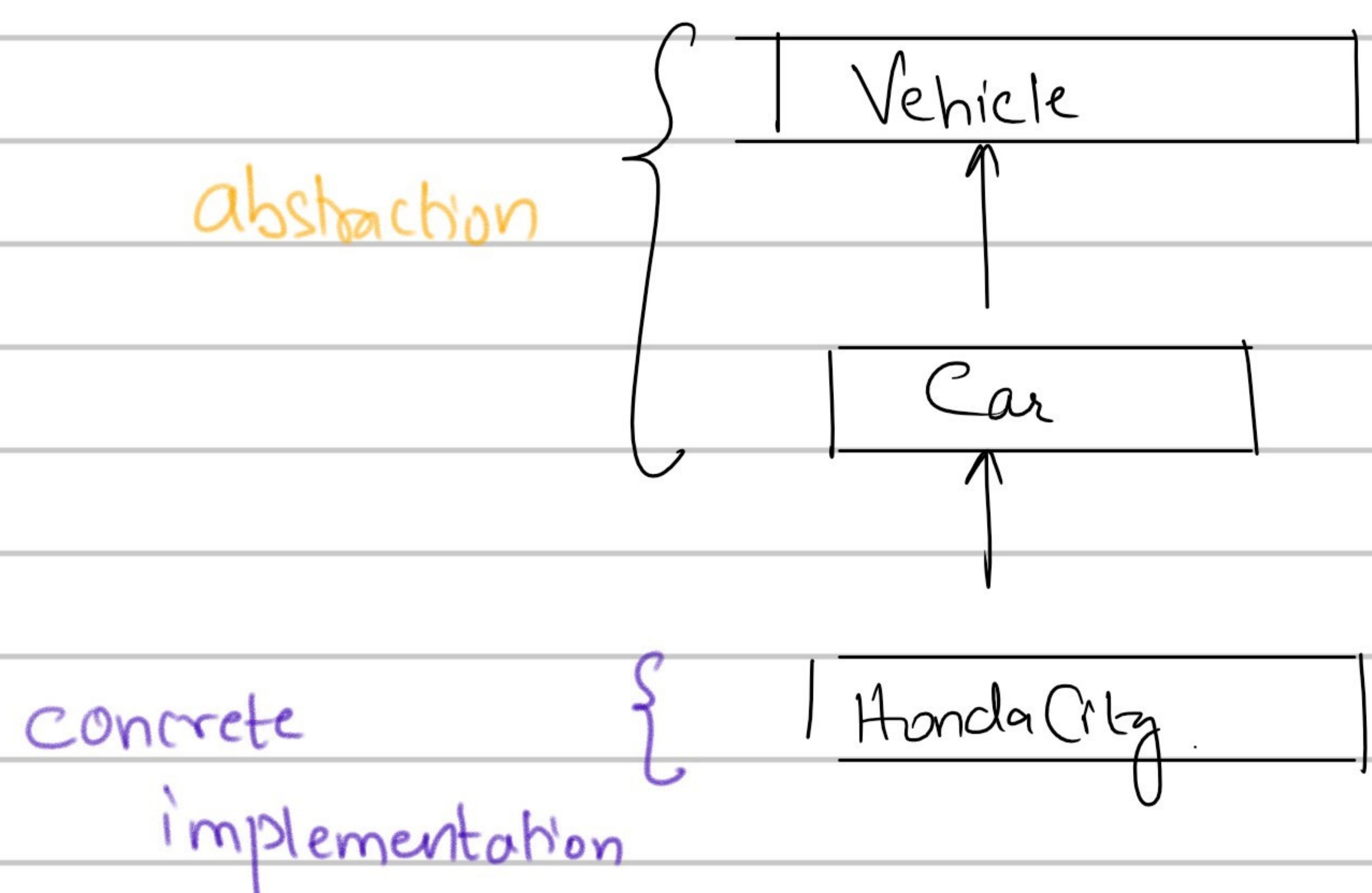
X

* Deferred Implementation *

↳ declare now
implement later



* Abstract Class *



* Abstract class is used when you want ~~to~~ your subclass

to override certain methods compulsorily

* To make a class abstract; we need to mark the class with keyword '**abstract**'.

abstract class Car {

}

* Abstract Methods

→ are those methods which are marked & declared as abstract

public abstract void start();

(){} → defining

(); → declaring

public abstract void stop();

abstract class Car {

An abstract class can have abstract as well as non-abstract methods.

public abstract void start();

We can create an object of abstract class; but not instantiate it.

public void mi() {

Car c; ✓
c = new Car(); X

y

J

Relation between an abstract super class & non-abstract sub class.

* When a non-abstract sub class extends an abstract super class, it has to override all the abstract methods; failing to which, the sub-class won't compile.

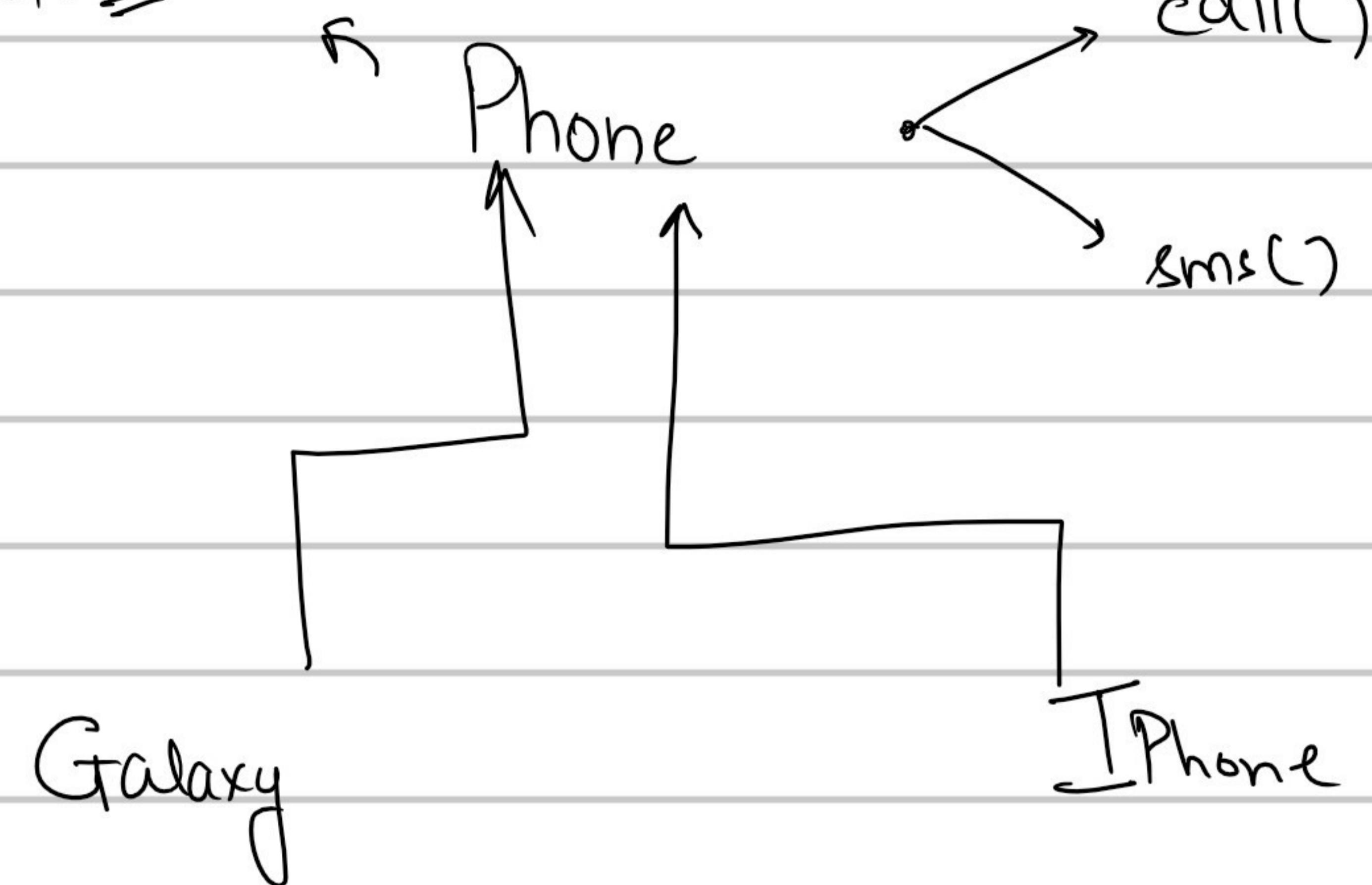
class HondaCity extends Car {

start()

stop()

y

abstract



CE

{ Certified
European }

p1()
p2()
p3()

* Interface *

is like a contract ; once signed
has to be complied with.

Interface CE {

}

final data
abstract methods

* An interface cannot have non-abstract methods

interface I1 {

public abstract void p1();

}

class A implements I1 {

public void p1(){
=

}

* When a class implements an interface, it should override all the abstract methods; failing to which the class won't compile.

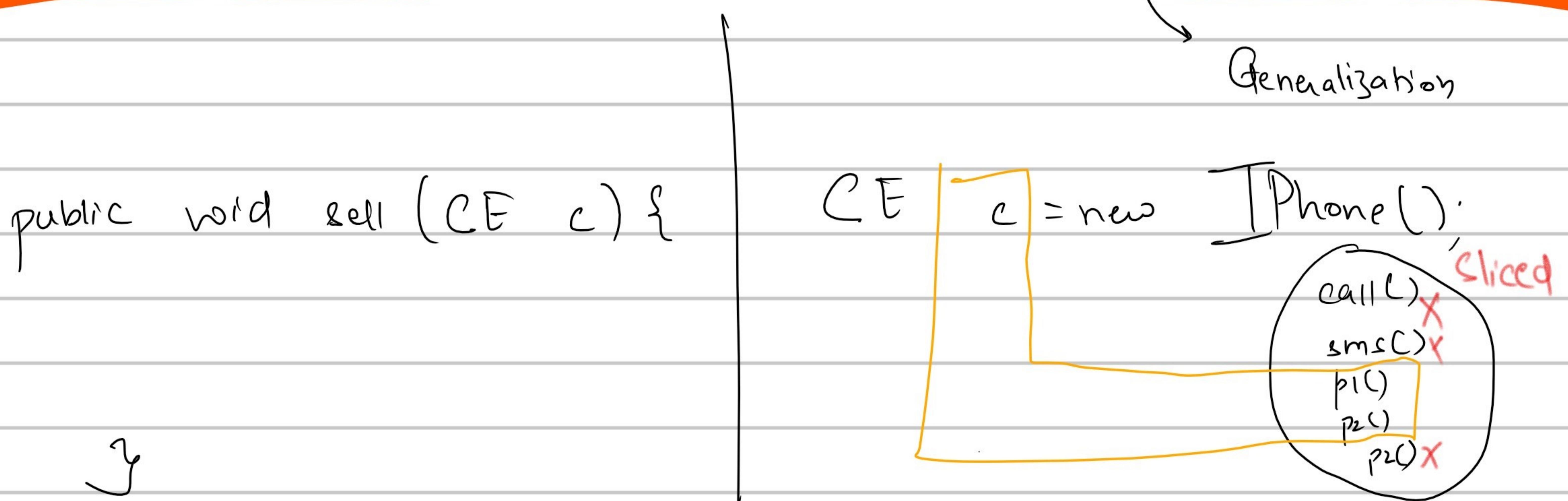
When you need to inherit & use interface at same time; we extends first & then implements

class iPhone extends Phone implements CE {
 }

* A class can implement more than one interface.

class iPhone extends Phone implements CE, ISO {
 }

Object of an interface can be made to point at an instance of a class which implements it.



Interface / Abstract Class

A class which implements/extends

* What to do

* How to do

* Packages *

in built packages

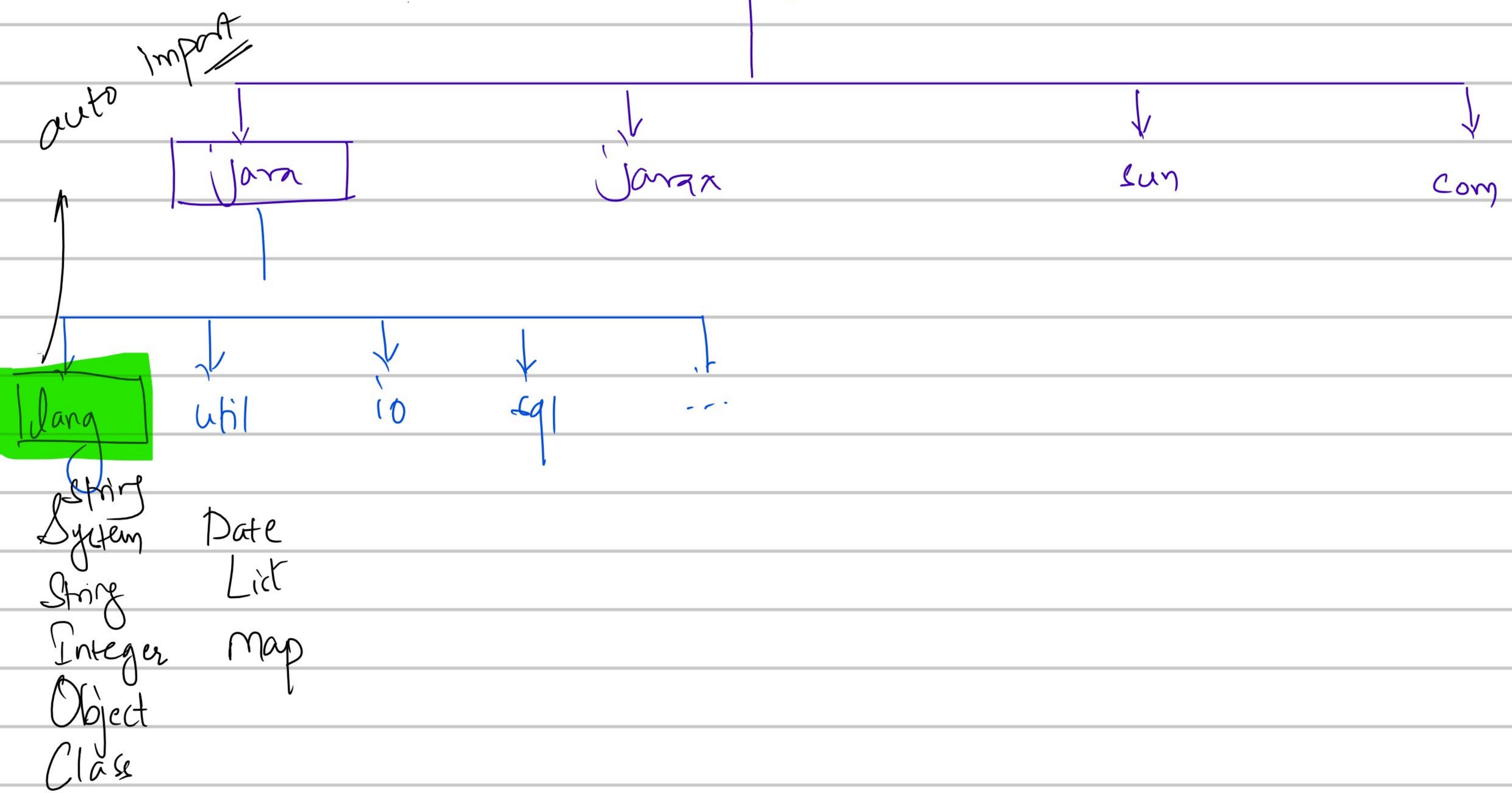
collection of classes & interfaces in some directory hierarchy

creating our own packages.

In-built Java Library Packages

rt.jar

(runtime · Java archive)



* A class in same directory can access another class *

TodaysDate.java

```
import java.util.*;  
class TodaysDate {  
    main (){  
        A a=new A();  
        Date dt=new Date();  
        System.out.println(dt);  
    }  
}
```

A.java

```
class A {  
    public void m1(){  
    }  
}
```

* How does a Java class recognize another Java class *

1. Tries to search for the class in same directory.
2. Tries to search in java.lang package
3. Tries to search in Imported package

* Creating our own package *

URL → www. sc.com
sc.com
↓

→ creating segregation in project.

Package → com.sc

Layered Architecture

com. & co. rpg. loans

beans

→ Customer.java

ui

→ CustomerUI.java

helper

service

→ CustomerService.java

controller

→ CustomerController.java

* Access Specifier *

* public → available everywhere.

If no access
specifier is defined.
uses default.

* private → only available in same class

* default → only available in same package

* protected → default + inheritance

com.gcb.a

```
public A {  
    public int x;  
    void m1();  
    protected void m2();  
}
```

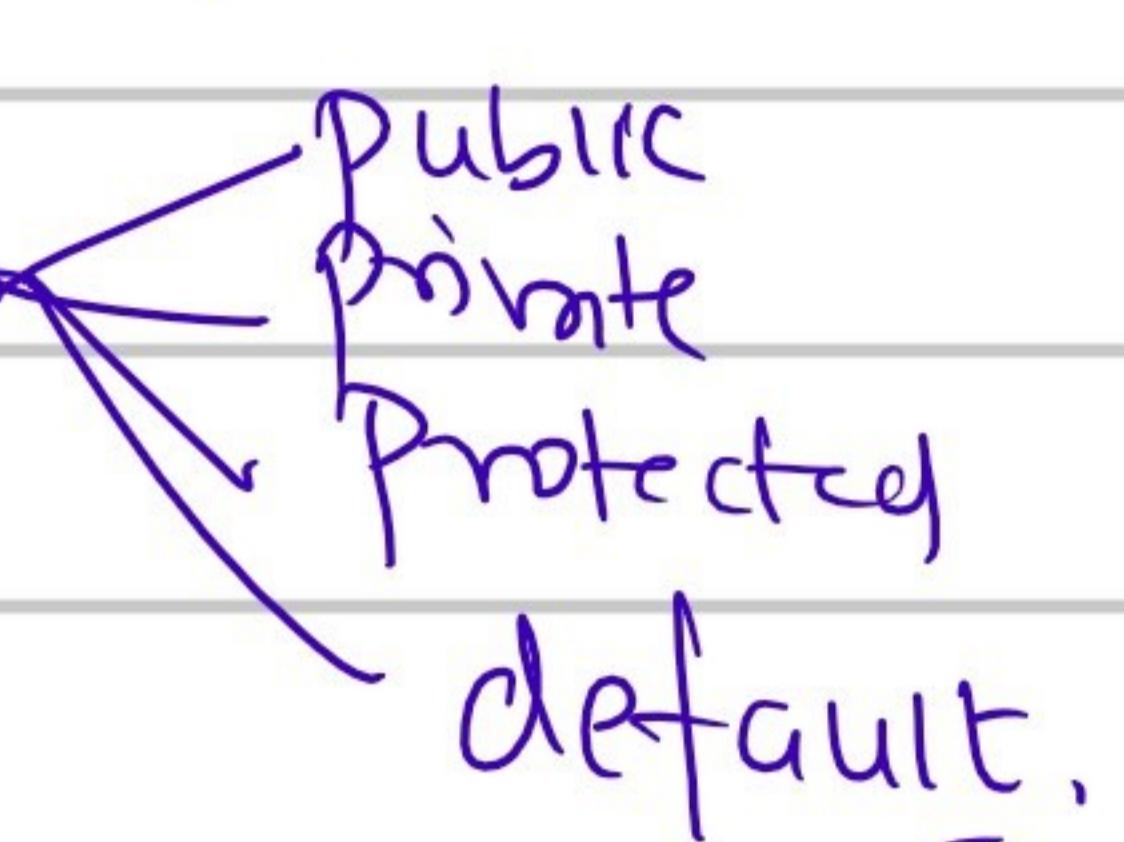
com.gcb.b

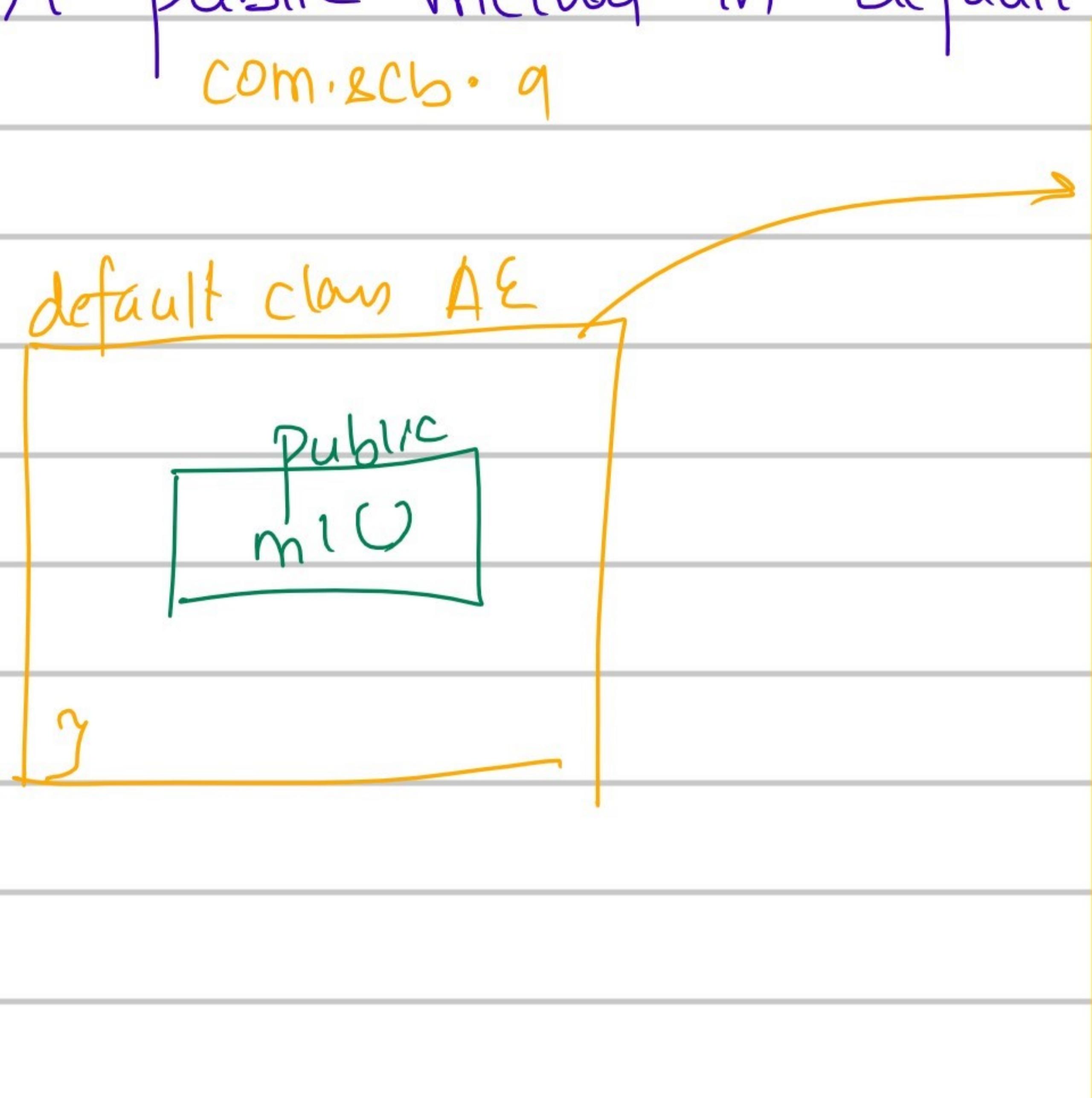
```
class C extends A {  
    x  
    m2();  
}
```

```
class B {  
    }  
}
```

```
class D {  
    A a = new A();  
    }  
}
```

* Quick facts about access specifiers *

- ① A class can only be marked default or public.
- ② A variable/method can be marked  public, private, protected, default.
- ③ A public method in default class will not be available outside the package



- ④ A default method in public class won't be available outside the package

Access Specifier	Same Class	Other class in same package	Sub-class in other package	Other class in other package
public	✓	✓	✓	✓
protected	✓	✓	✓	✗
default	✓	✓	✗	✗
private	✓	✗	✗	✗

String

Pen p;

p=new Pen();

String s="Tom";

String s=new String ("Tom");

String Object

Collection of characters.

Java.lang.String.

String gets special

String
Constant
Pool

treatment from JVM

S t a n d a r d
0 1 2 3 4 5 6 7

0 1 2 3 4 5 6 7 8
H a m b u r g e r
1 2 3 4 5 6 7 8 9

String are of nature immutable

String s="—";

String s1 = s.concat("—");

StringBuffer

Exception Handling.

Exceptions are unexpected erroneous events occurring at runtime breaking normal flow of the code.

Exceptions can be handled in two ways.

① try... catch

② declaring that exception

① try... catch

try {

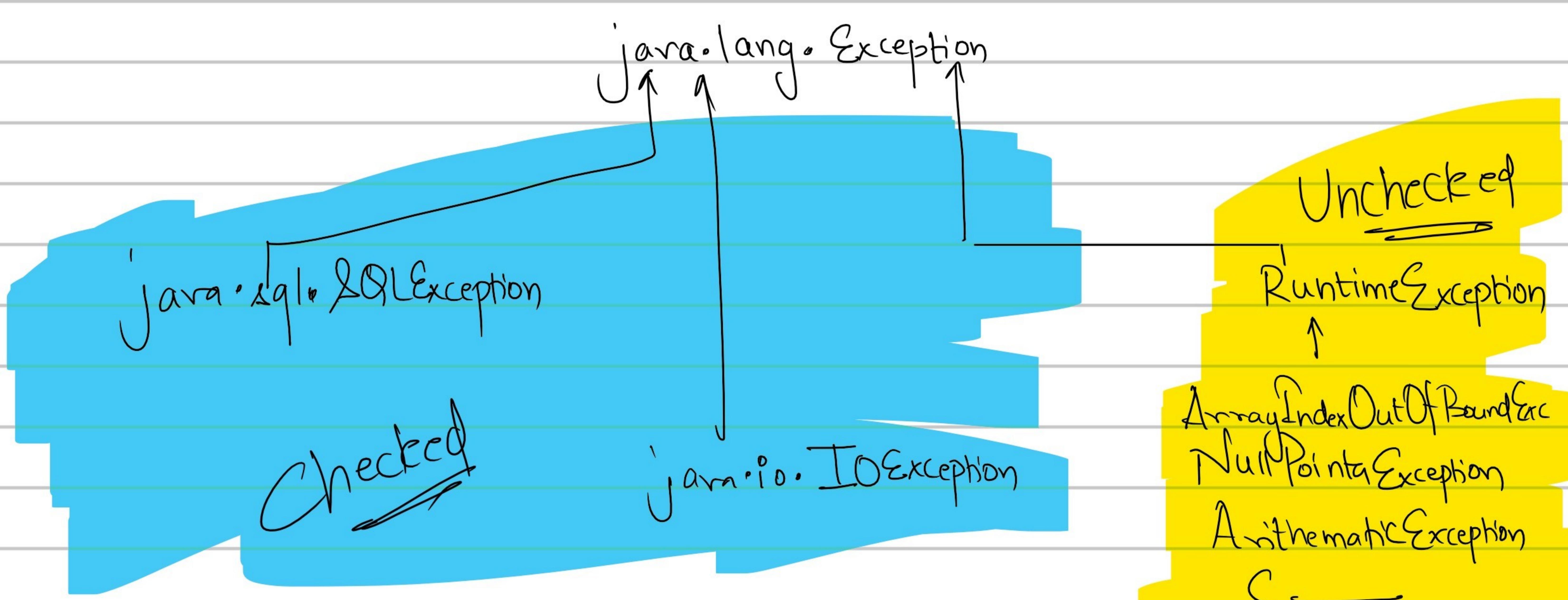
Any piece of code which is prone to throw an exception goes in try block

}

catch (Exception e) {

When exception happens what steps are to be taken.

}



Unchecked Exception → Any subclass of RuntimeException including

RuntimeException

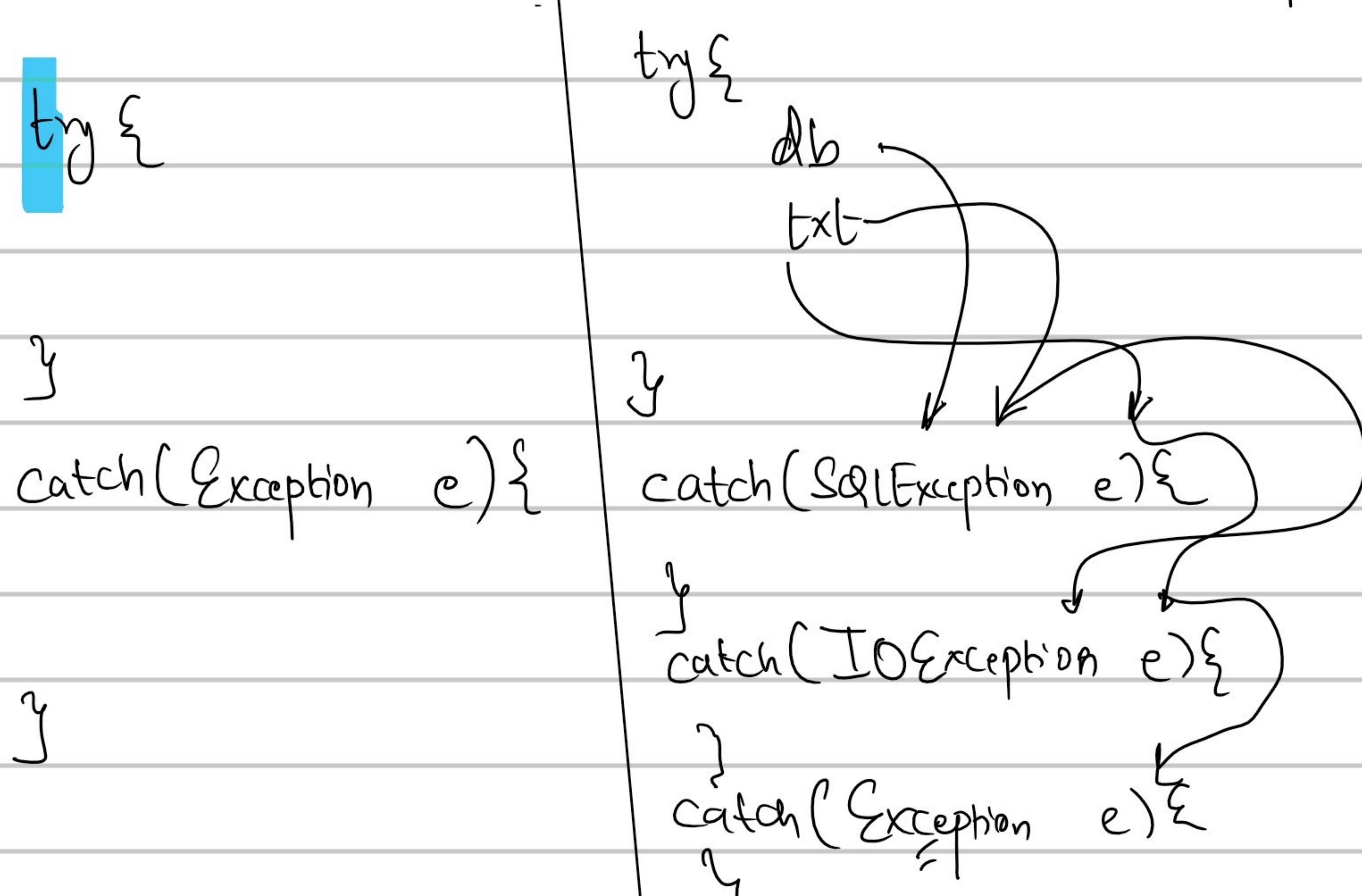
Checked Exception → Any subclass of java.lang.Exception except for

RuntimeException

UNCHECKED EXCEPTION ARE NOT REQUIRED TO BE
HANDLED OR THROWN

CHECKED EXCEPTION REQUIRES TO BE HANDLED / THROWN

Animal a = new Elephant();
Exception e = new SQLException();



Exception Message

`e.getMessage()` → String → what went wrong.

`e.printStackTrace()` → print everything from top to bottom

try {

}

catch () {

} finally {

}

house keeping job

in case of successful
run or exception

* Collection API *

Array

→ fixed type

→ no convenience methods

way to store the data.

→ allows you to store heterogeneous data

→ tons of convenience methods

→ `java.util`

`java.util.Collection`

List

Set

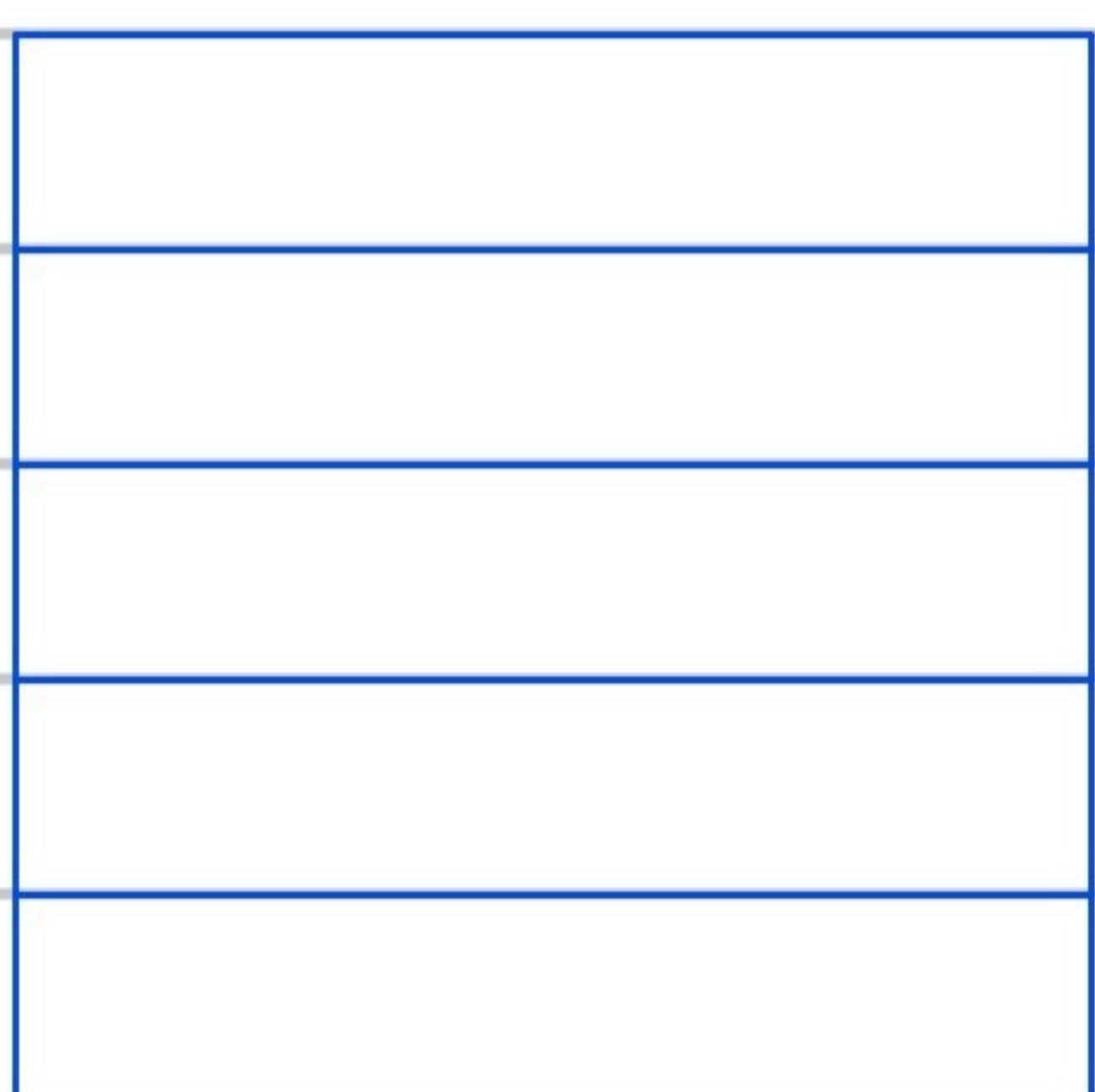
Queue

Map

List

Set

Queue



store duplicates
use an
array internally

does not allow
duplicates

FIFO

Set

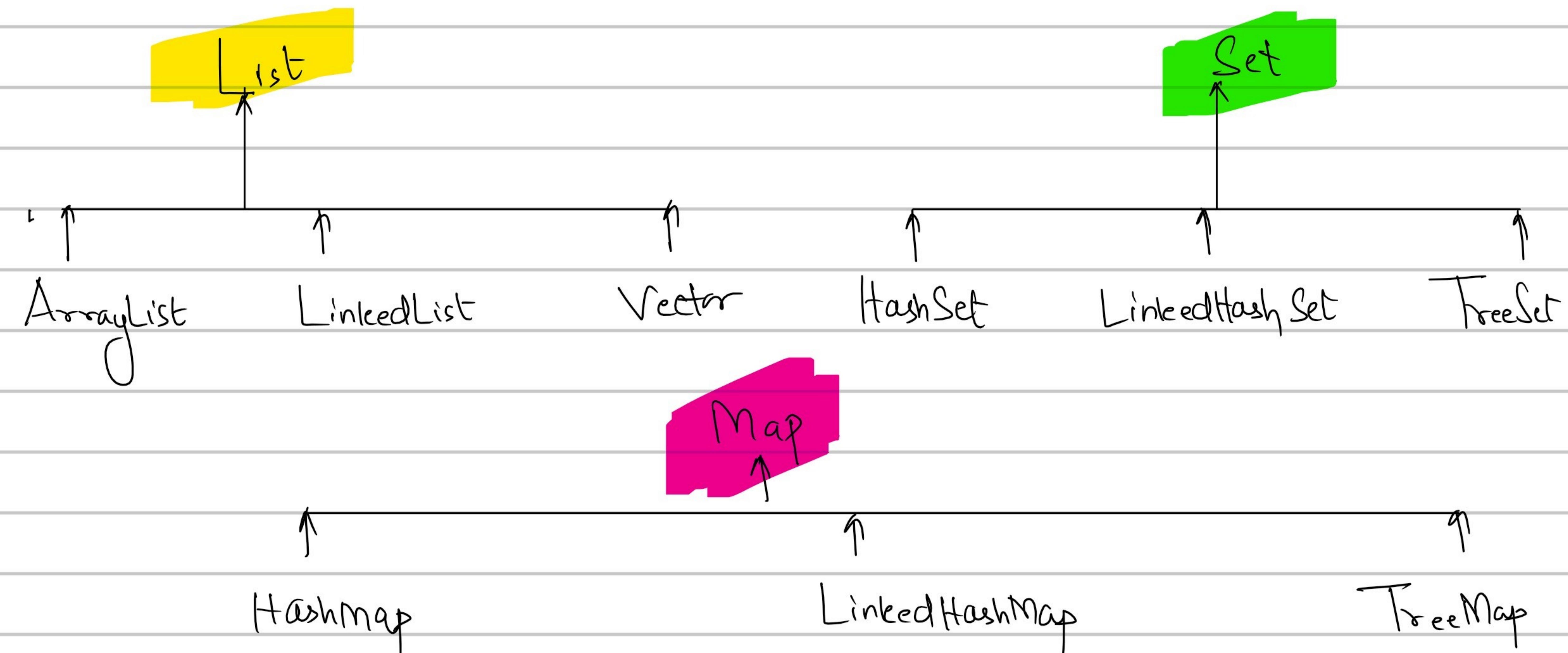
Map

Value

List

Set





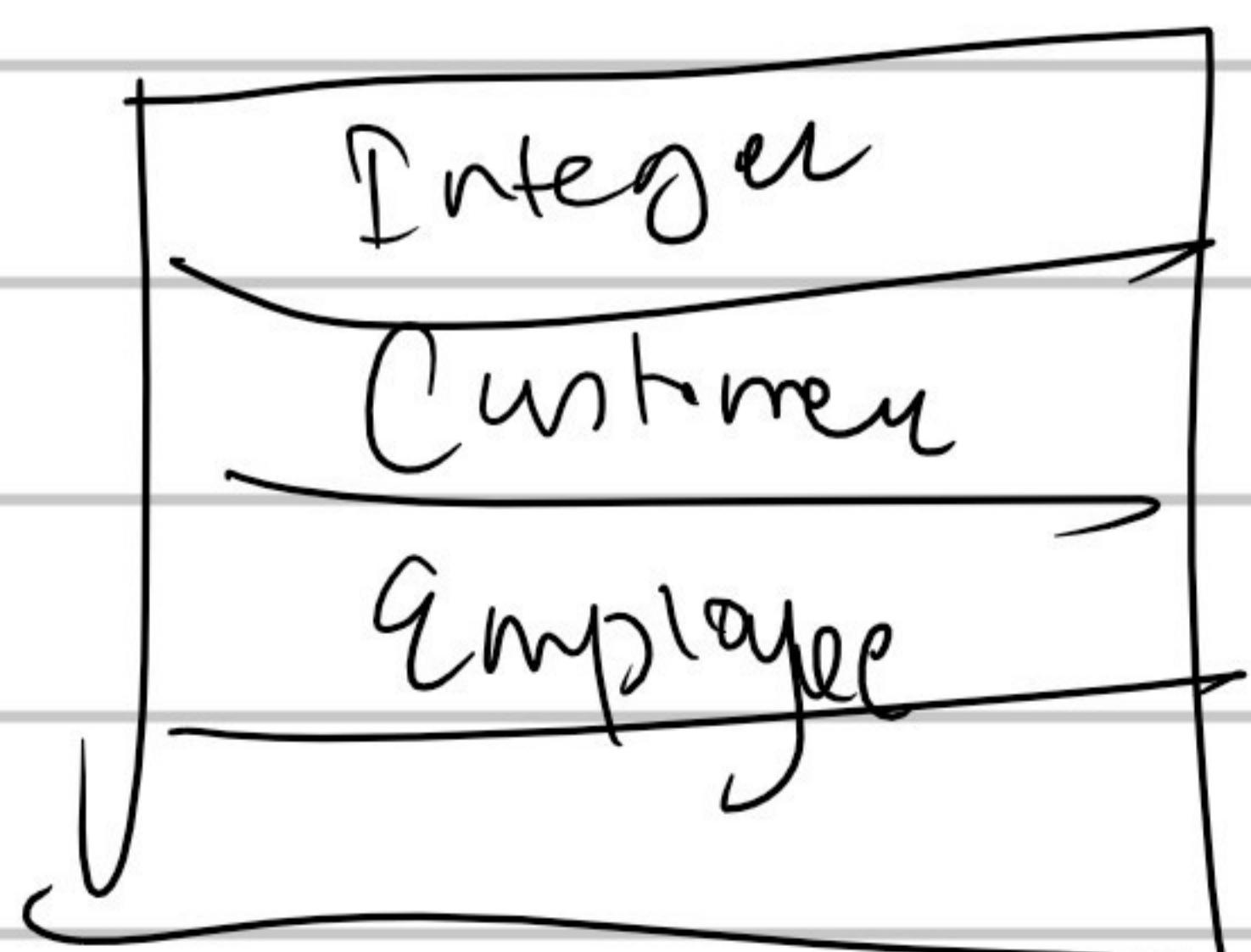
Hash → No order is maintained.
→ faster → hash algorithm

Linked → Order of insertion is maintained

Tree → Sorted by Natural key

ArrayList list = new ArrayList();

↳ can store heterogeneous data



Generics

ArrayList<String> list = new ArrayList<String>();

ArrayList<int> list = ~~X~~ —;

Wrapper

byte
boolean

char
short

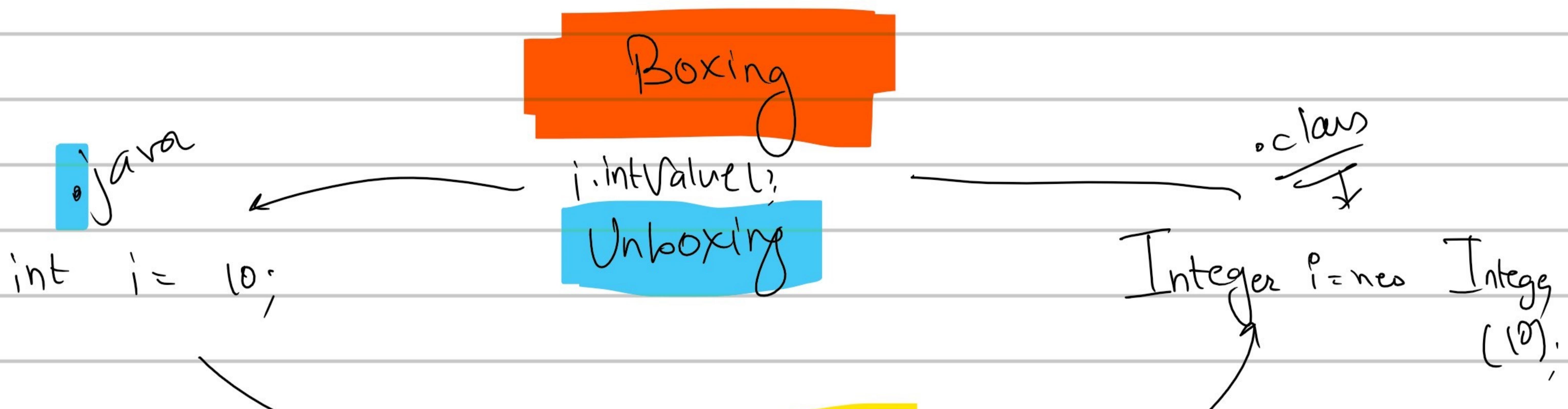
int
long
float
double

java.lang

Byte
Boolean
Character
Short

Integer
Long
Float
Double

ArrayList< Integer > list = new ArrayList< Integer >();
only hold integers



primitive
Box
Wrapper Class

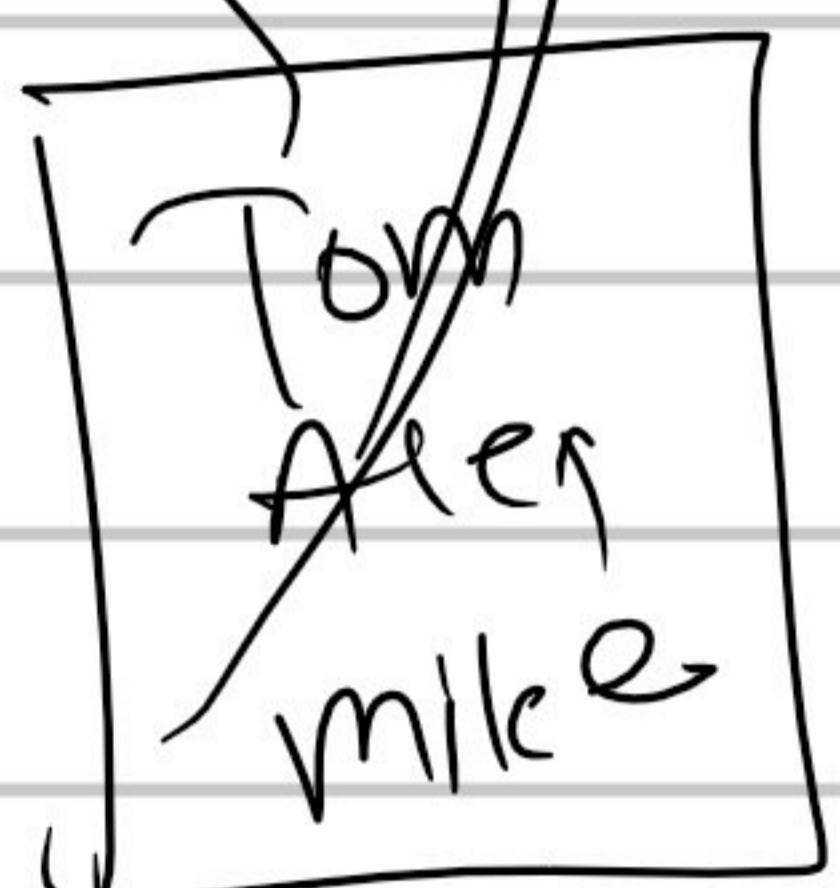
* Fetching the data *

* for each loop

for (Type element : collection) {

for (String name : namesList) {

 I ← (name);



Iterator → available for List, Set, Map, Queue
part of collection

List namesList = —

Iterator i = namesList.iterator();

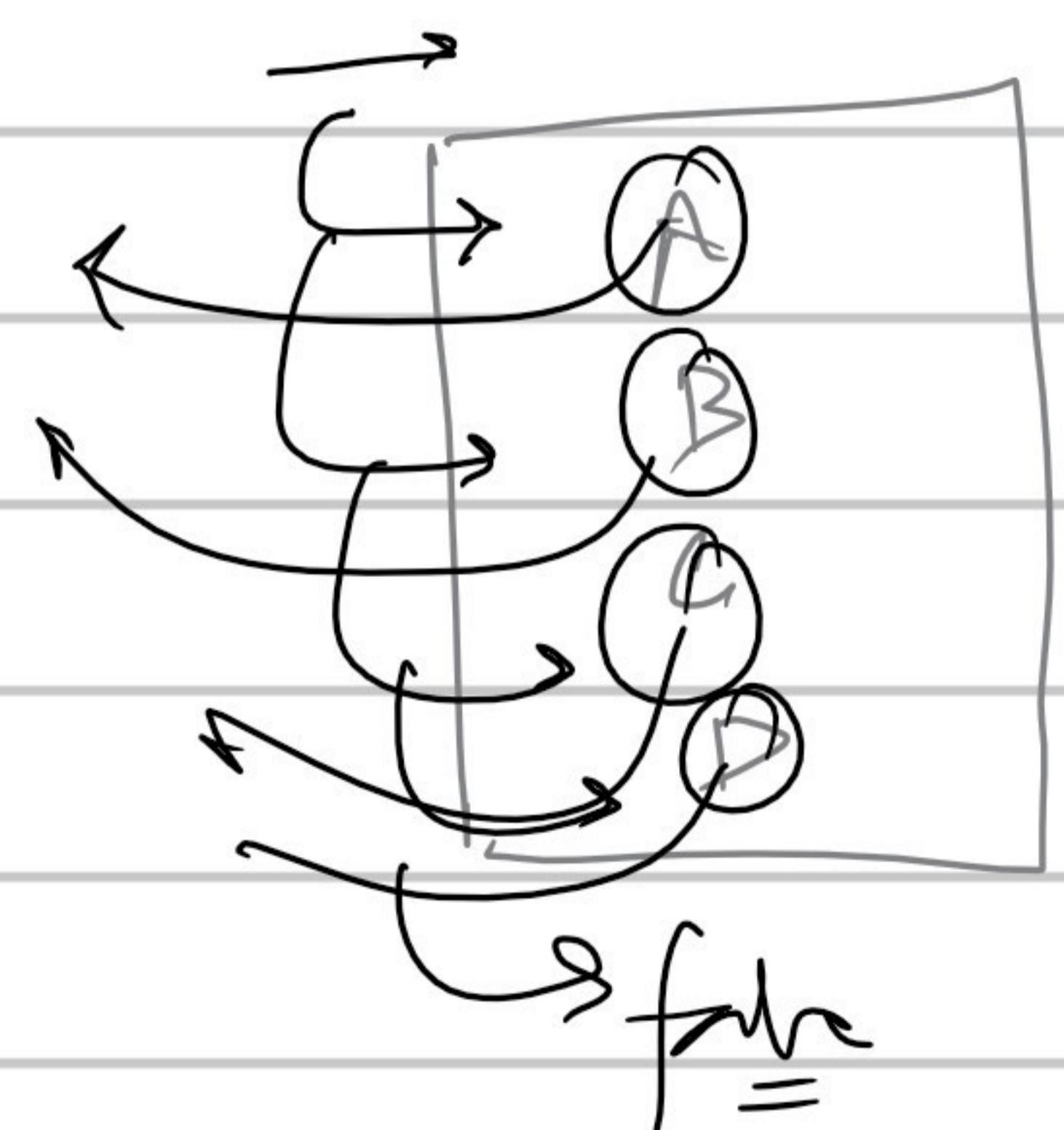
i.hasNext() → to check if collection have the element

i.next() → ref to the element

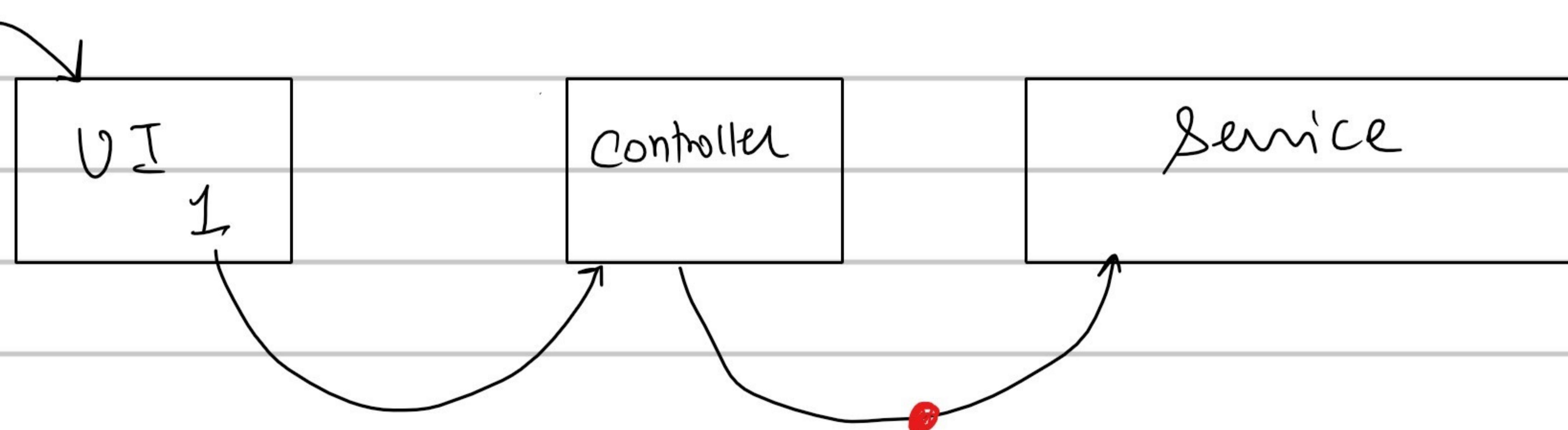
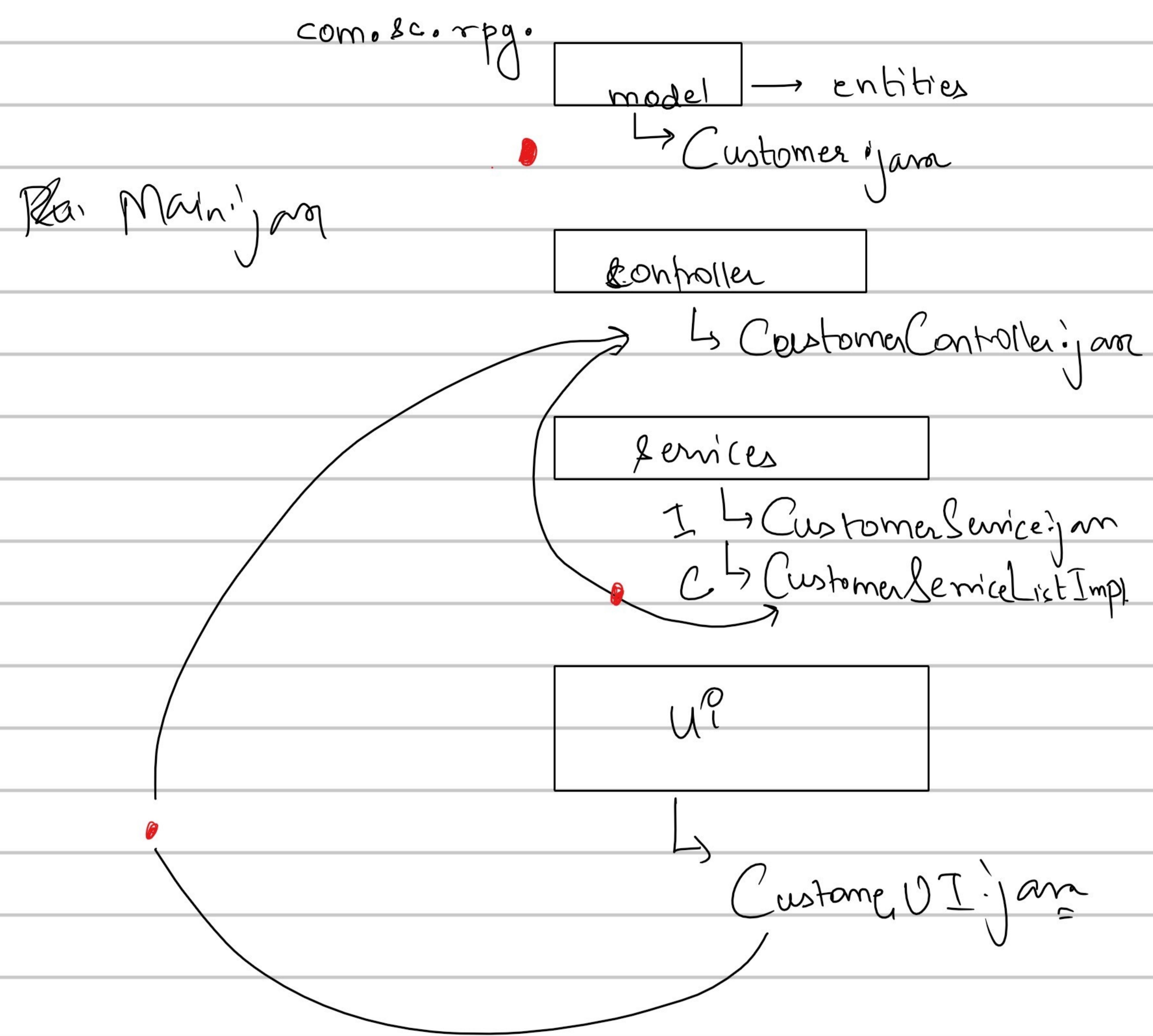
while (i.hasNext()) {

 String name = i.next();

 if



UML CASE



JDBC → Java Database Connectivity

→ API which helps to connect a Java application to the backend



RDBMS

→ JDBC can connect to any underlying RDBMS using appropriate driver.

Components of JDBC API

- Driver

→ Connection

→ ResultSet

- DriverManager ❤

- Statement

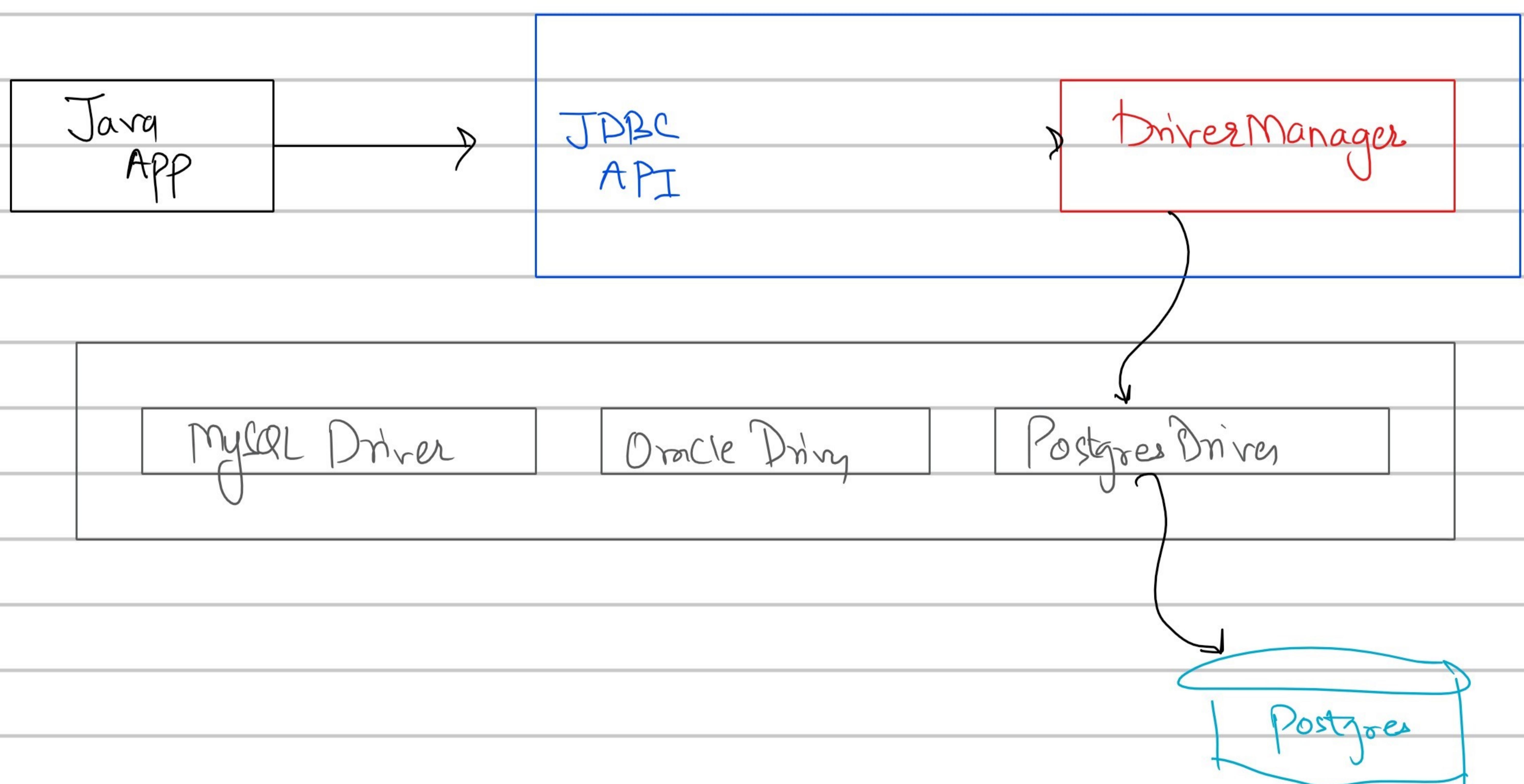
→ ResultSetMetaData

Java App

JDBC API

Driver

Postgres



Steps for Connecting JDBC with Database

Step 1: Import the required packages

- All the classes & interfaces required to work with JDBC API are in java.sql package

```
import java.sql.*;
```

Step 2: Load & Register the Driver

- Driver (provided by RDBMS) has to be loaded explicitly by calling `forName()` in `java.lang.Class`

```
Class.forName("org.postgresql.Driver");  
("com.mysql.jdbc.Driver")
```

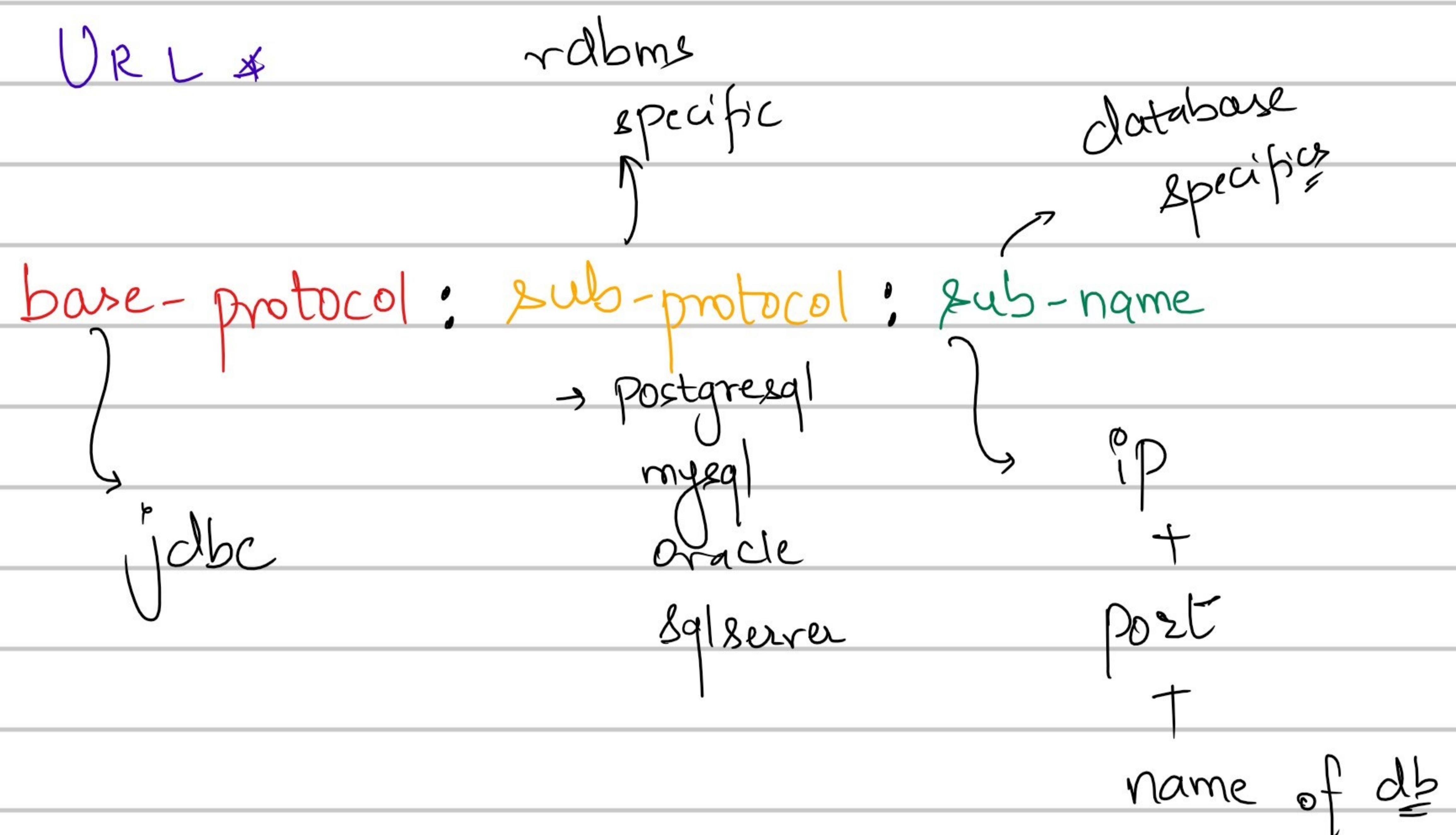
} The loaded Driver get registered with DriverManager implicitly.

Step 3: Establish Connection

- * Connection is like a session with database.

- To acquire this connection ; we need to call `getconnection()` of Driver by using JDBC URL

* JDBC URL *



```
String url = "jdbc:postgresql://localhost:5432/scb-rpg-2022";
```

```
String username = "postgres";
```

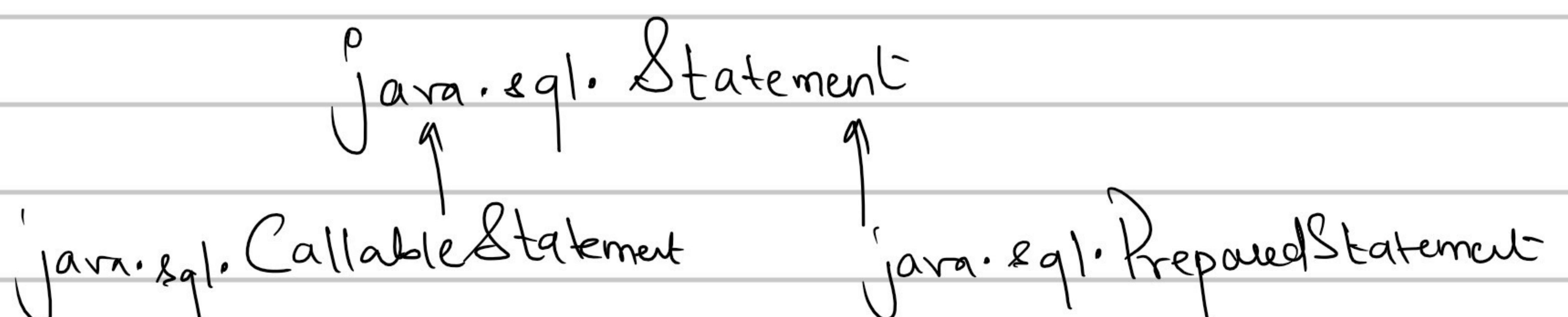
```
String password = "admin";
```

```
Connection con = DriverManager.getConnection(url, user, password);
```

Step 4: Create a Statement

→ Statement is like a container to pass a SQL Query.

→ Statement are of 3 types



To acquire a statement there are multiple ways.

Statement `stmt = con.createStatement();`

query → compile → 100
execute → 100

PreparedStatement `pstmt = con.prepareStatement(query);`

query → compile → 1
executed → 100

CallableStatement `cstmt = con.prepareCall(query);`

Query for prepared statement comes with placeholders

String query = "Insert into customer (name, email, contact, account_type)
values (?, ?, ?, ?)";

PreparedStatement `pstmt = con.prepareStatement(query);`

Step 5 : Execute the Query.

SQL statements are of 3 types

DDL → Data Definition Language → CREATE, ALTER, DROP

DML → Data Manipulation Language → INSERT, UPDATE, DELETE

DQL → Data Retrieval Language → SELECT

Type	Method	Return
DDL	<code>pstmt.execute();</code>	boolean false → successful true → unsuccessful
DML	<code>pstmt.executeUpdate();</code>	int ↳ number of rows affected
DQL	<code>pstmt.executeQuery();</code>	ResultSet ↳ output of SQL query in table format

* Prepared Statement *

```
String name = "Tom";
String email = "tom@gmail.com";
long contact = 7715012345
```

String query = "Insert into customer (name, email, contact, account_type)
values (?, ?, ?, ?);" ; String aType = "Current";
 1 2 3 4

PreparedStatement pstmt = con.prepareStatement(query);

```
pstmt.setString(1, name);
pstmt.setString(2, email);
pstmt.setLong(3, contact);
pstmt.setString(4, aType);
```

String query = "Update customer set email=? where id=?";
 1 2

PreparedStatement pstmt = con.prepareStatement(query);

pstmt.setString(1, "tom@outlook.com");

pstmt.setInt(2, 3);

Step 6: Fetch the Results.

6.1 → DDL

String query = "Create table test (id int, name text);"

Prepared Statement pstmt = con.prepareStatement(query);

boolean status = pstmt.executeUpdate();

6.2 → DML

String query = "Insert into test values (?, ?)";

Prepared Statement pstmt = con.prepareStatement(query);
pstmt.setInt(1, 1000);
pstmt.setString(2, "Tom")

int rowAffected = pstmt.executeUpdate();

6.3 → DRL

customer

	id	name	email	contact	account_type
	1	Tom	t@gmail.com	12345	S
	2	Alex	a@g-	12346	C
	3	Mike	m@g-	12347	S
	4	John	j@g-	12346	C

String query = "Select name, contact, email, account_type from customer";
Prepared Statement pstmt = con.prepareStatement(query);

ResultSet rs = pstmt.executeQuery();

ResultSet → represents a Java object which has a table inside it & it contains the result returned by the query,
→ It has cursor on it.

before-first
first
last
after last

* → when next() / previous() is called, cursor moves accordingly
→ it checks if the pointed row has value or not

	Tom	12345	t@ →	C
	Alex	12346	a@g →	C
	Mike	12347	m@g →	S
	John	12348	j@g →	C

Navigations

Cursor is always placed before first

rs.next();
rs.previous();

rs.first();
rs.last();

Fetching

rs.getString(colName); rs.getInt(colName);

rs.getFloat();

while(rs.next()) {

String name = rs.getString("name");

String email = rs.getString("email");

long contact = rs.getLong("contact");

String accountType = rs.getString("account_type");

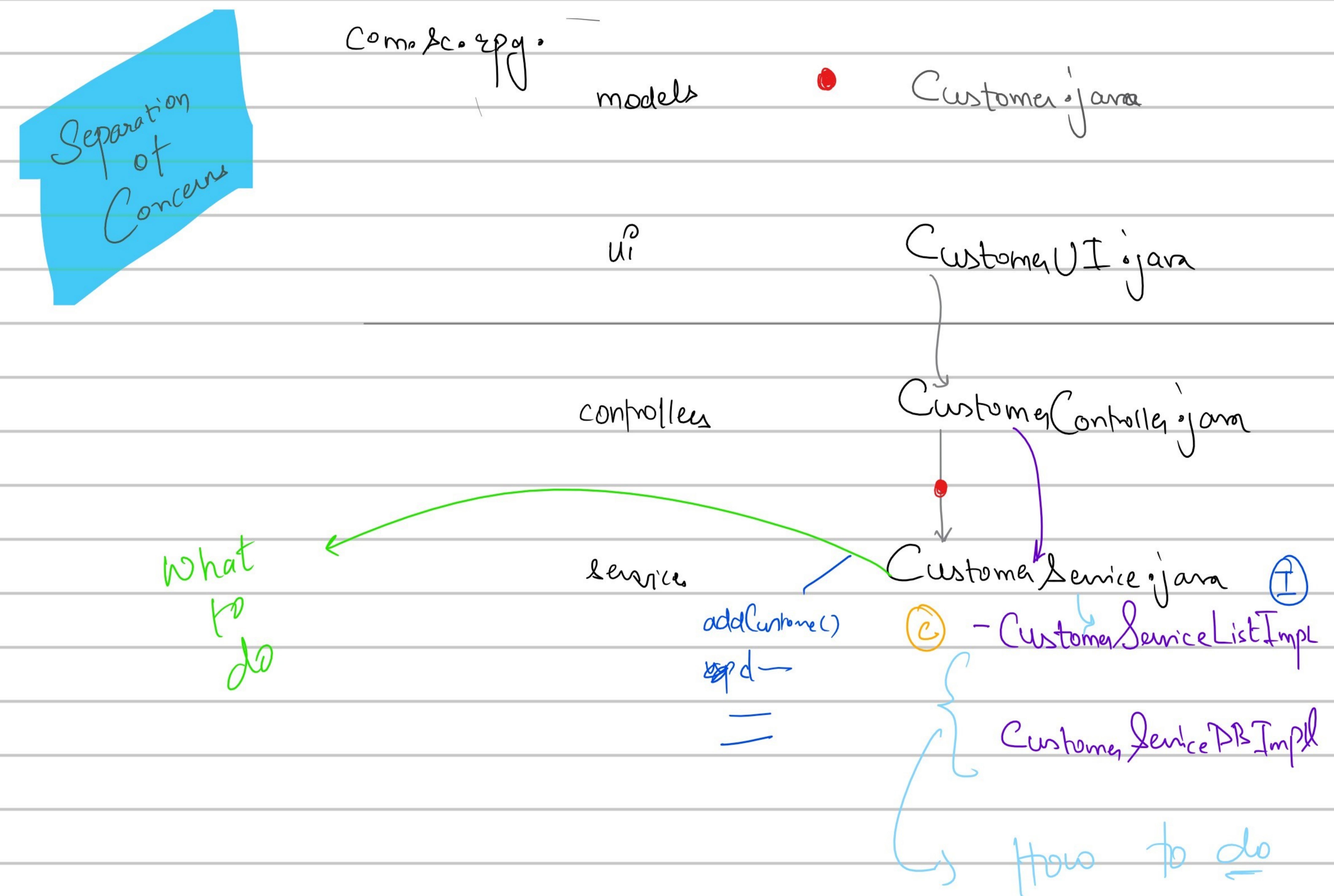
}

Step 7: Close the Connection

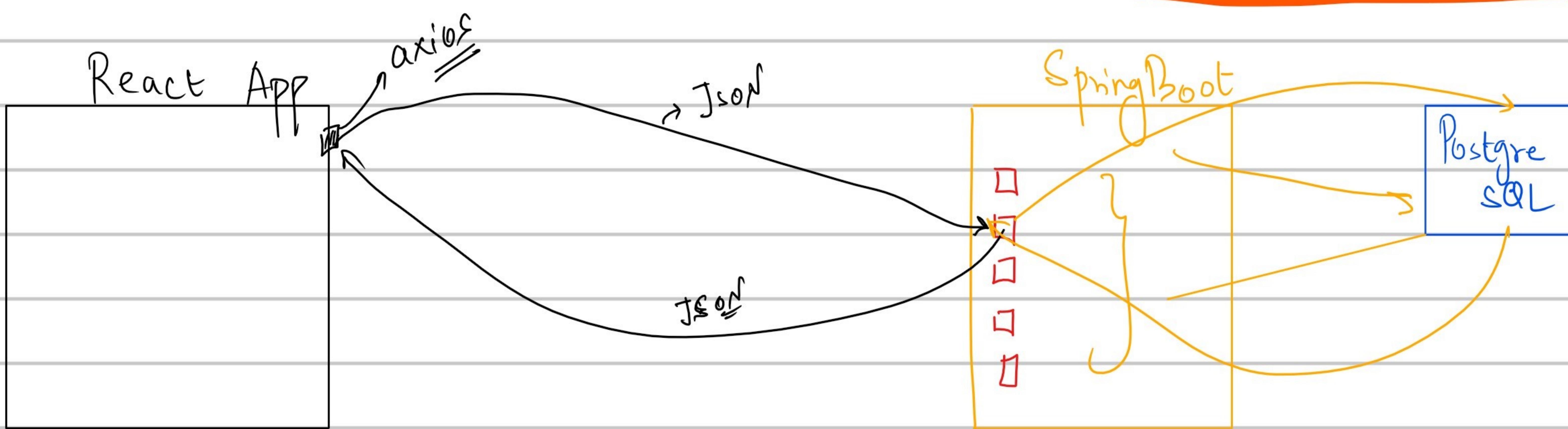
```
stmt.close();
```

```
con.close();
```

Layered Architecture



REST API



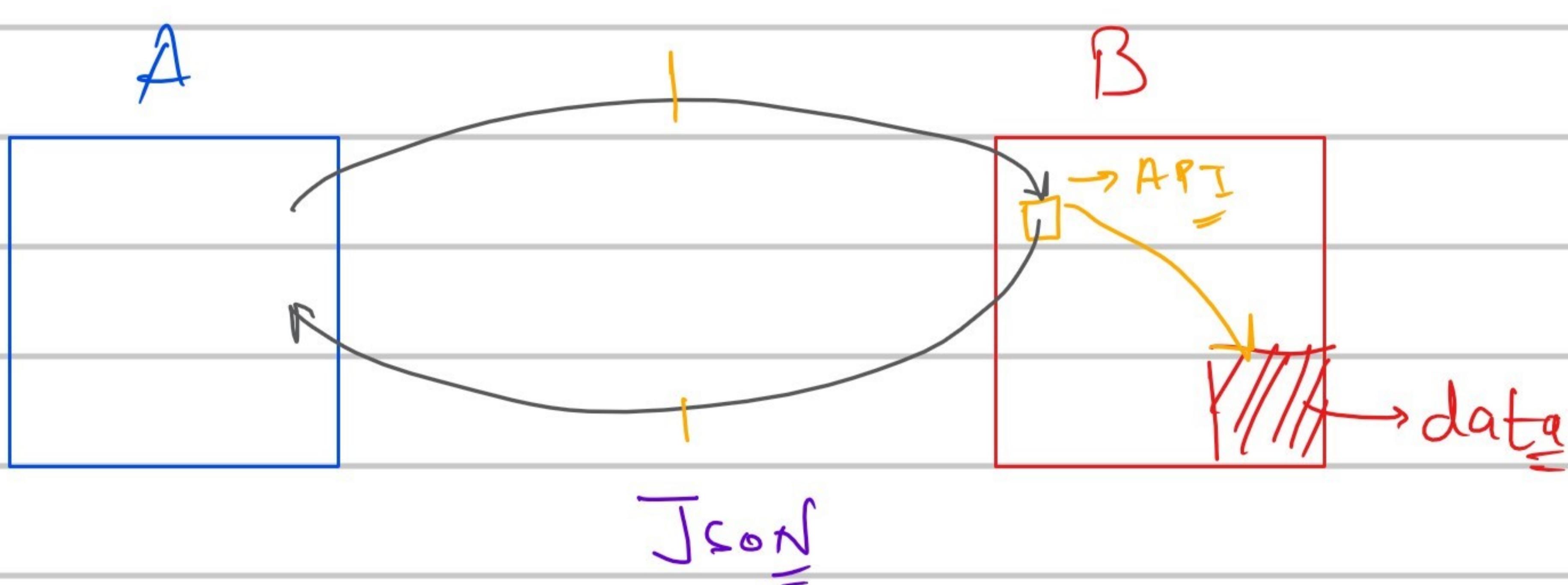
REST API

An API is a mechanism that enables an application or a device to access a resource in another application.

Representation

State

Transfer



- * Spring Boot
- * Node JS
- * Django
- * ASP
- * AWS serverless functions

* Types of HTTP REQUESTS *

Omnipotent

Action

Http Request Type

* Fetch the data from backend

GET

* Store some data to backend

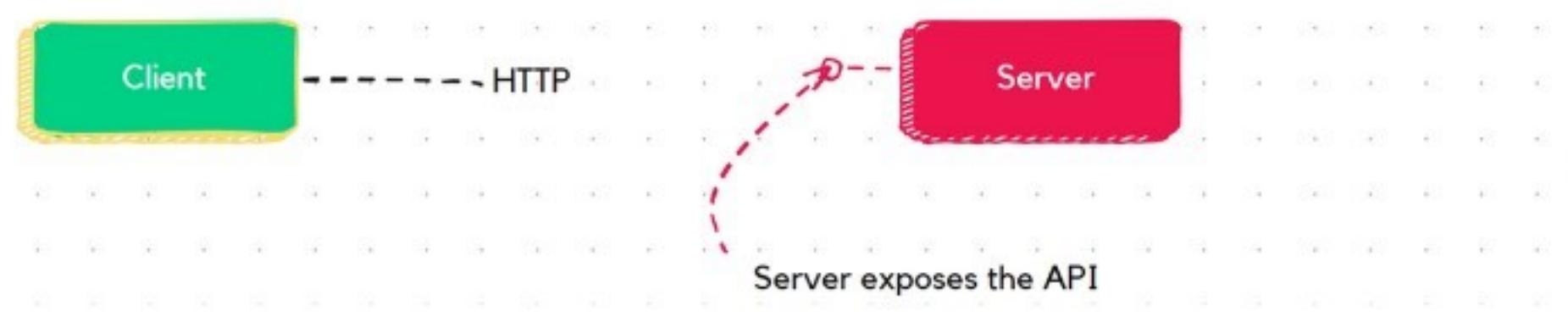
POST

* UPDATE some existing data

PUT / PATCH

* DELETE some existing data

DELETE



url
http://vidly.com/api/customers / videos
protocol
RESOURCE

HTTP METHODS / VERBS

HTTP METHODS

GET

POST

PUT

DELETE

customer

GET api/customers
api/customers/3

GETTING A CUSTOMER

Request

GET api/customers

• 200

• 404

• 500

Response

```
[  
  {id:1,name:.....},  
  {id:2,name:.....}  
]
```

• Success

• Failure

• Server Failure

Idempotent: ✓

Request

GET api/customers/1

• 200

• 404

• 500

Response

```
{id:1,name:.....}
```

CREATING A CUSTOMER

Request

POST api/customers

Request Body

```
{id:1,name:.....}
```

• 200, 201

• 404

• 409 (conflict)

• 500

Response

```
{id:1,name:.....}
```

Create a new resource

Idempotent: X

UPDATING A CUSTOMER

Request

PUT api/customers/1

Request Body

```
{id:1,name:.....}
```

• 200, 201, 204

• 404

• 500

Response

```
{id:1,name:.....}
```

update the resource

Idempotent: ✓

DELETING A CUSTOMER

Idempotent: ✓

Request

DELETE api/customers/

DELETE api/customers/1

- 200
- 404
- 500

Response

{id:1, name:....}

delete the resource
with the given
Identifier

- 200 - OK - (GET Request)
- 201 - Created - (POST Request)
- 301 - Moved Permanently
- 302 - Moved Temporarily
- 304 - Not Modified
- 400 - Bad Request
- 401 - Unauthorised
- 403 - Forbidden
- 404 - Not Found
- 405 - Method Not Allowed
- 408 - Request Timeout
- 500 - Internal Server Error
- 503 - Service Unavailable
- 504 - Gateway Timeout

Idempotent

Post
/api/customers

x 100 → 100

X

PUT
/api/customers/4

x 100 → 1

✓