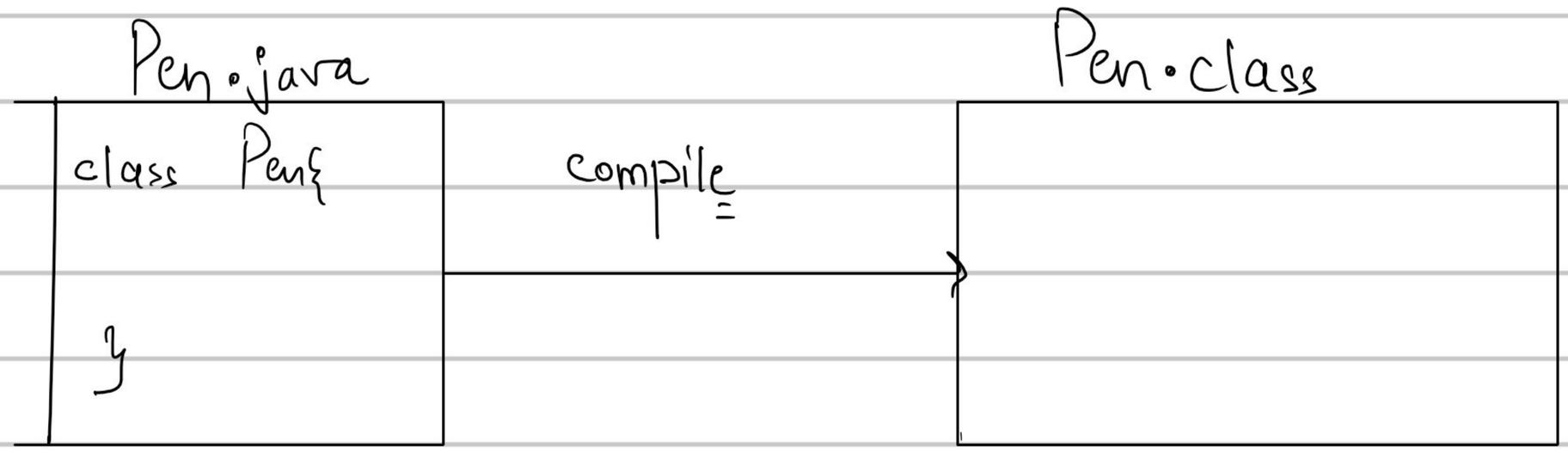
Java

→ object oriented programming language.

Everything in Java is an object.



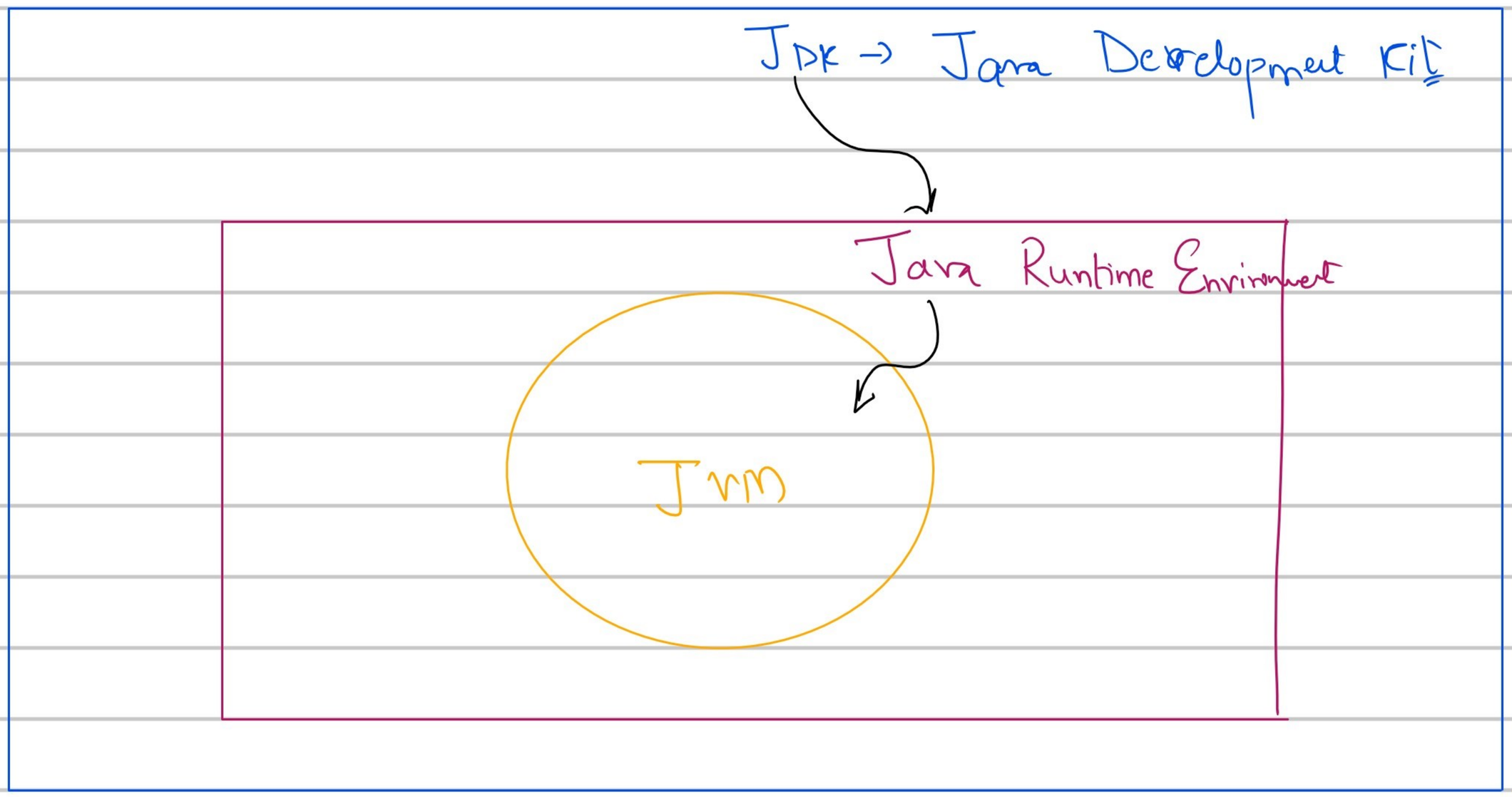
Components of Java Environment.

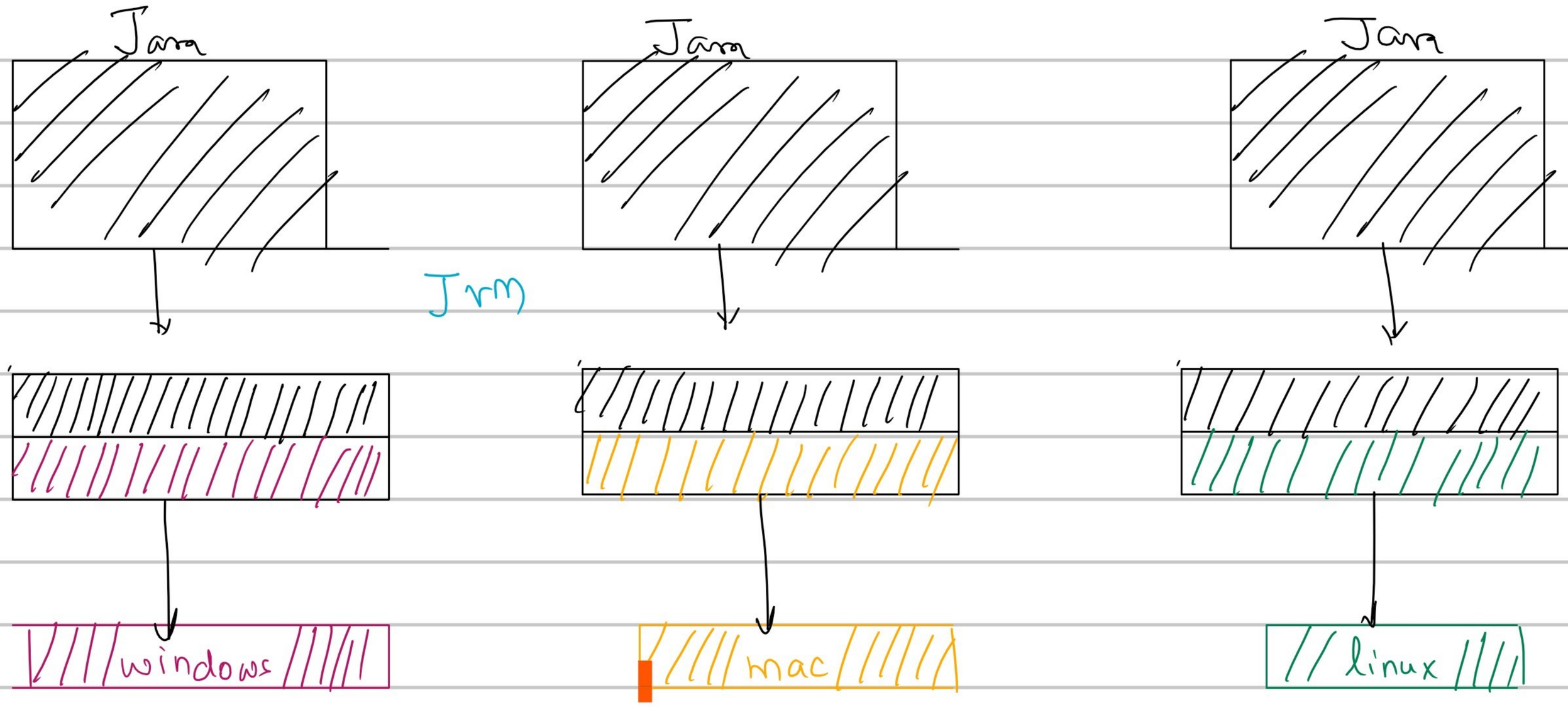
JDK → Java Development Kit

JRE → Java Runtime Environment

JVM → Java Virtual Machine

virtual machine
↓
a machine that runs inside another machine.





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

Pascal Casing → OneTwoThreeFour

Class / Interface

Methods / Variables

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 1.4×10^{-45}
double	real	8	1.7×10^{308} to 4.9×10^{-324}

Control statements

Operators

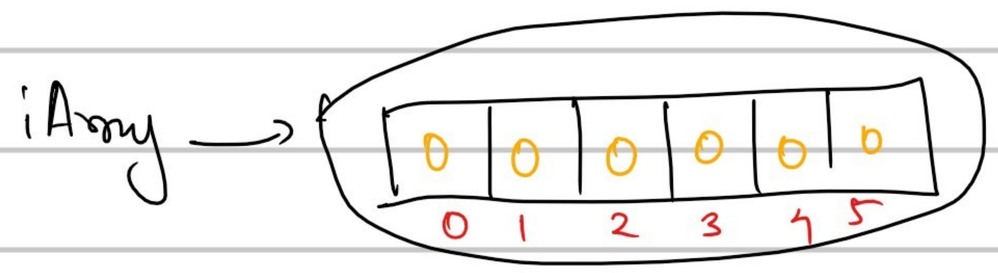
Loops

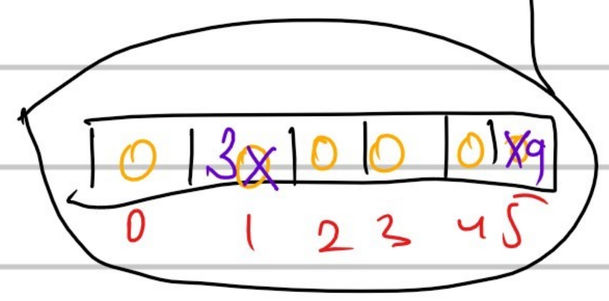
Arrays

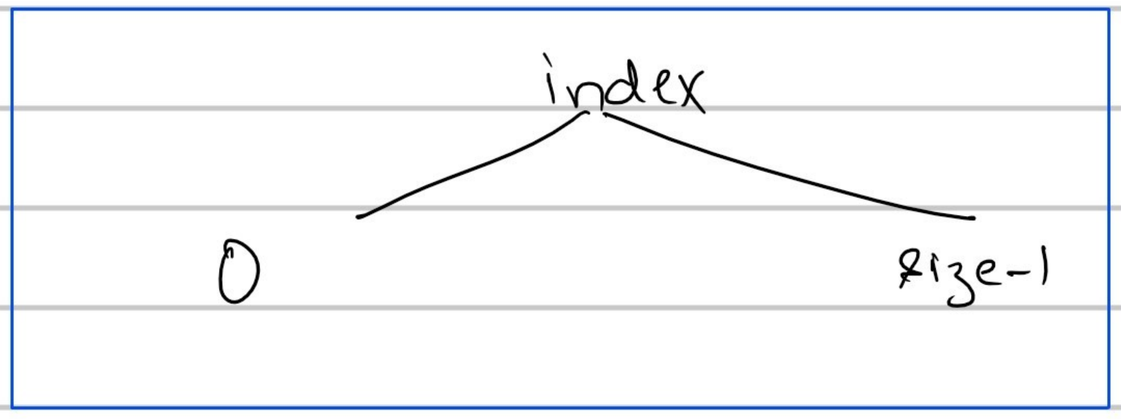
Arrays in Java are object who has a container inside it which can hold elements of a similar type.

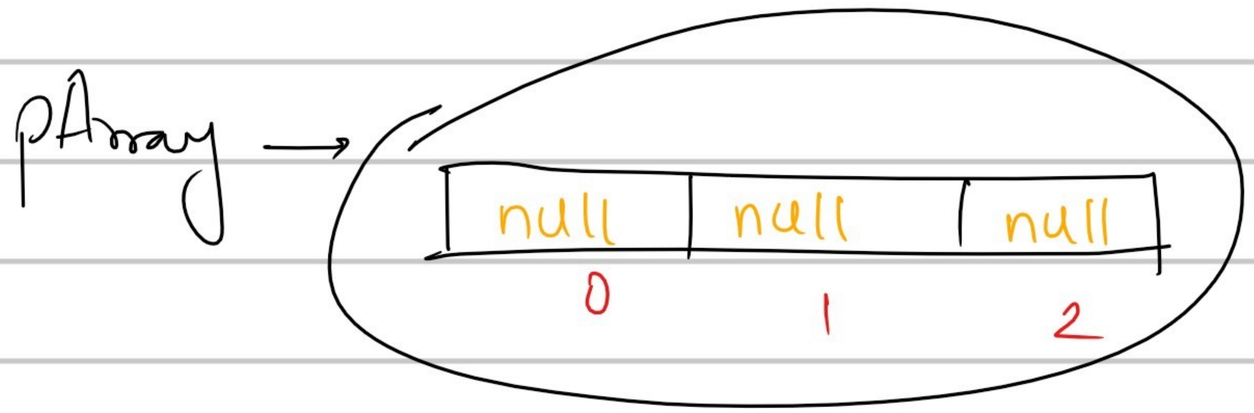
type → primitive data type / class

DECLARE	CONSTRUCT	INITIALIZE
type[ ] nameOfArray;	nameOfArray = new type[size];	nameOfArray[index] = value;
int[ ] iArray;	iArray = new int[6]	iArray[1] = 3;
Pen[ ] pArray;	pArray = new Pen[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.

Pen
Car
Oxygen Level
Air Pressure

ability to be measured

→ 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%.

Car

State	Behavior
• model	start()
• color	stop()
• engine	run()
• transmission	fillTank()
• fuelLevel	changeGears()

100% ↓ 90% ↓ 50%

E-Commerce

* 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;
```

```
String s;
```

```
Pen p;
```

p is an object of Pen

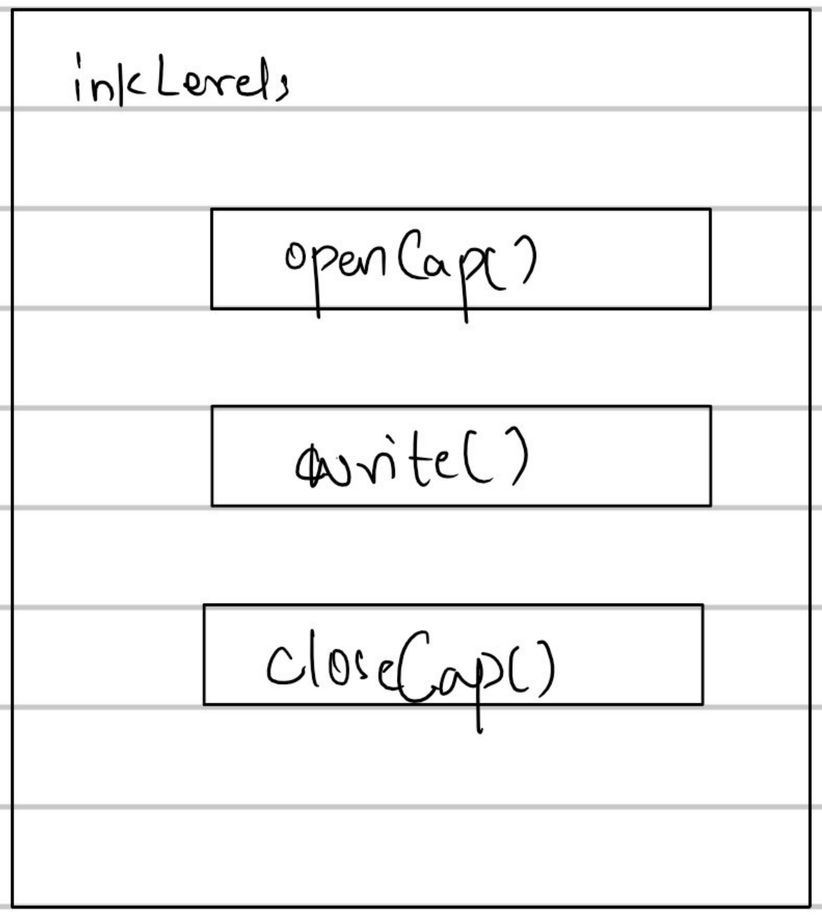
```
Can c;
```

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

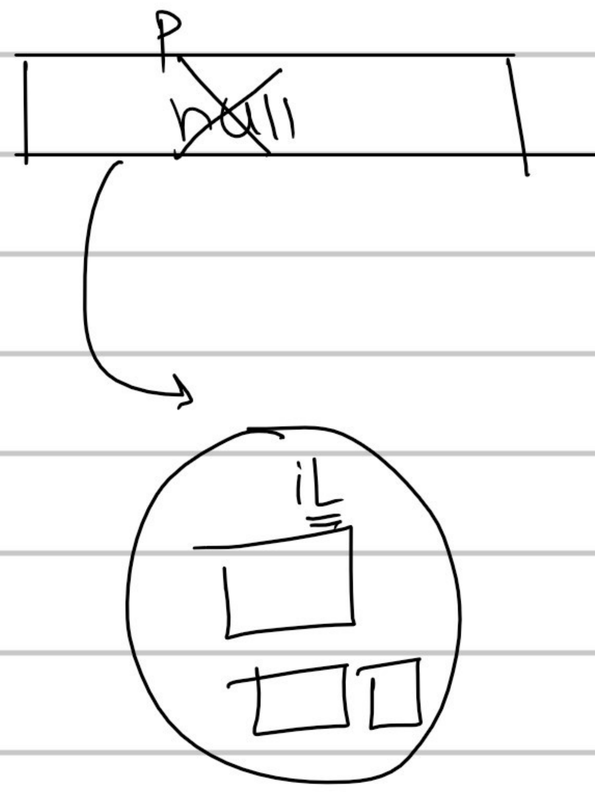
p.write(); ✗ → object is not instantiated

p = new Pen();

p → 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.

STRUCTURE OF A METHOD

public private
protected default final, static, abstract

access-specifier ▾ access-modifier [optional] ▾ return-type ▾ methodName (parameters [optional])

can be any valid identifier.

return-type: ① void ② primitive class

parameters: how methods takes input.

```
{
    statement(s)
    —
    —
    —
    last-line
}
```

if a method doesn't return anything (void) there is no obligation on last line.

but if a method returns something; the last line has to be a 'return' statement

```
class MyMethods {

    public void display() {
        Sy____("SCB");
    }

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

}
```

No Parameter
No Return Type

Parameters
No Return Type

```
private int getTaxRate() {
    return 18;
}
```

No Parameter
Return Type

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

Parameter
Return Type

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();
    obj.add(1,2);
int tax_R = obj.getTaxRate();
int sq = obj.square(5);
```

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) ✓

Car → car() / car(x) ✗

② No return type (not even void)

* Constructor are of two types *

① Default ② Parameterized

```
class Pen {
    Pen() {                    → Default
    }
    Pen(String color){         → Parameterized
    }
}
```

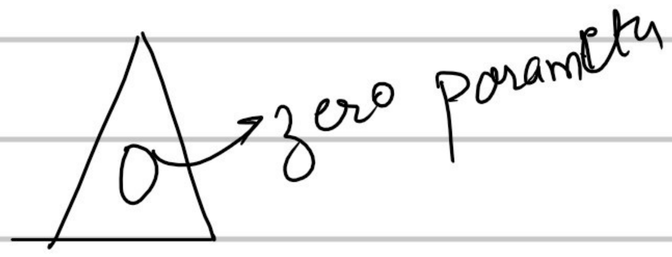
* Constructors are invoked when an object is instantiated.

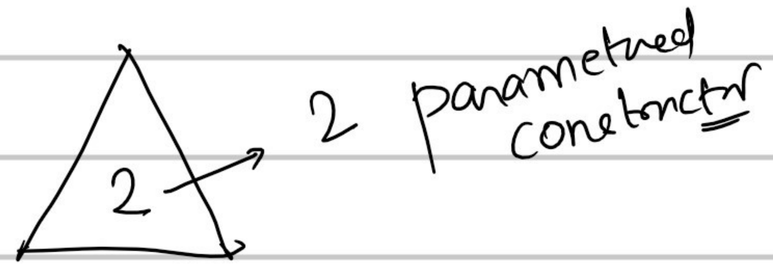
Constructor defines the state in which object will be created.

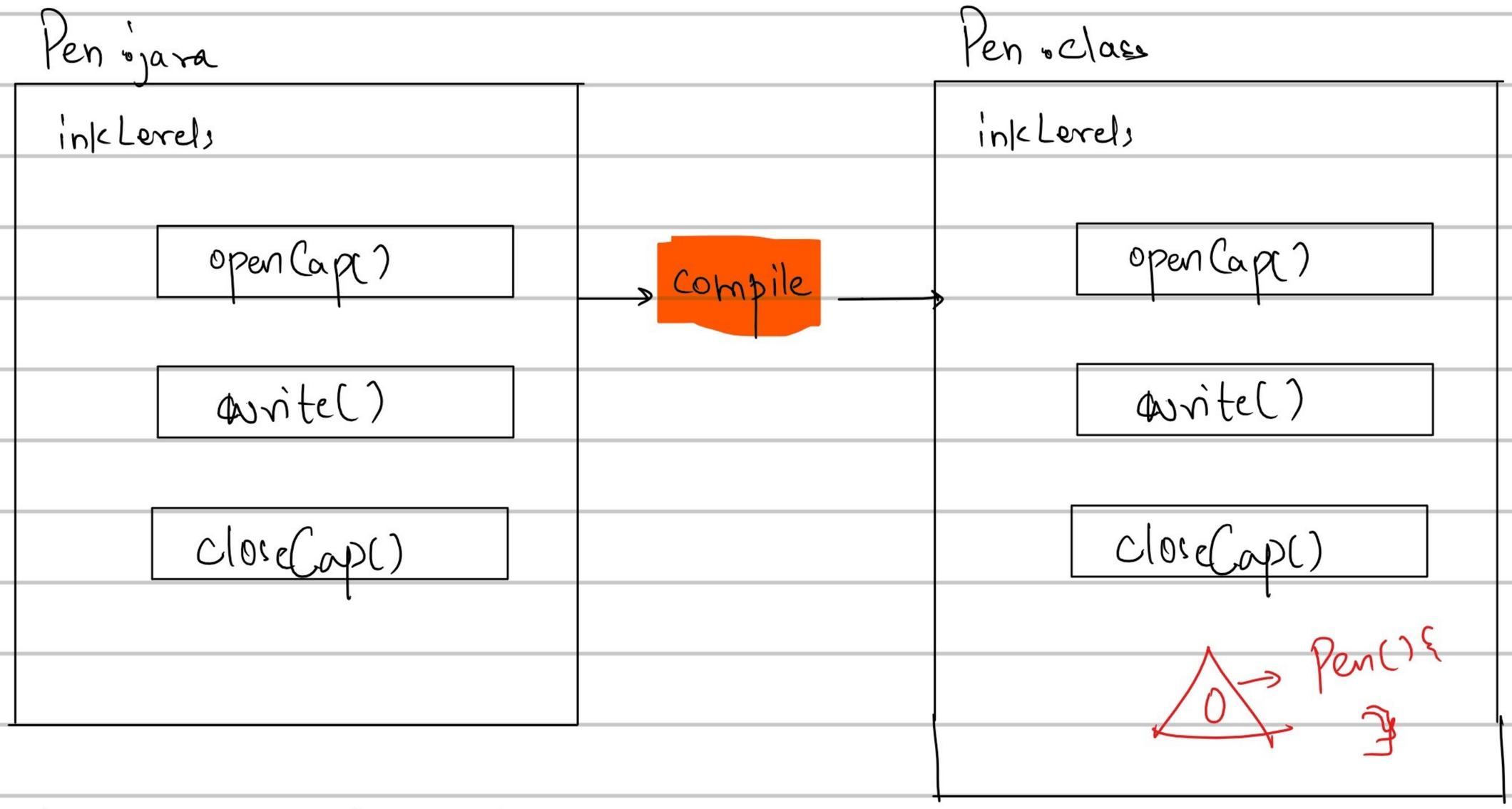
Constructor	Methods
→ invoked while object is getting ready	→ invoked after an object is ready

Tom gets ready & go to the office & work

Constructor → methods

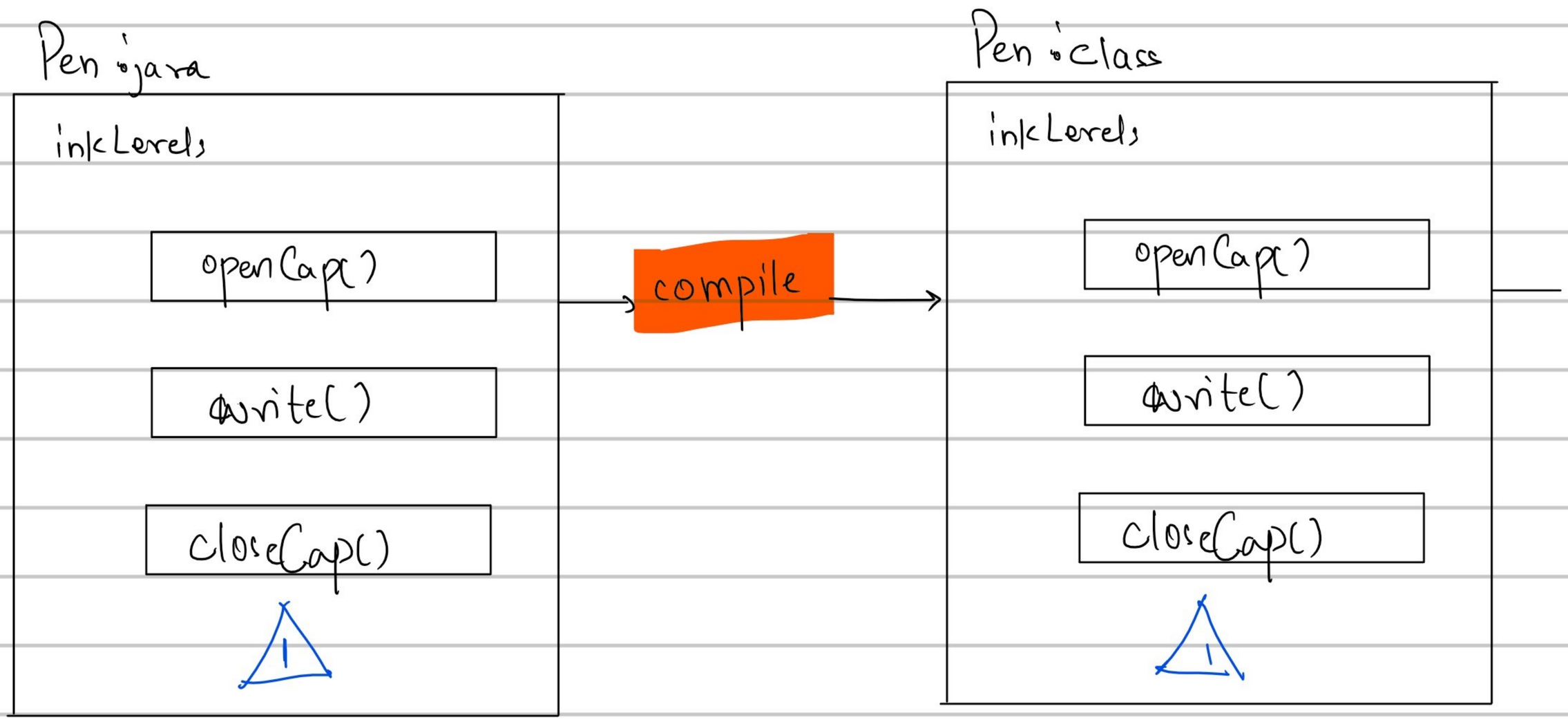






Whenever a .java file is compiled & if it doesnot have any constructor definition inside it; Java Compiler provides a free default constructor in its .class file

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



State

variables → placeholder for data whose value can change

- Primitive
- Object with HAS A relationship

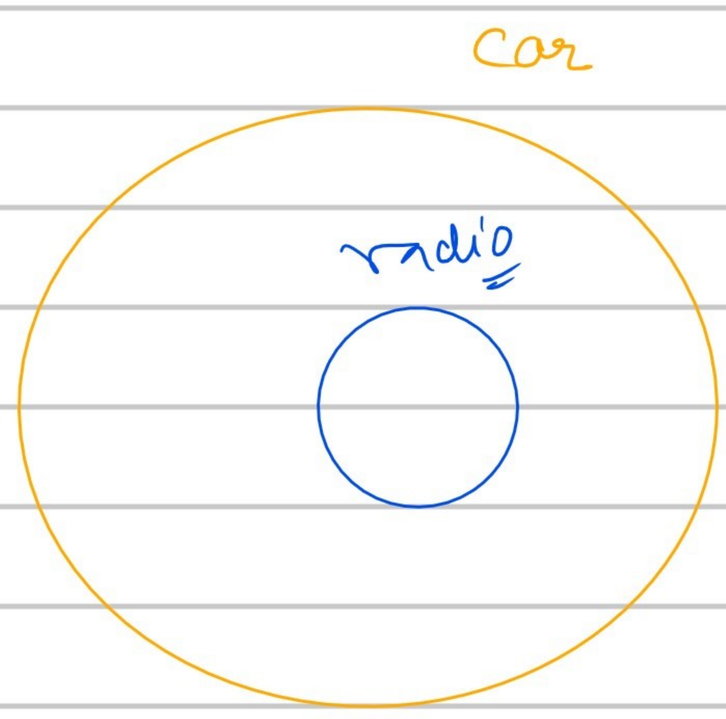
* Primitive Variables

↳ are those variables which are declared using datatypes

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(){
    S.out.(("R "+volume);
  }
}
```

```
class Car {
  String name;
  int fuelLevel;
  Radio r=new Radio();

  public void run(){

  }
}
```

O

P

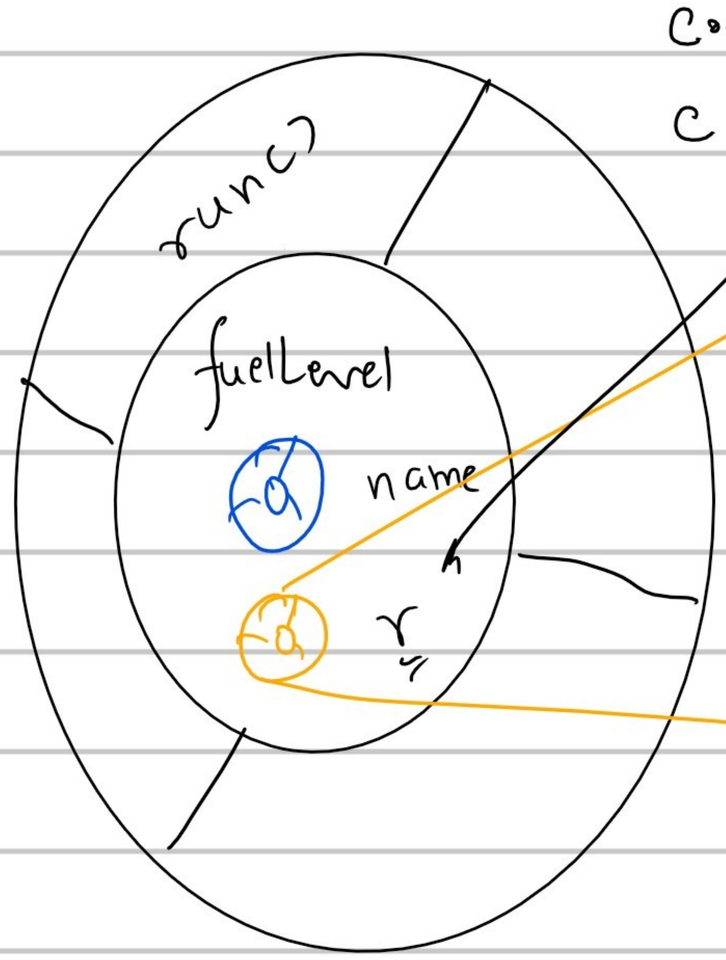
O

Car c = new Car();

c.fuelLevel;

c.r.play();

c.r.volume;

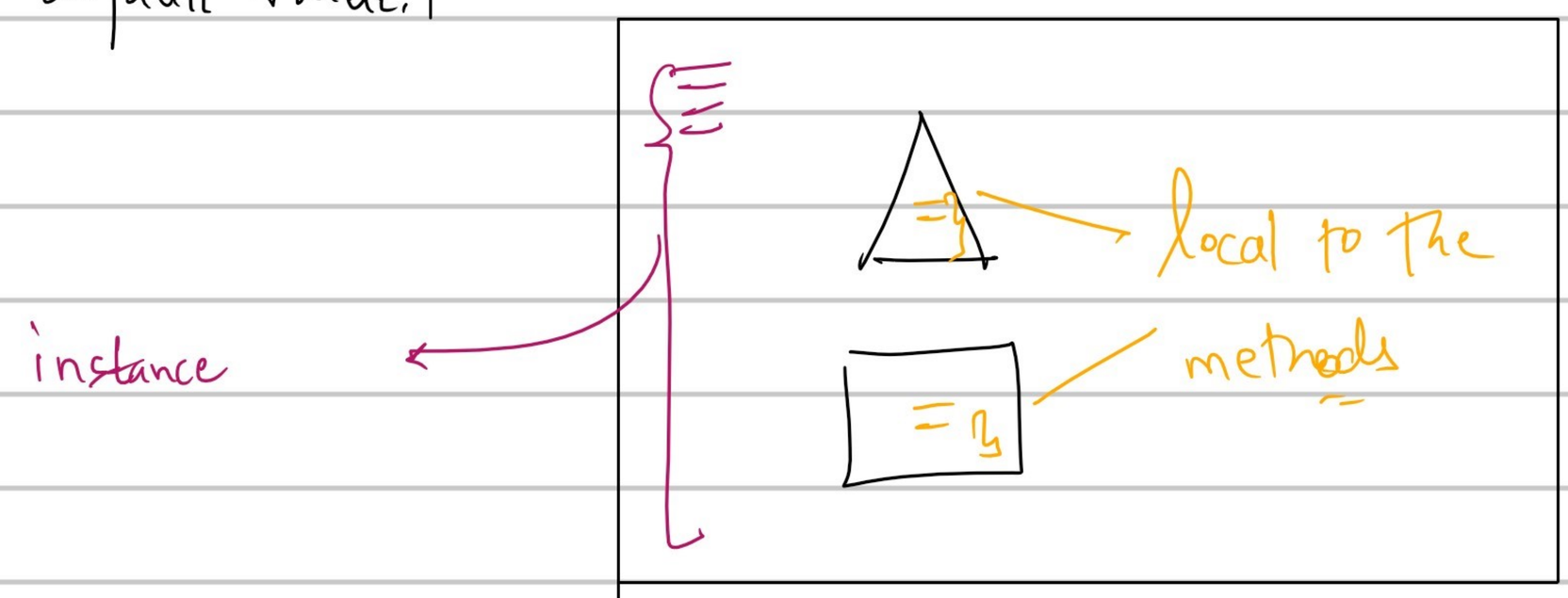


Based on the position of the variables,

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
Scope	
→ available throughout the class.	→ available in the constructor/method where it is declared
Declared	
→ int i; If a variable is declared Pen p; In instance area it is assigned to its default value.	→ if left declared → will result in error



Local & Instance Variables can have same name

```
class Circle {

    int radius;

    Circle(int radius) {
    this.radius = radius;
    }
    public void setRadius(int radius) {
    this.radius = radius;
    }
}
```

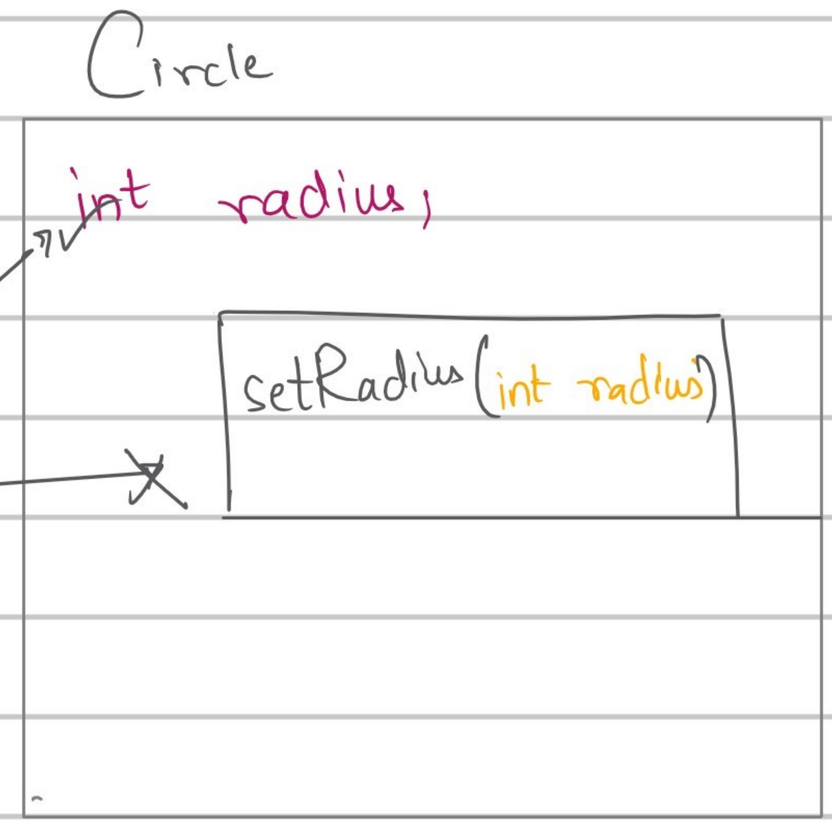
instance

local

ambiguity

local

```
Circle c = new Circle();

c.radius;
```

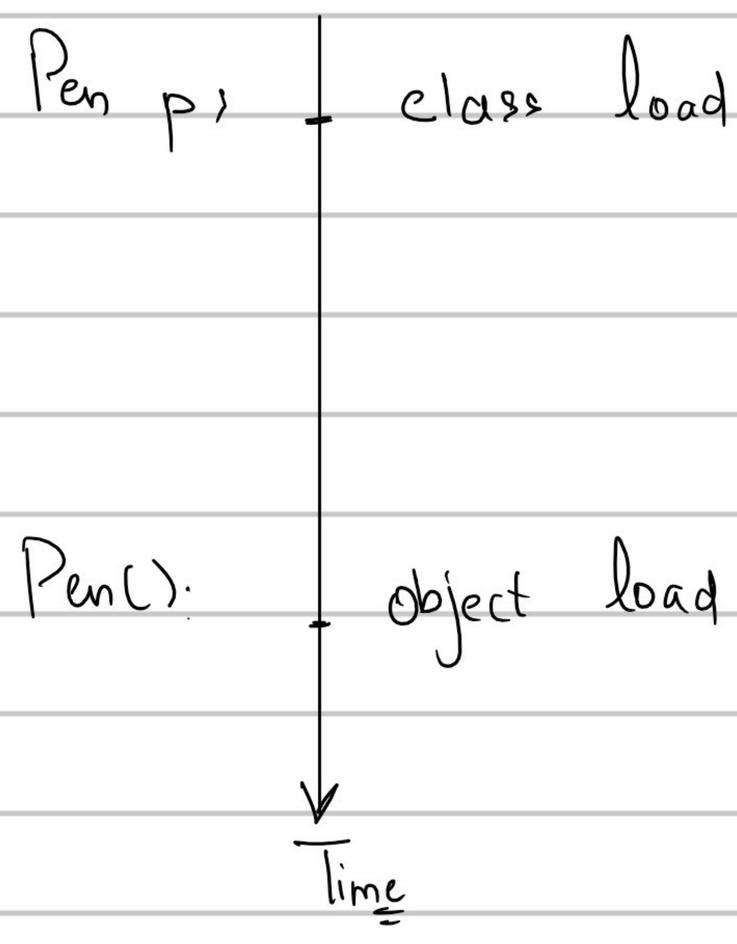


this keyword
↳ points to an instance of same class

---

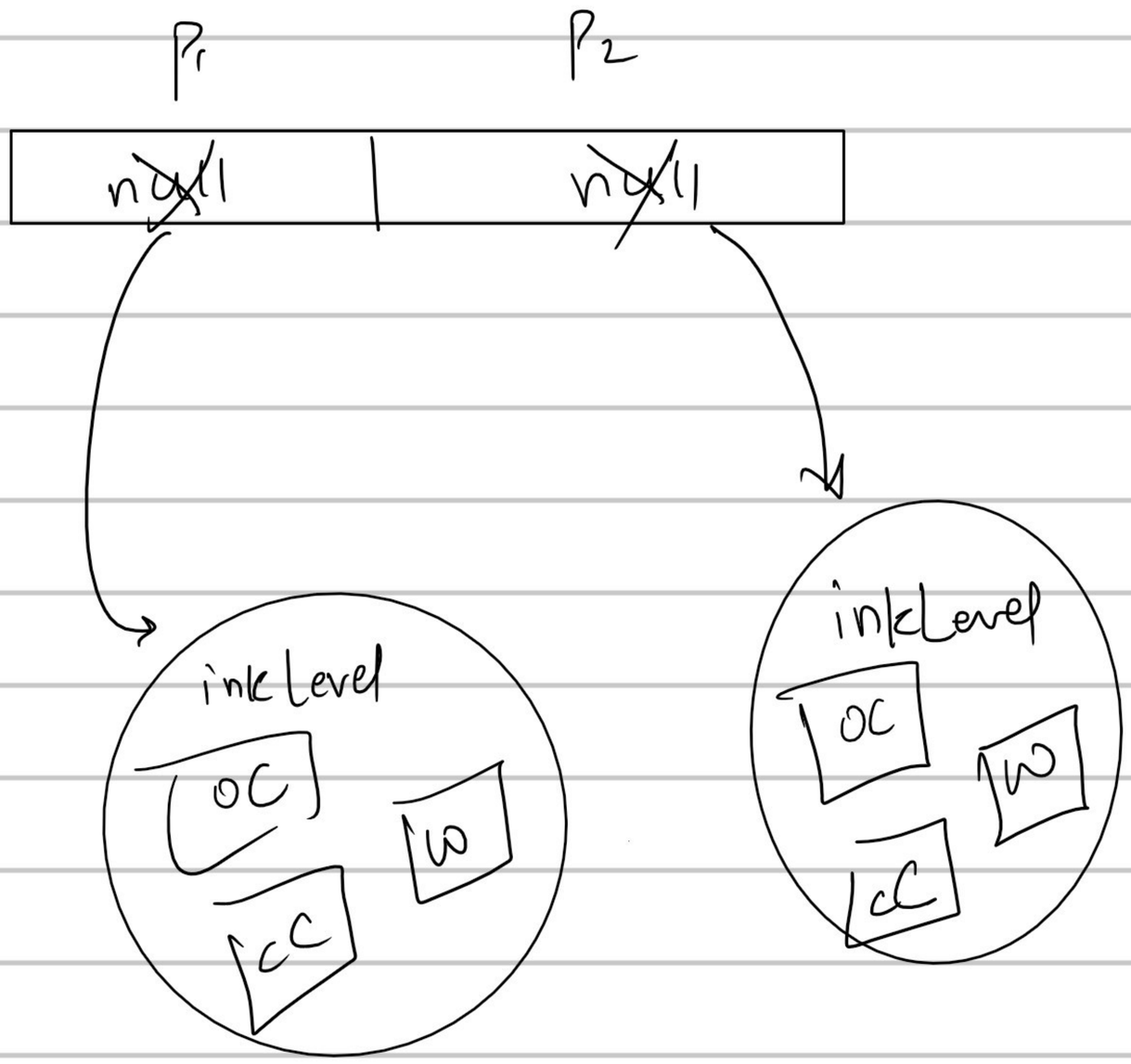
static

- modifier
- variable
- method
- block



* Anything which is of nature static

1. is loaded when the class is loaded.
2. have one copy for all the instances
3. can be accessed directly using the class-name

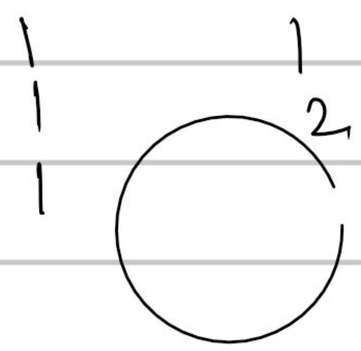


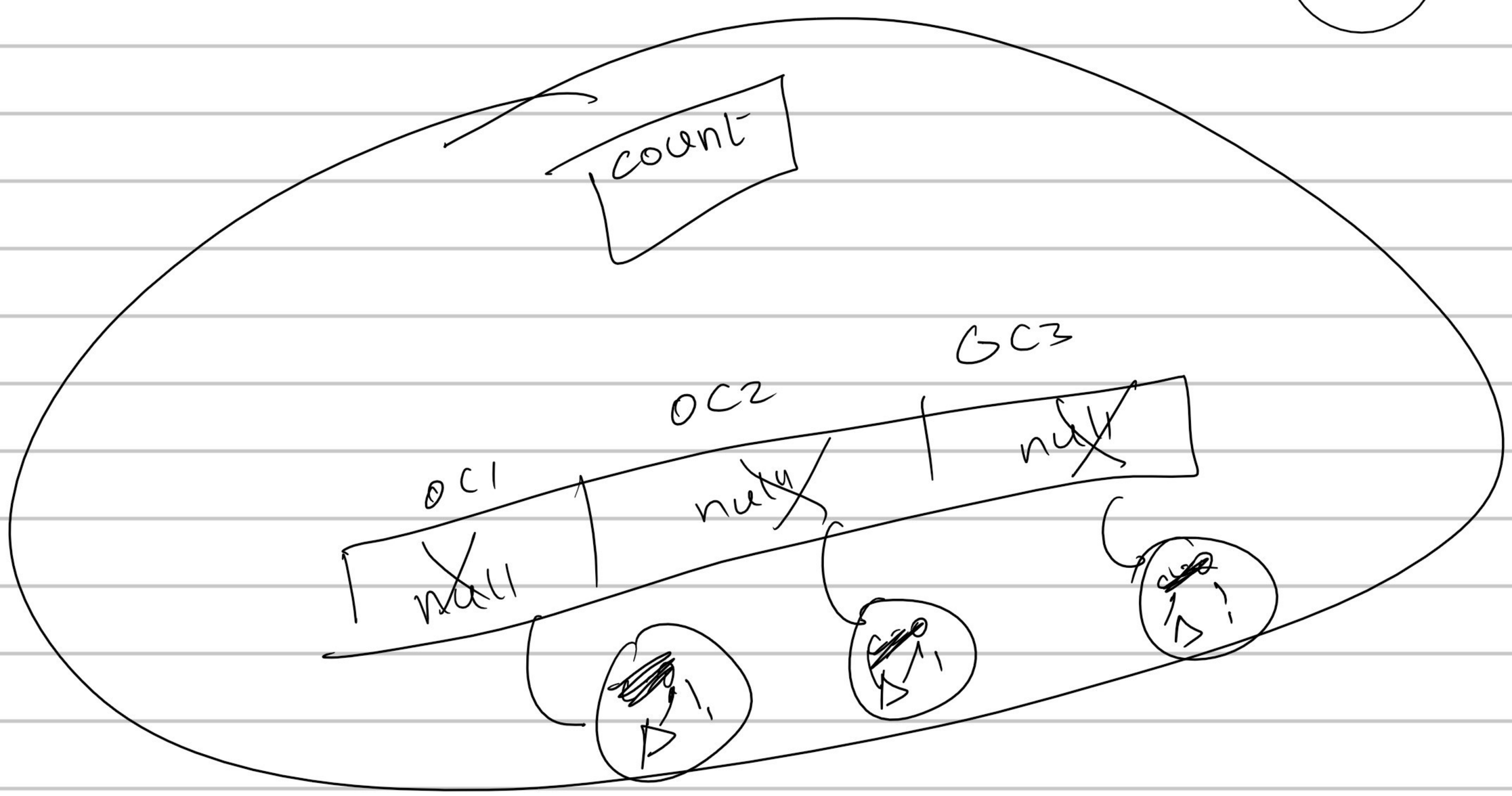

```
Pen  p1, p2;

p1 = new Pen();

p2 = new Pen();
```

```
class ObjectCounter {

static int count = 0;

ObjectCounter() {

    count ++;
}

}
```

```
ObjectCounter oc1, oc2, oc3;

oc1 = new ——— ();

oc2 = new ——— ()

oc3 = new ——— ();
```

? count



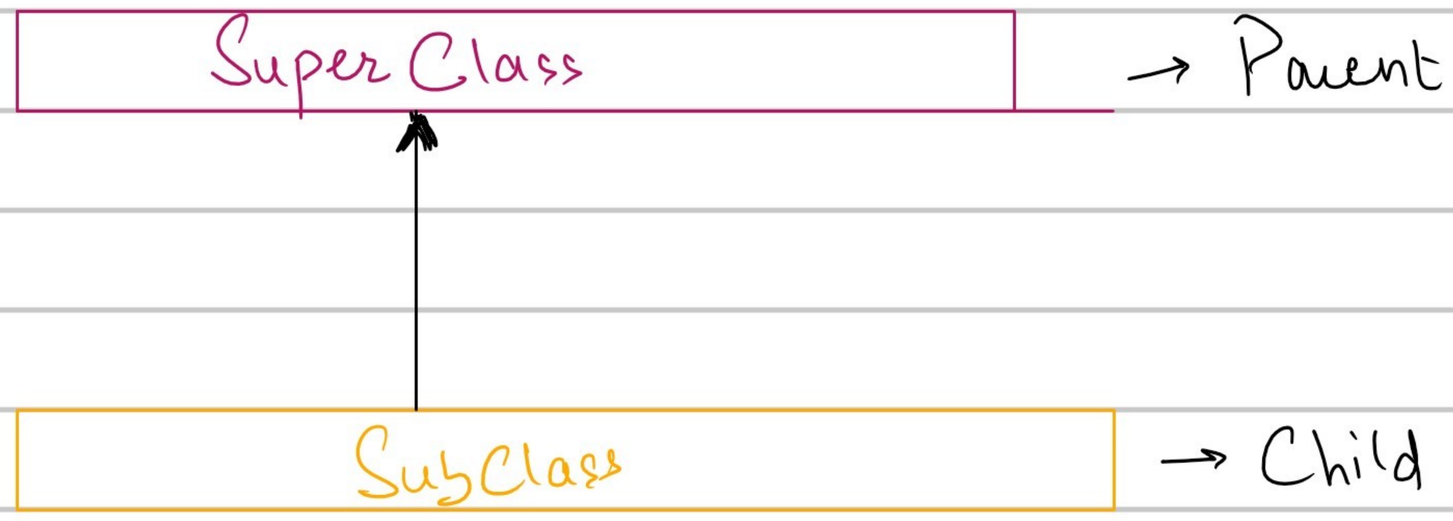


Inheritance

Process through which object of sub-child acquires the state & behavior of it's super class

Parent → Super Class

Child → Sub Class



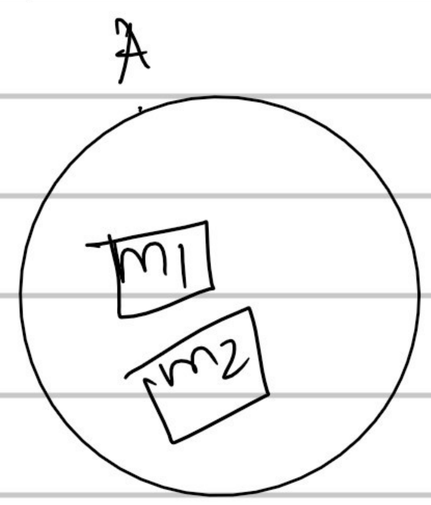
* How inheritance happens *

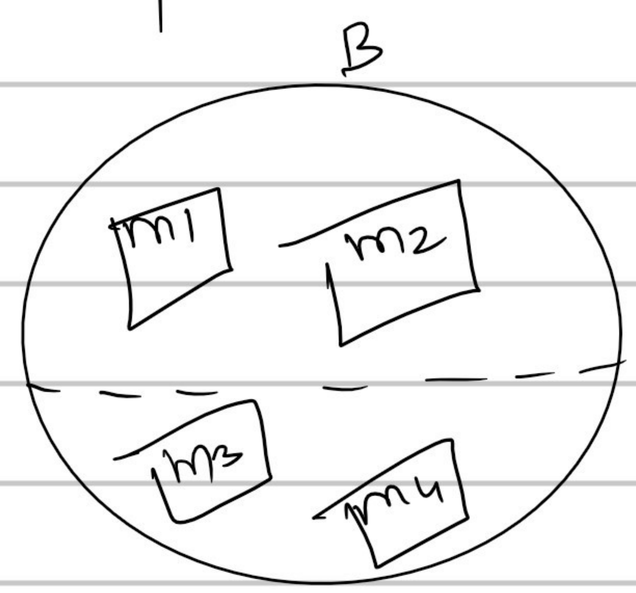
→ extends

```
class Parent {

}
```

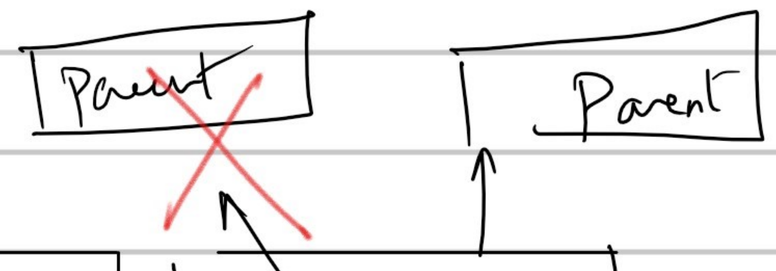
```
class Child extends Parent
{

}
```

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



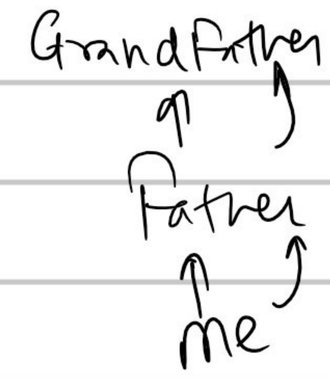


* Java doesnot support multiple inheritance.

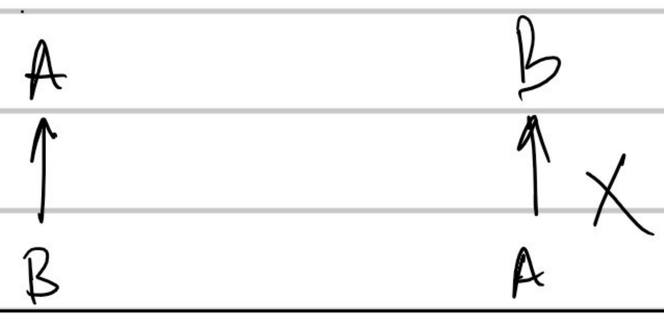


→ At any given point in time; a Java class will only have one direct super-class.

→ Java supports multi-level inheritance



→ Java do not support cyclic inheritance



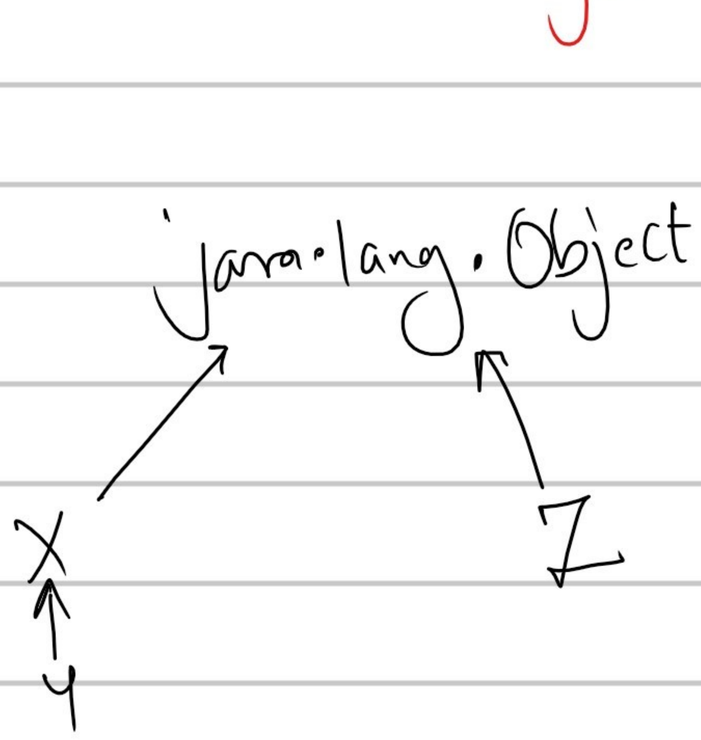
* A java class atleast & atmost have one super-class

```
class X {
}
```

```
class Y extends X {
}
```

```
class Z {
}
```

* Adam of Java *



java.lang.Object

→ ~~par~~

→ alpha-class

→ created one super-class for all; but sub-class for none.

→ provides us with methods;

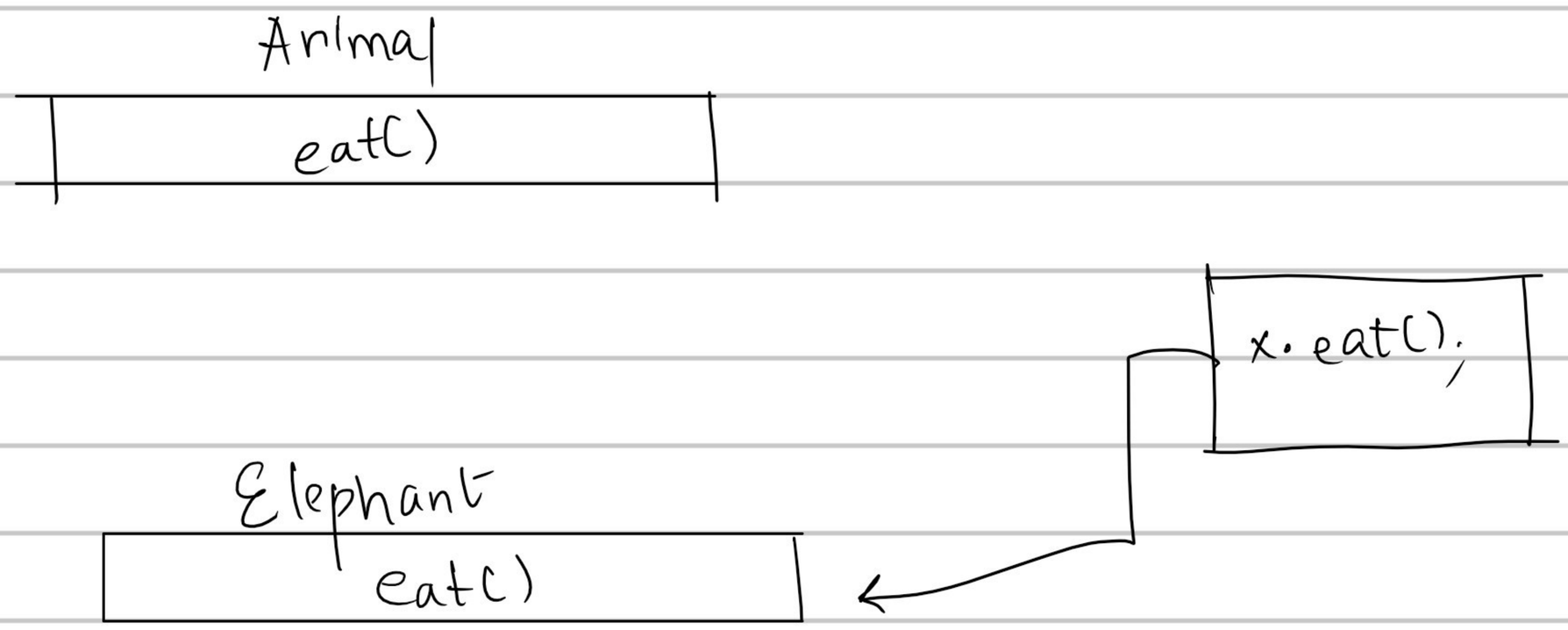
* Method Overriding *

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

name of method — parameters

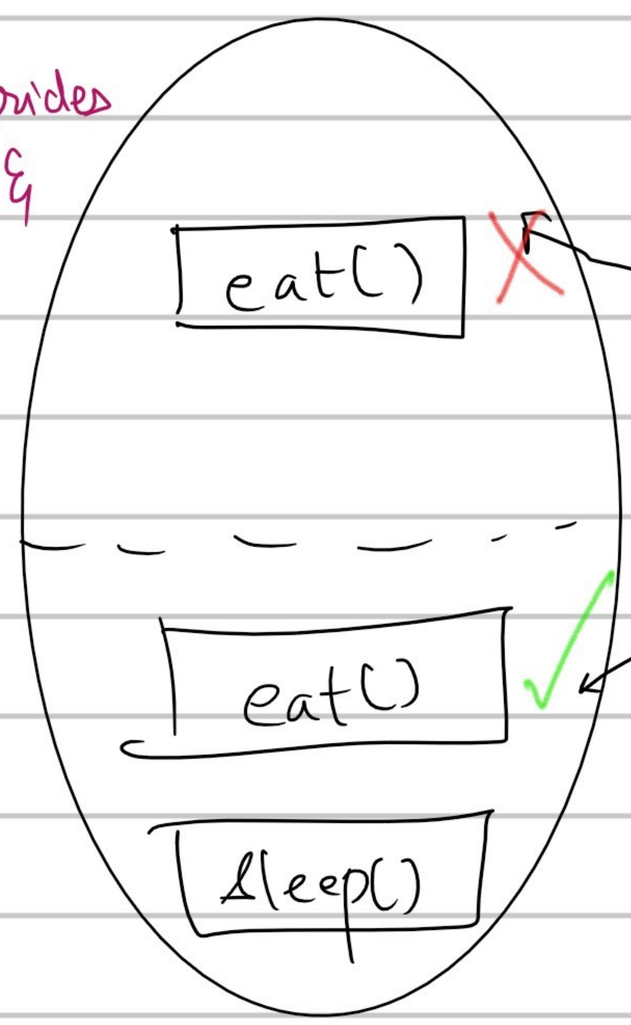
```
class Animal {
    public void eat() {
        ——— ("Eat → Animal");
    }
}
```

```
class Elephant extends Animal {
    public void sleep() {
    }
    public void eat() {
        ——— ("Eat of Elephant");
    }
}
```



* 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 it's sub-class instance.

```
Animal a = new Elephant();
```

Generalization

```
public void treat(Animal a ){
    a.eat();
}
```

someObj.treat(e);

When an object of super-class points at it sub-class instance, it access only overriden methods; other methods are sliced.

When super-class object pointing to sub-class instance invokes the overriden 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 overriden.

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

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

final class

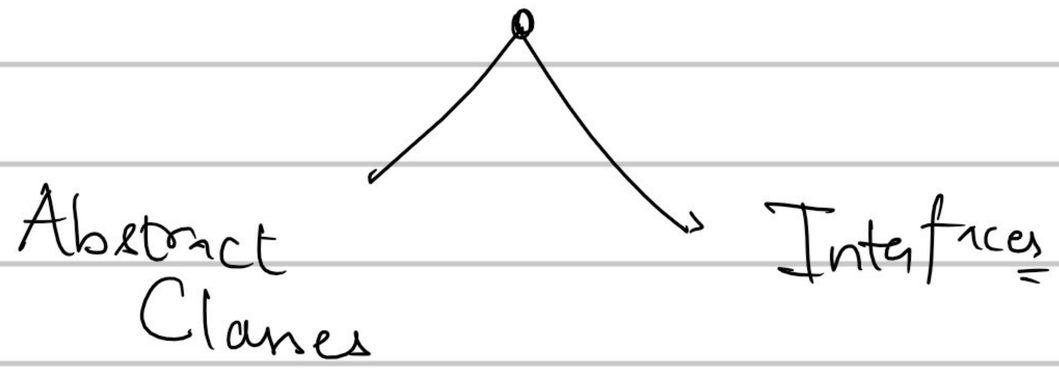
A class marked as final; cannot be sub-classed

```
final class Y {
}
```

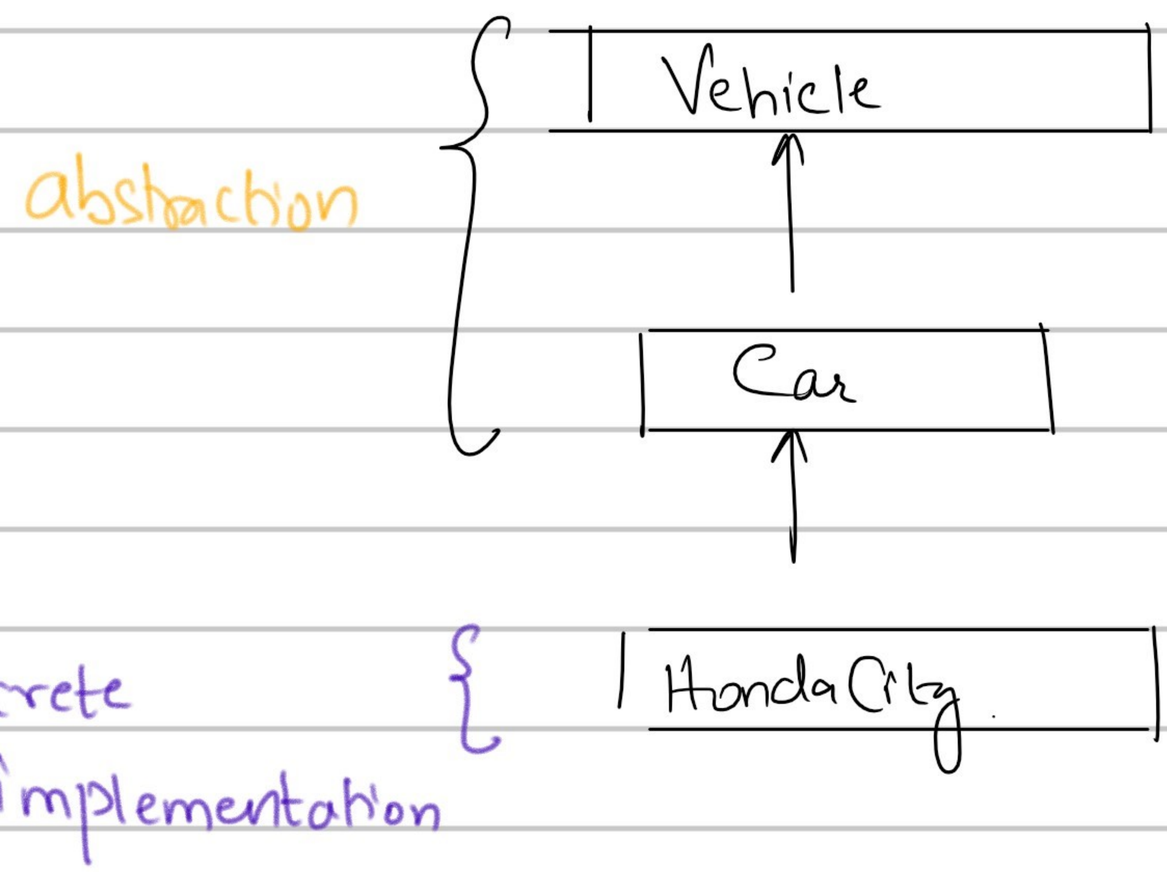
```
class Z extends Y {
}
```

* Deferred Implementation *

↳ declare now
implement later



* Abstract Class *



* Abstract class is used when you want ~~to~~ your sub-class 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();
public abstract void stop();
```

() { } → defining

() ; → declaring

```
abstract class Car {
public abstract void start();
public abstract void stop();
public void m1(){
}
}
```

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

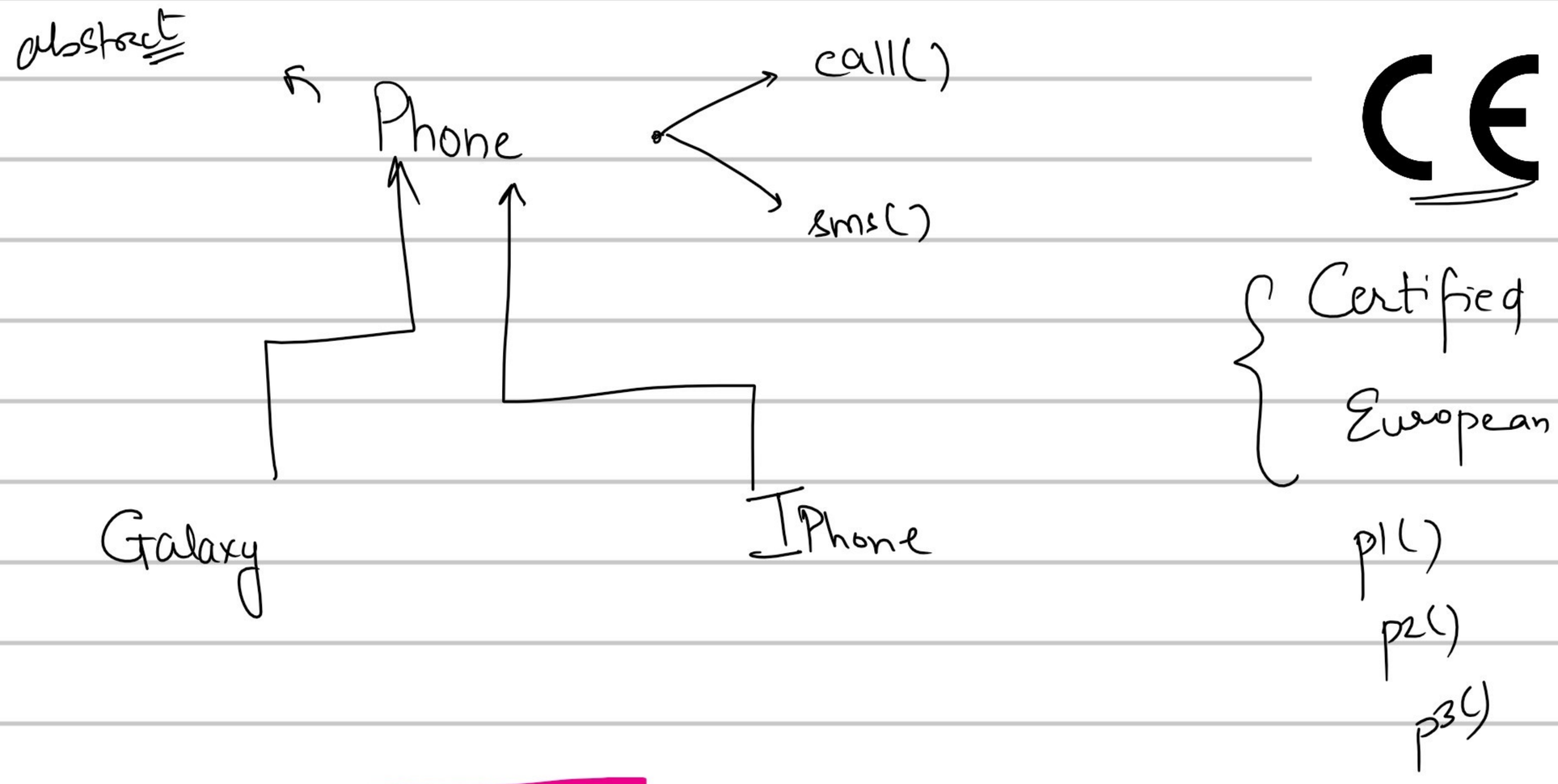
We can create an object of abstract class; but not instantiate it.

```
Car c;
c = new Car();
```

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()
}
```



* **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; use extends first & then implements

```
class IPhone extends Phone implements CE {

}
```

* A class can implement more than one interface.

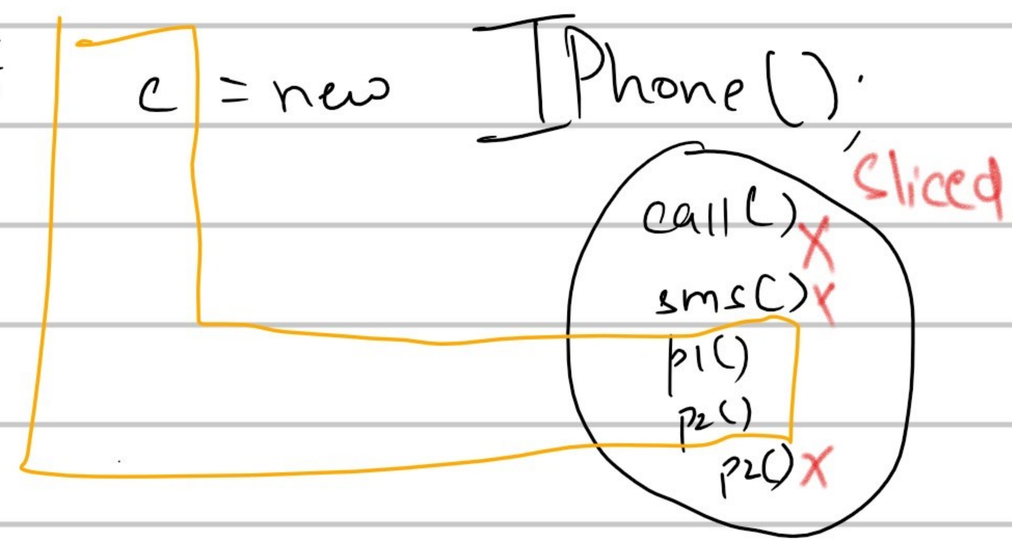
```
class IPhone extends Phone implements CE, ISO {

}
```

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

Generalization

```
public void sell (CE c) {

}
```

```
CE c = new IPhone();
```



Interface / Abstract Class	A class which implements / extends
* what to do	* How to do

---

* Packages * → collection of classes & interfaces in some directorial hierarchy

- in built packages
- creating our own packages.

In-built Java Library Packages

rt.jar (runtime.jarra archive)

auto import

- java
  - lang
    - String
    - System
    - String
    - Integer
    - Object
    - Class
  - util
    - Date
    - List
    - Map
  - io
  - sql
  - ...
- javax
- sun
- com

* 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
↓

Package → com.sc

↳ Creating segregation in project.

Layered Architecture

com.sc.rpg.loans

- beans → Customer.java
- ui → CustomerUI.java
- helper
- service → CustomerService.java
- Controller → CustomerController.java

* Access Specifier *

* public → available everywhere.
* private → only available in same class
* default → only available in same package
* protected → default + inheritance

if no access specifier is defined. uses default.

com.sc.b.a

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

class B {
}
```

com.sc.b.b

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

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

* 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

com.scb.a

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

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();

- Collection of characters.
- java.lang.String.
- String get's special treatment from JVM

String Constant Pool

String literal

String s="Tom";

String s=new String("Tom");

String Object

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

H	a	m	b	u	r	g	e	r
0	1	2	3	4	5	6	7	8
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 occuring at runtime breaking normal flow of the code.

Exceptions can be handled in two ways.

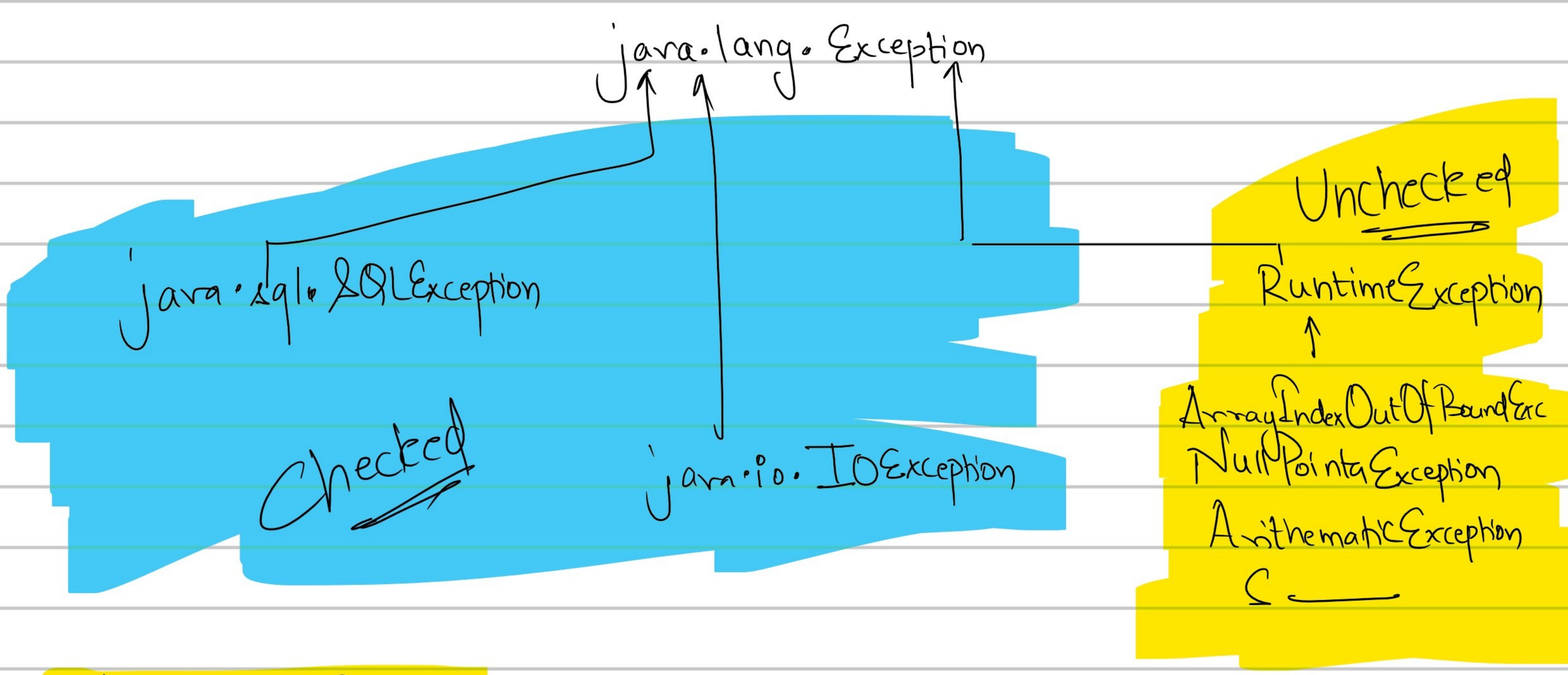
1. try... catch
2. declaring that exception

1. try... catch

```
try {

}
catch ( Exception e){

}
```

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

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();
```

```
try {

}
catch(Exception e){

}
```

```
try{
    db
    txt
}
catch(SQLException e){
}
catch(IOException e){
}
catch(Exception e){
}
```

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

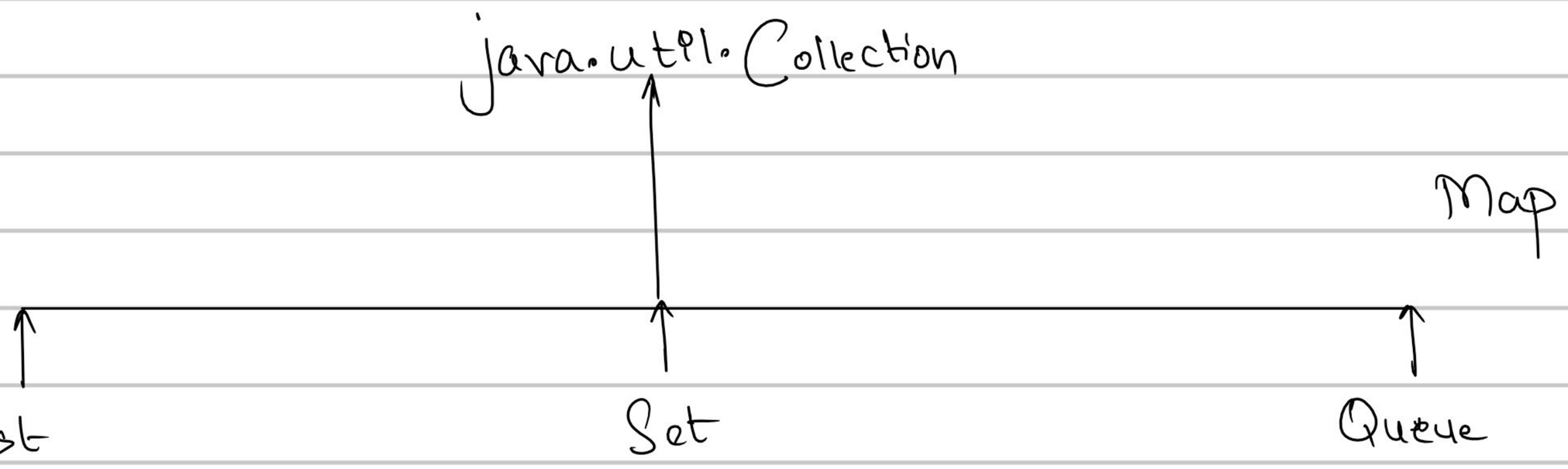
* Collection API *

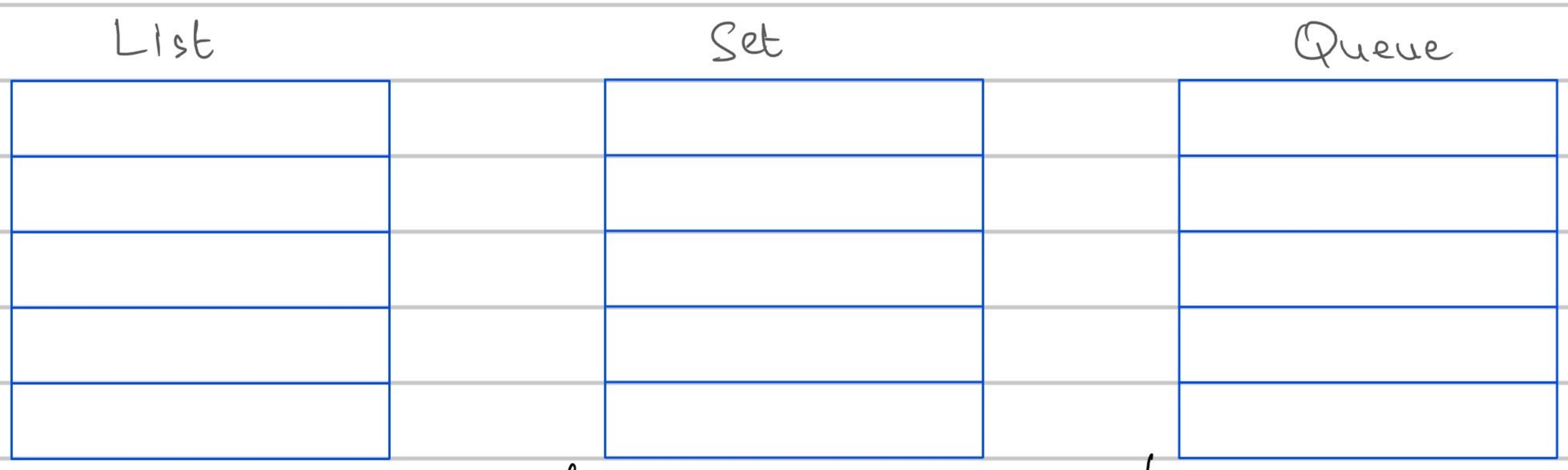
Array

- → fixed type
- → no convinience methods

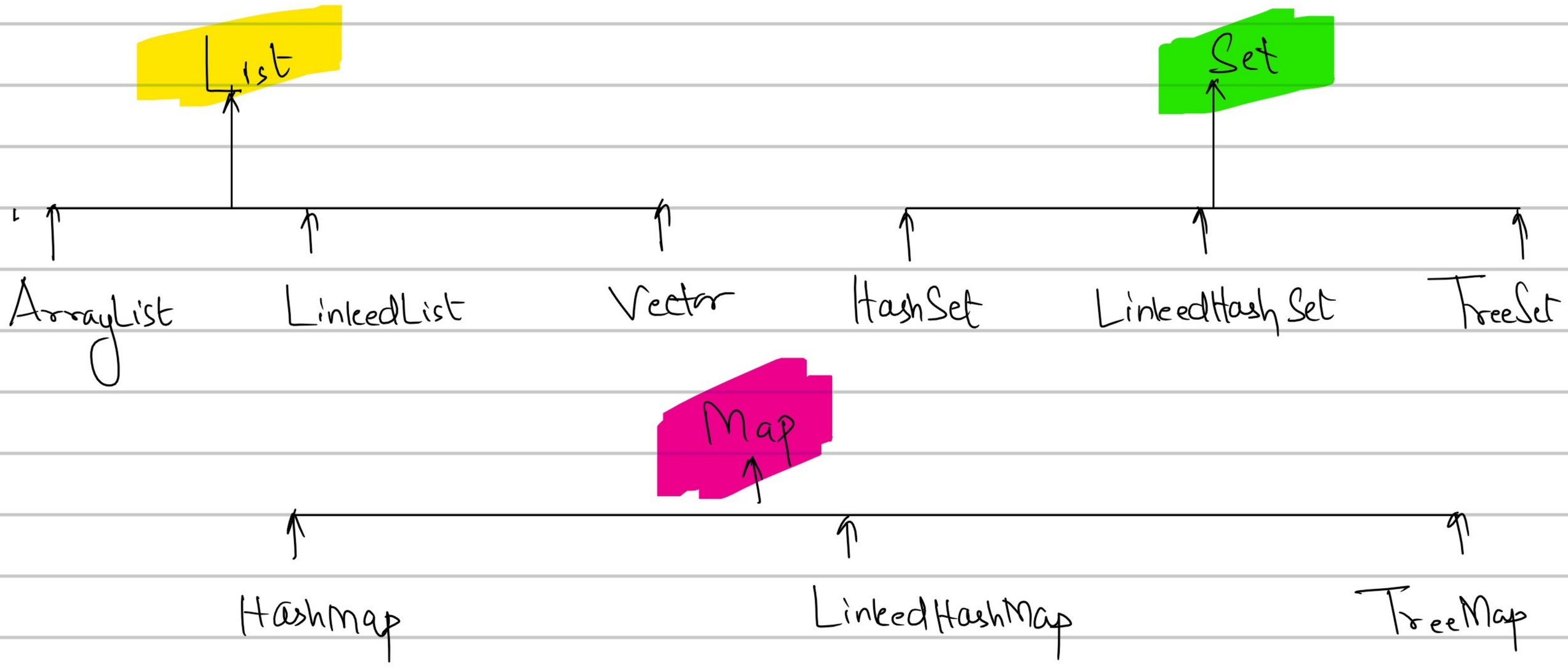
Collection API → way to store the data.

- → allows you to store hetrogeneous data
- → tons of convinience methods
- → java.util





- List → store duplicates, uses an array internally
- Set → does not allow duplicates
- Queue → FIFO



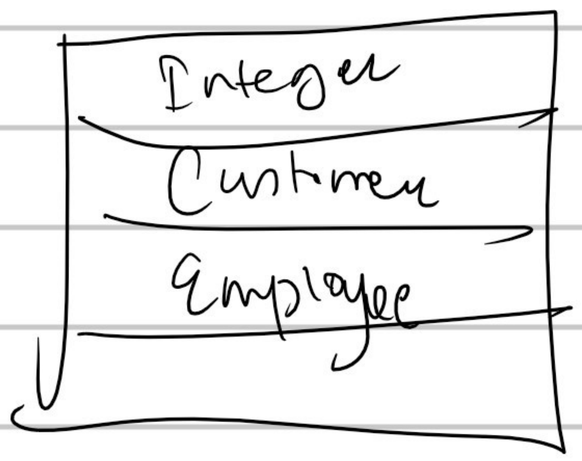
Hash → No order is maintained.
→ faster → hash algorithm

Linked → Order of insertion is maintained

Tree → Sorted by Natural key

```
ArrayList list = new ArrayList();
```

↳ can store hetrogeneous data



Generics

```
ArrayList<String> list = new ArrayList<String>();
ArrayList<int> list = ✗ ___;
```

Wrapper

	java.lang
byte	Byte
boolean	Boolean
char	Character
short	Short
int	Integer
long	Long
float	Float
double	Double

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

only hold integers

Boxing

i.intValue();

Unboxing

.class

int i = 10;

Integer i = new Integer(10);

Auto boxing

primitive

Box

Wrapper Class

* Fetching the data *

* for each loop

```
for ( Type element : collection) {

}
```

```
for ( String name : namesList) {

    S ——— (name);
}
```

Iterator → available for List, Set, map, Queue
↳ part of collection

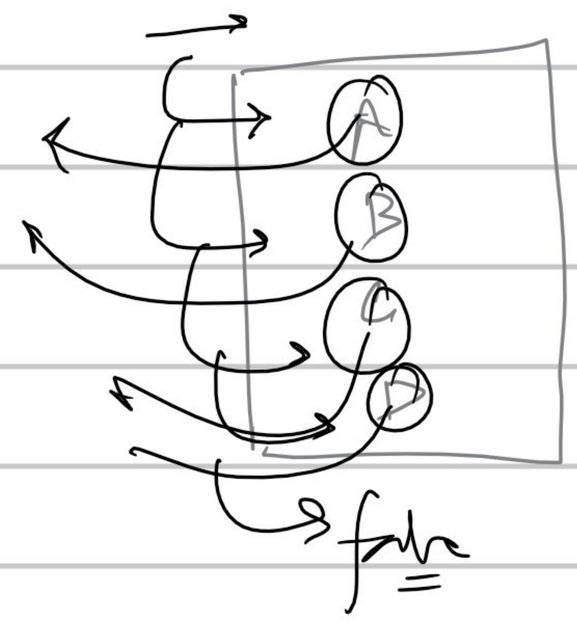
List namesList = ——

```
Iterator i = namesList.iterator();
```

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

i.next() → ref to the element

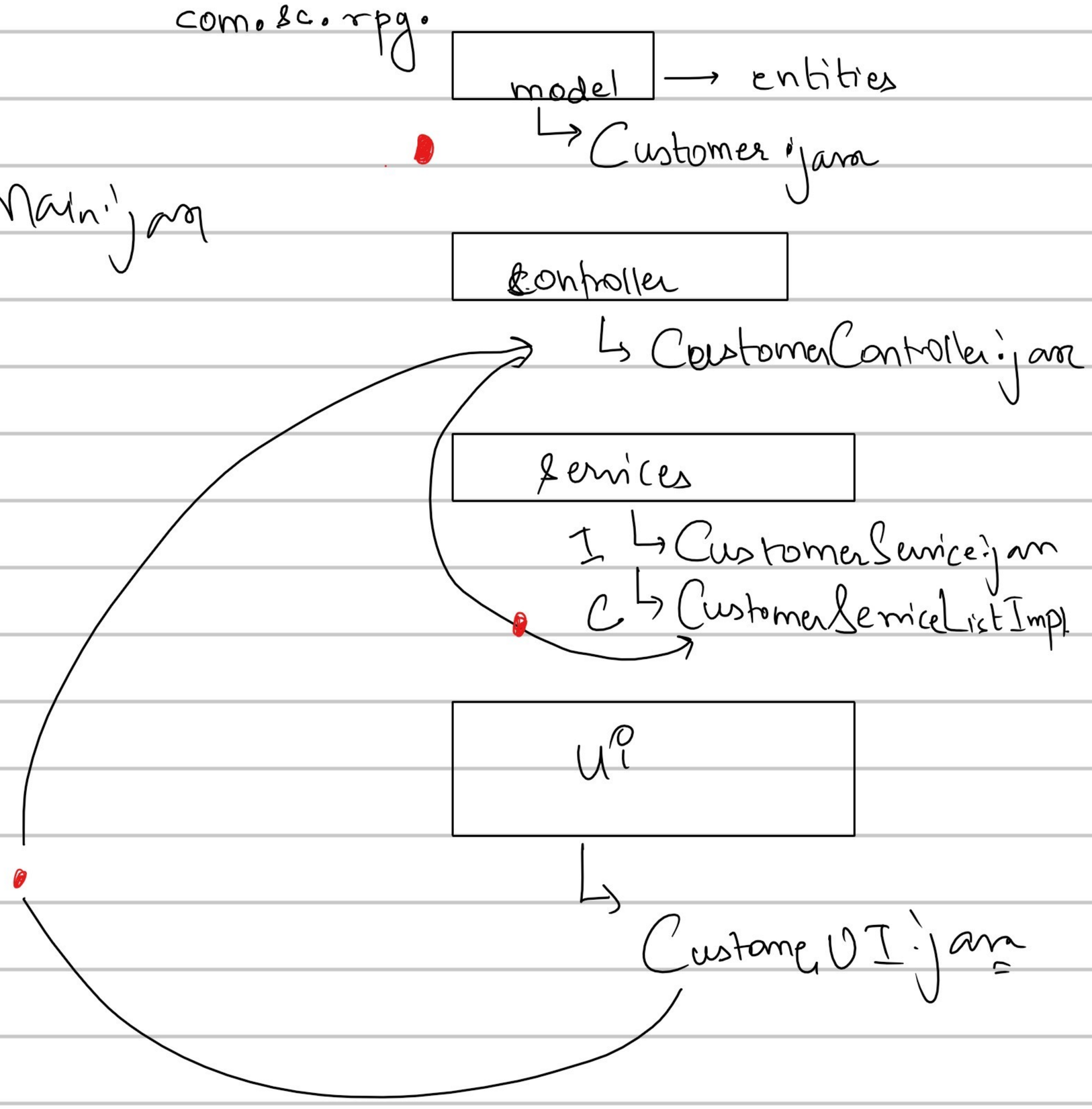
```
while ( i.hasNext() ) {

    String name = i.next();

}
```

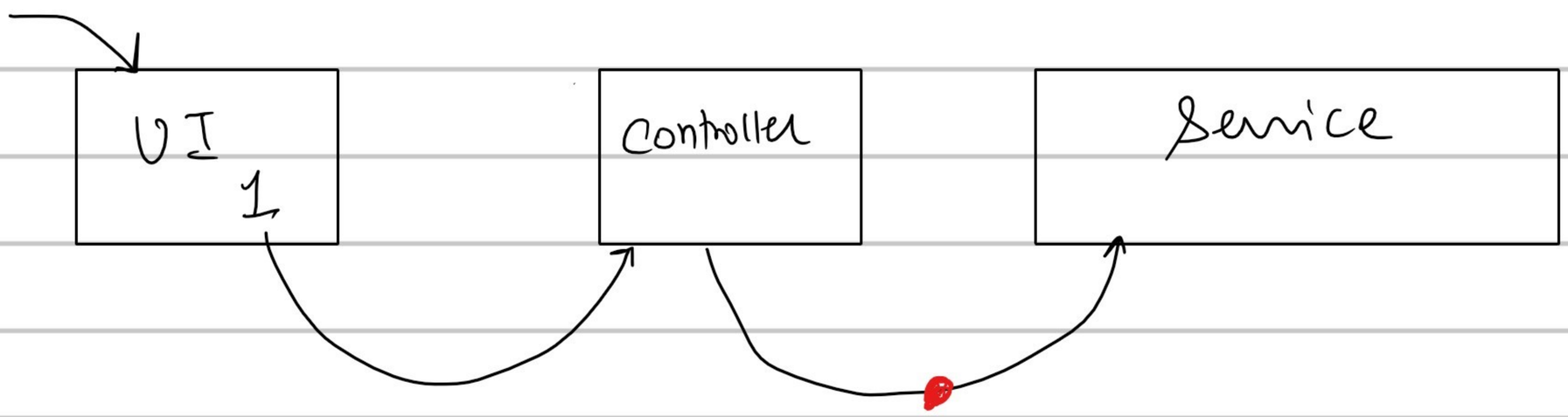


USE CASE

com.sc.rpg.

Rai Main.java

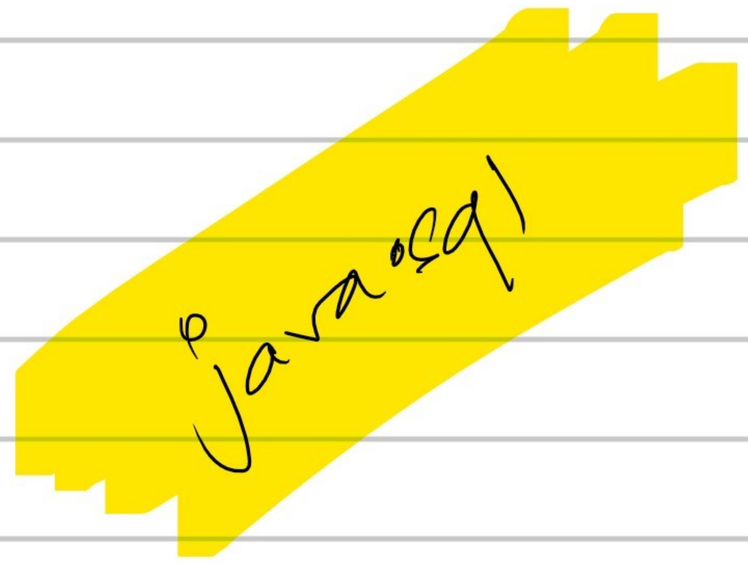




JDBC → Java Database Connectivity.

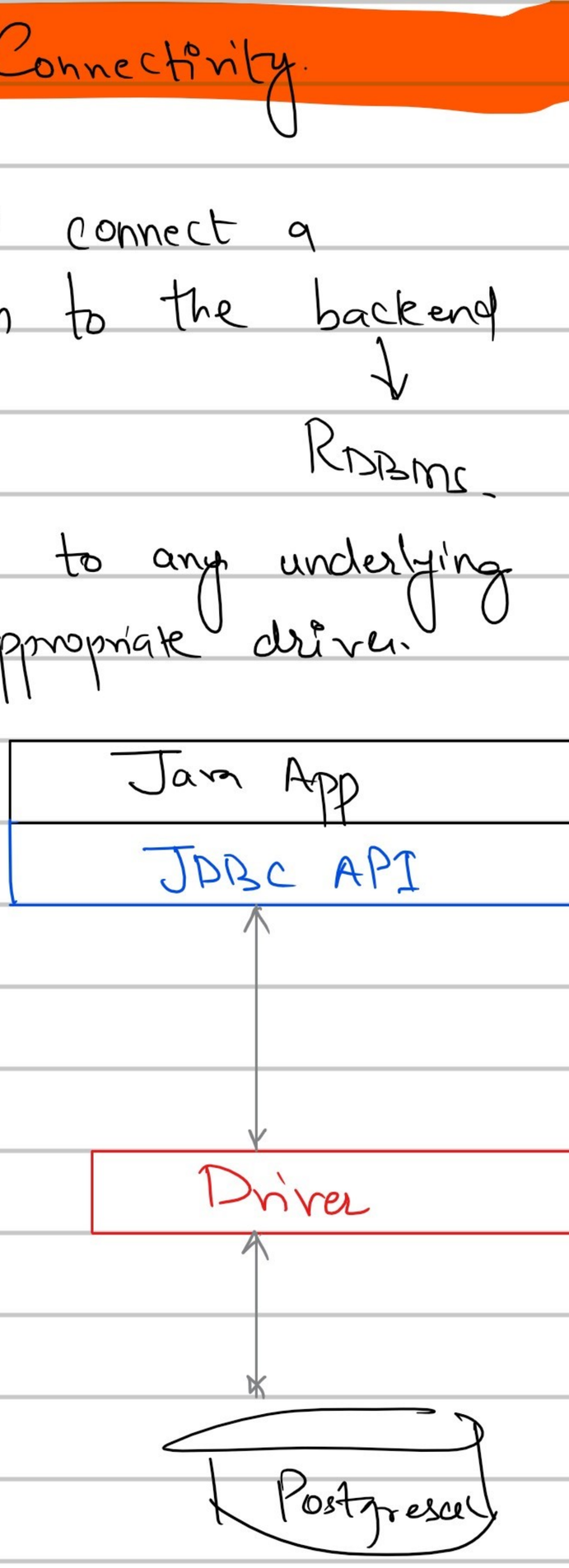
→ API which helps to connect a Java application to the backend → RDBMS.

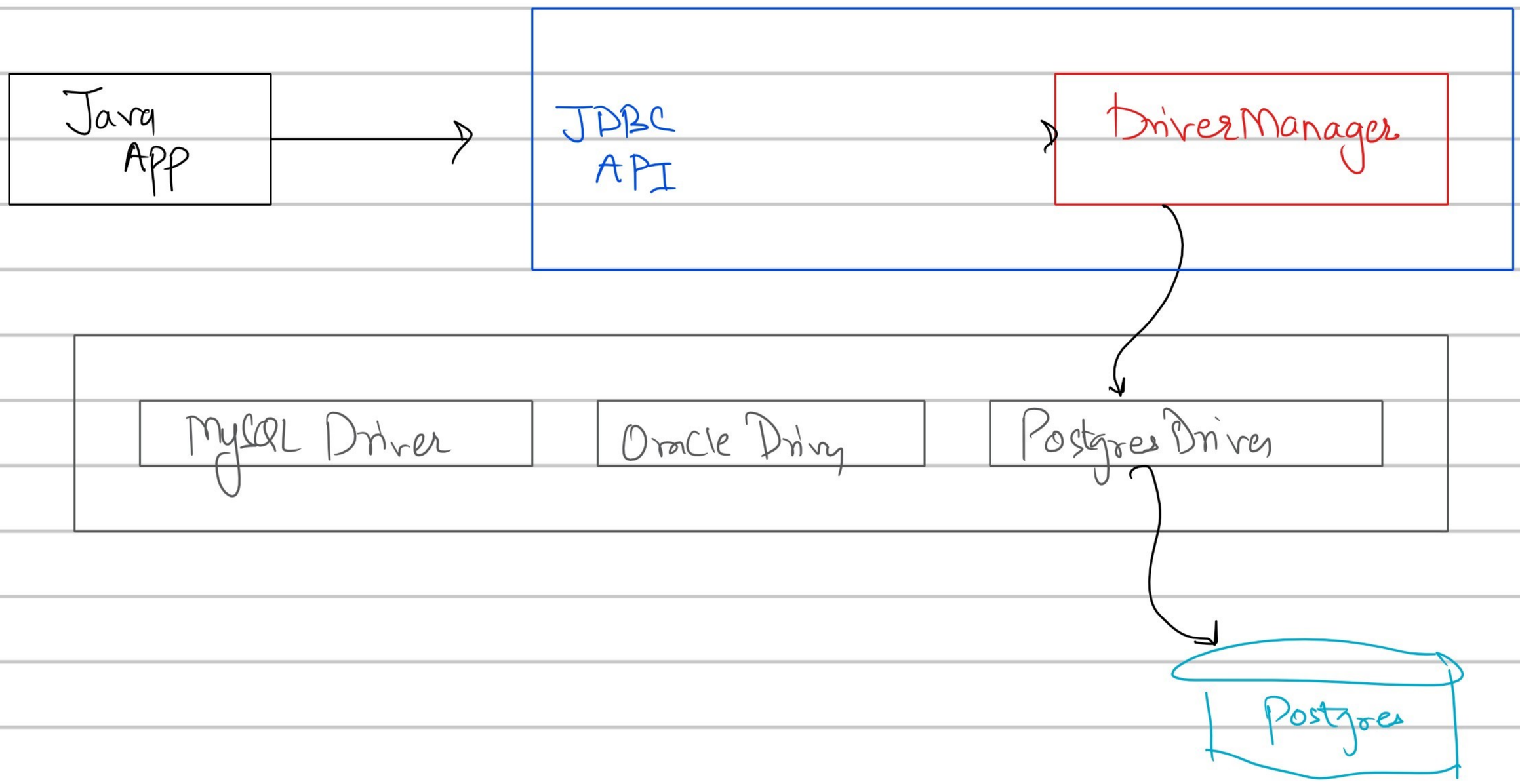
→ JDBC can connect to any underlying RDBMS using appropriate driver.



Components of JDBC API

- Driver
- DriverManager
- Connection
- Statement
- ResultSet
- ResultSetMetaData





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 *

rdbms specific

database specific

base-protocol : sub-protocol : sub-name

base-protocol → jdbc

sub-protocol → postgresql, mysql, oracle, sqlserver

sub-name → ip + port + name of db

```
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.

java.sql.Statement

java.sql.CallableStatement

java.sql.PreparedStatement

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 (?, ?, ?, ?) ";
1 2 3 4

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

DRL → Data Retrieval Language → SELECT

Type	Method	Return
DDL	pstmt.execute();	boolean false → successful true → unsuccessful
DML	pstmt.executeUpdate();	int ↳ number of rows affected
DRL	pstmt.executeQuery();	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)";
PreparedStatement pstmt = con.prepareStatement(query);
boolean status = pstmt.execute();
```

6.2 → DML

```
String query = "Insert into test values (?, ?)";
PreparedStatement pstmt = con.prepareStatement(query);
pstmt.setInt(1, 1000);
pstmt.setString(2, "Tom");

int rowAffected = pstmt.executeUpdate();
```

6.3 → DRL

customer

id	name	email	contact	accountty
1	Tom	t@gmail.com	12345	S
2	Alex	a@g—	12346	C
3	Mike	m@g—	12347	S
4	John	j@g—	12348	C

```
String query = "Select name, contact, email, account_type from customer";
PreparedStatement 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.

→ when next()/previous() is called, cursor moves accordingly

→ it checks if the pointed row has value or not

before-first

first

last

after last

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

Navigation

Cursor is always placed before first

```
rs.next();          rs.first();
rs.previous();      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");
}
```

name of col in db

Step 7: Close the Connection

```
pstmt.close();

con.close();
```

Layered Architecture

Separation of Concerns

com.sc.rpg.

models — Customer.java

ui — CustomerUI.java

controllers — CustomerController.java

service — CustomerService.java (I)

addCustomer()

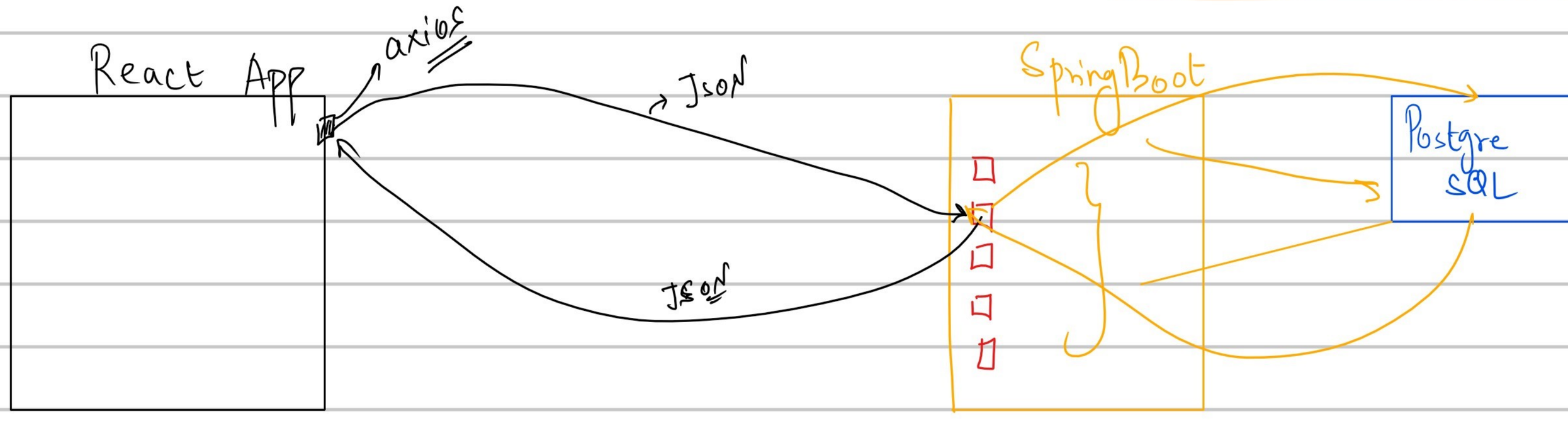
(C) - CustomerServiceListImpl

CustomerServicePBImpl

What to do

How to do

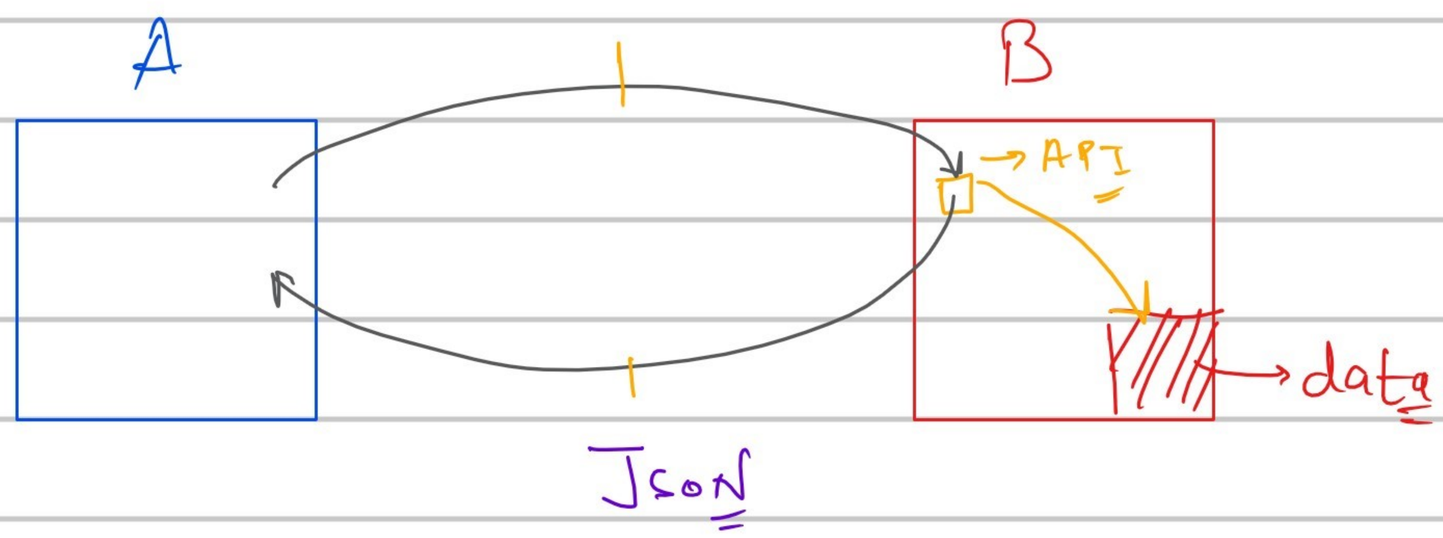
REST API



REST API

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

Representation
State
Transfer

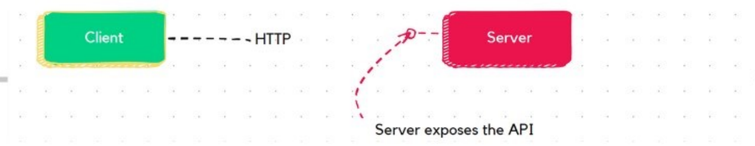


* Spring Boot
* Node JS
* Django
* ASP
* AWS serverless functions

* 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



http://vidly.com/api/customers

/ videos

protocol

URL

RESOURCE

HTTP METHODS / VERBS

customers

HTTP METHODS

GET

POST

PUT

DELETE

GET api/customers
api/customers/3

GETTING A CUSTOMER

Request

```
GET api/customers
```

Response

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

- 200
- 404
- 500

- Success
- Failure
- Server Failure

Idem'potent : ✓

Request

```
GET api/customers/1
```

Response

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

- 200
- 404
- 500

CREATING A CUSTOMER

Idem'potent : ✗

Request

```
POST api/customers
```

Request Body

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

Response

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

Create a new resource

- 200, 201
- 404
  409 (conflict)
- 500

UPDATING A CUSTOMER

Idempotent: ✓

Request

```
PUT api/customers/1
```

Request Body

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

Response

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

update the resource

- 200, 201, 204
- 404
- 500

DELETING A CUSTOMER

Idempotent : ✓

Request

DELETE api/customers/1

DELETE api/customers/1

Response

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

delete the resource with the given identifier

- 200
- 404
- 500

- 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 ✗

PUT /api/customers/4 x 100 → 1 ✓

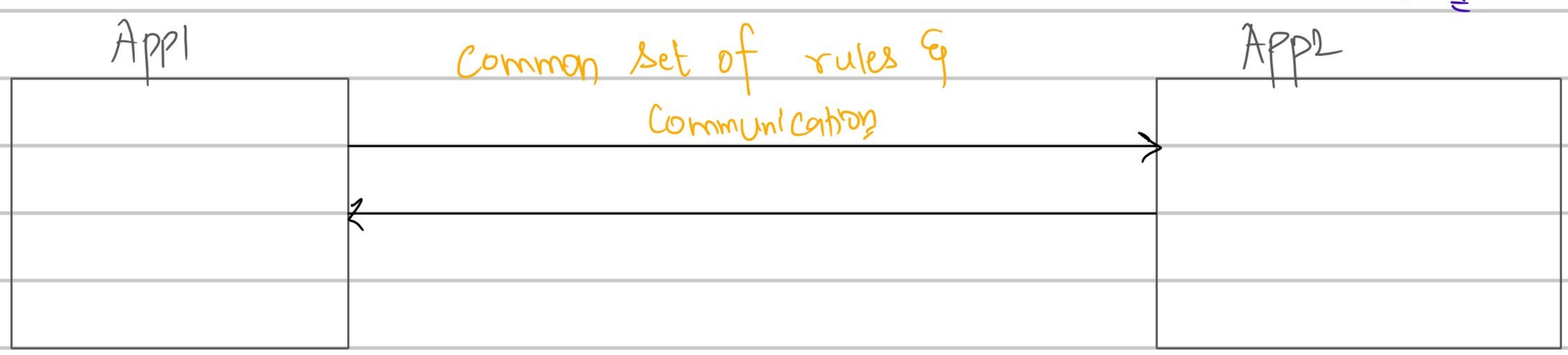
XML

eXtensible Markup Language

tags are user defined

XML uses < > </ >

storing & transporting data




```
[ <employee-list> ]  → root
    <employee>
        <id> 101 </id>
        <name> Tom </name>
    </employee>
    <employee>
        <id> 102 </id>
        <name> Alex </name>
    </employee>
[ </employee-list> ]
```

parent

child

siblings

Child

* Maven *

→ build tool

Yiddish Language → "accumulator of knowledge"

Java Based Application → building, managing

Objectives

1. Making the build process easy.
2. Provides a uniform build system
3. Provides quality project information
4. Encourages better development practices

Maven

- Structure to your project
- Build
- Repository Management

A → maven

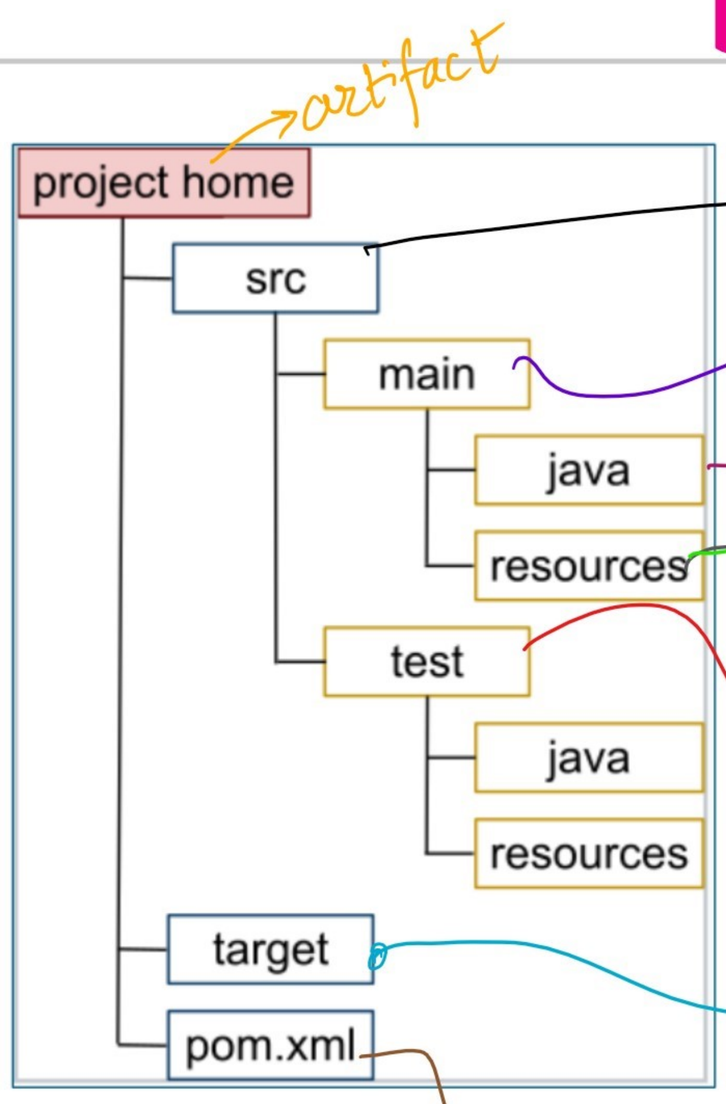
B → maven

C → maven

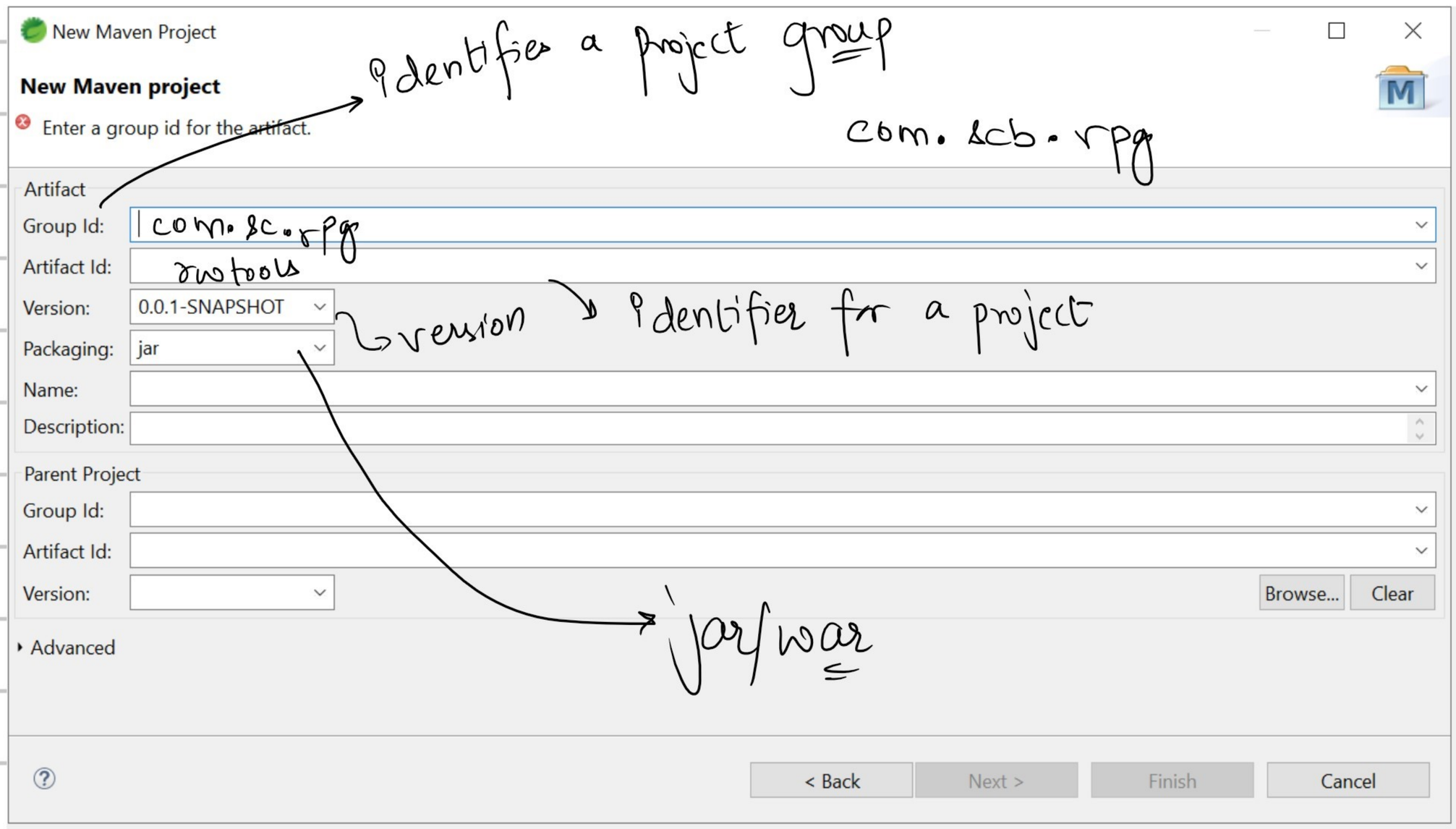
archtype → Architecture Type

Maven provides a uniform structure rather than developers using their own structure

Basic Maven Java Project



- src → source folder
- main → main java files goes here.
- java → java packages & then java files
- resources → image, pdf, property.
- test → all the test related jam files & resource goes here
- target → projects → build → jar / war
- pom.xml → project object model { responsible for bringing in dependencies & managing information }



Maven & Dependency Management

Non-maven

PostgreSQL website → jar

postgresql.jar

faker website → jar

faker.jar

2 Problems with this approach

① Every project needs their own library

② Visit multiple official websites & keep track

Centralized Local Repositry.

Maven Project 1

Maven Project 2

Maven Project 3

Maven Project 4

.m2

repositry.

org

Postgres

jar

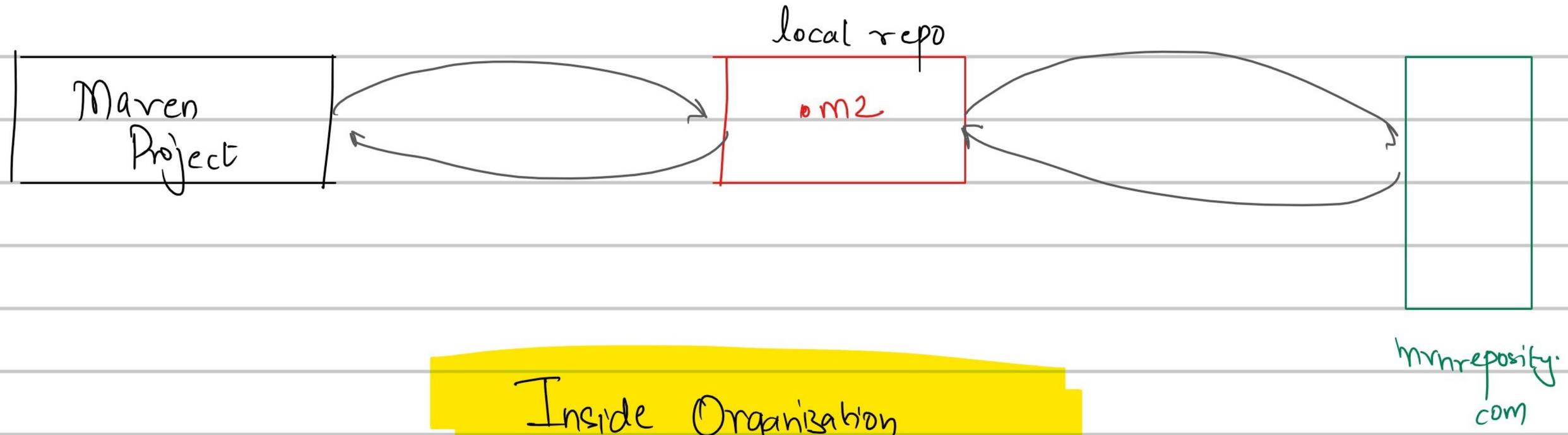
if the dependency we have is available in local .m2 repositry, use it else take it from central repositry

Online Central Repositry

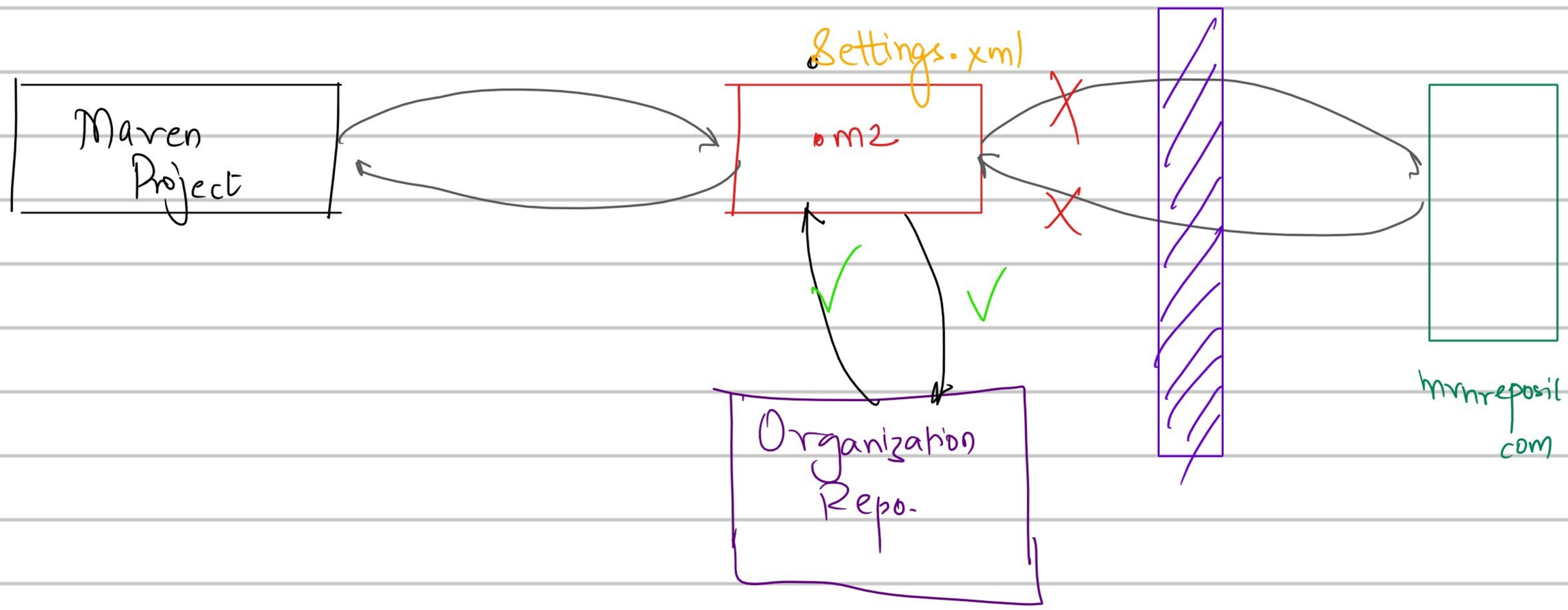
mvnrepositry.com

mavendemo/pom.xml JDBCBoilerPlate.java JDBCBoilerPlate.java

```
1 <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="h
2   <modelVersion>4.0.0</modelVersion>
3   <groupId>com.sc.rpg</groupId>
4   <artifactId>mavendemo</artifactId>
5   <version>0.0.1-SNAPSHOT</version>
6
7   <dependencies>
8
9   <!-- https://mvnrepository.com/artifact/org.postgresql/postgresql -->
10 <dependency>
11     <groupId>org.postgresql</groupId>
12     <artifactId>postgresql</artifactId>
13     <version>42.4.0</version>
14 </dependency>
15
16
17   </dependencies>
18
19 </project>
```



Inside Organization



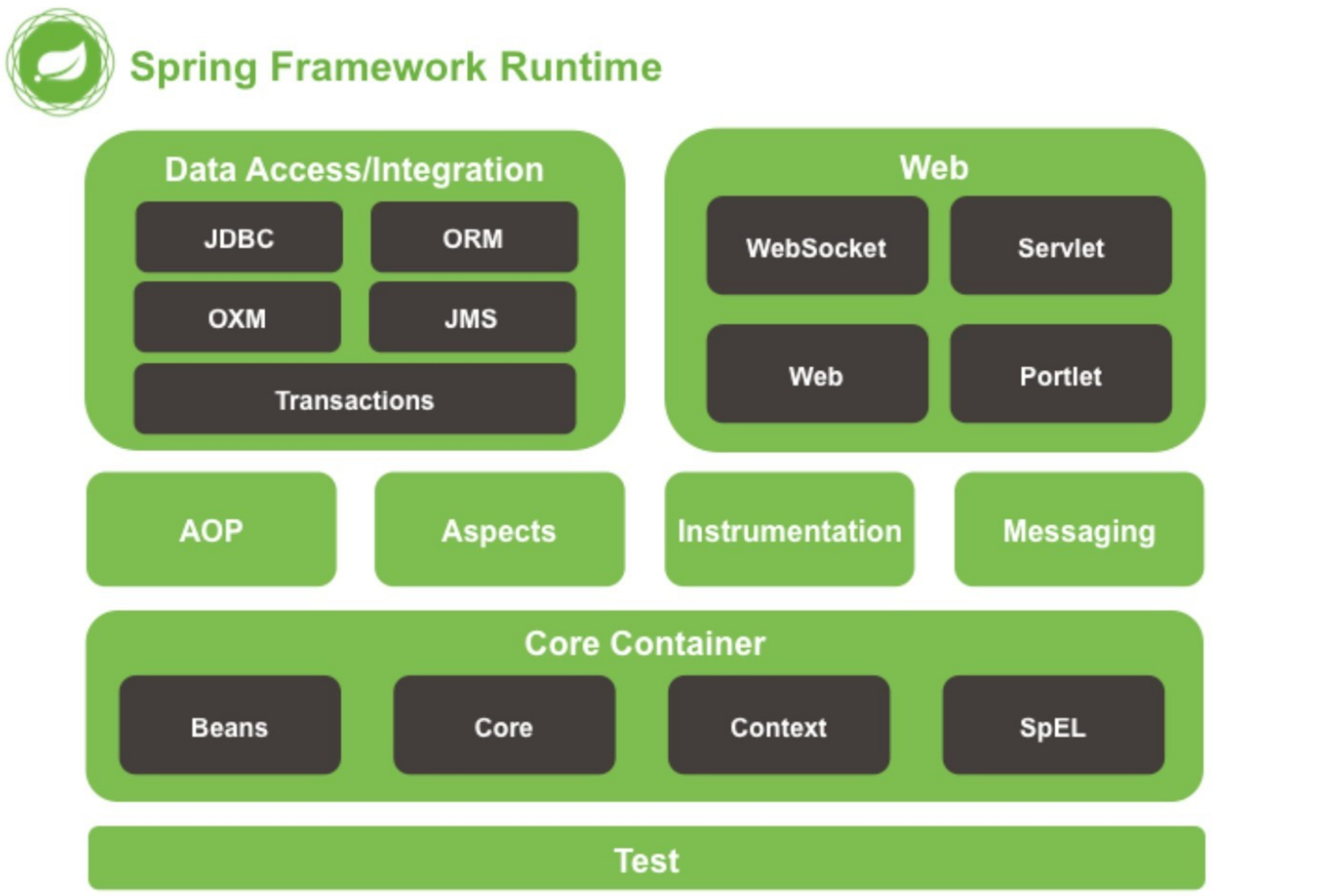
Spring Boot

→ SpringBoot is a successor of Spring Framework.

→ SpringBoot under the hood uses Spring Framework

Spring Boot
↓
Spring Framework

Spring Framework → framework of frameworks



lot of configuration was required

Spring Boot → makes it easy to create stand-alone, production-grade Spring based application that you can "Just Run".

Do you need to know Spring Framework for understanding Spring Boot ?? → No!!

* Features of Spring Boot

① Auto - Configuration

② Stand - Alone

③ Opinionated

① Auto Configuration → Developers can auto-configure a whole app using annotations & dependencies

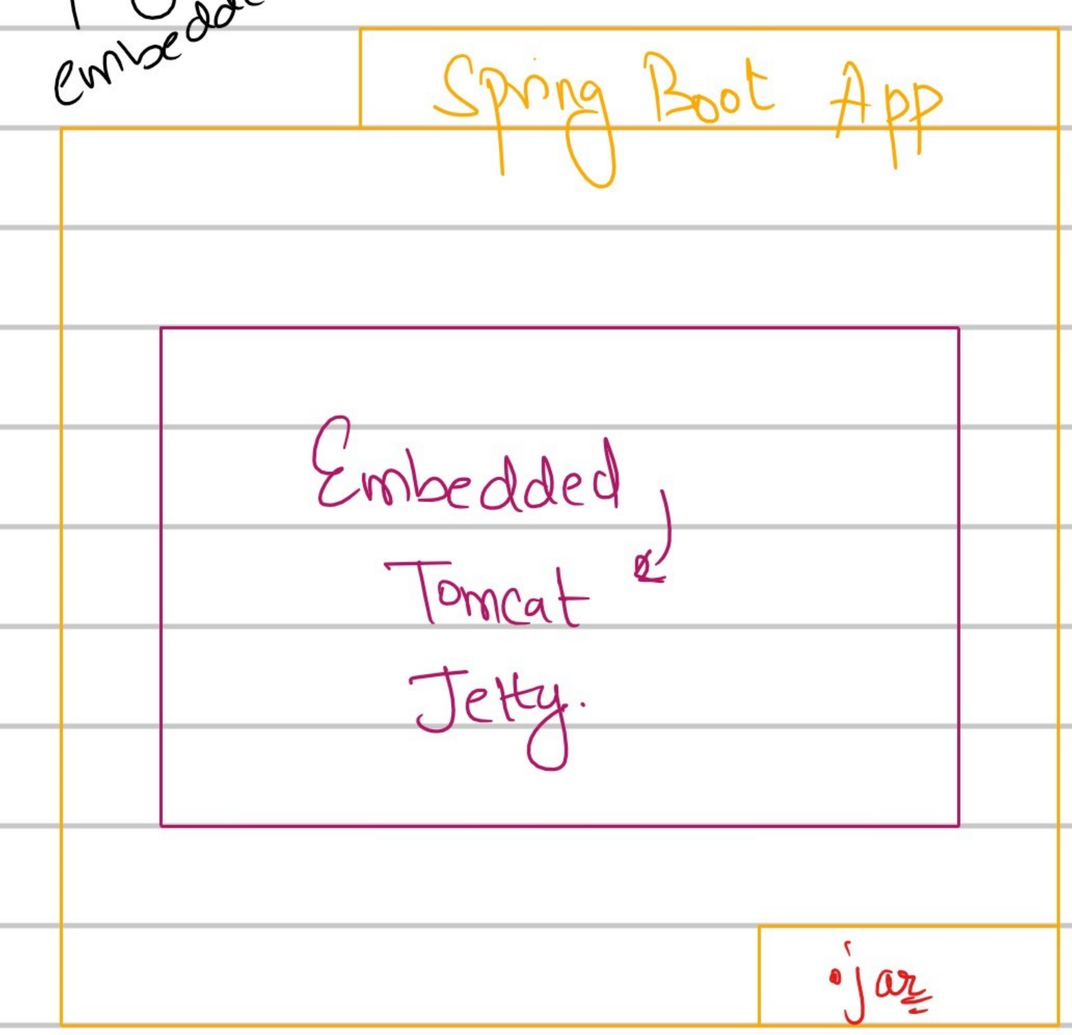
→ Even with auto-configuration; Spring Boot allows to make the customizations

② Stand - Alone

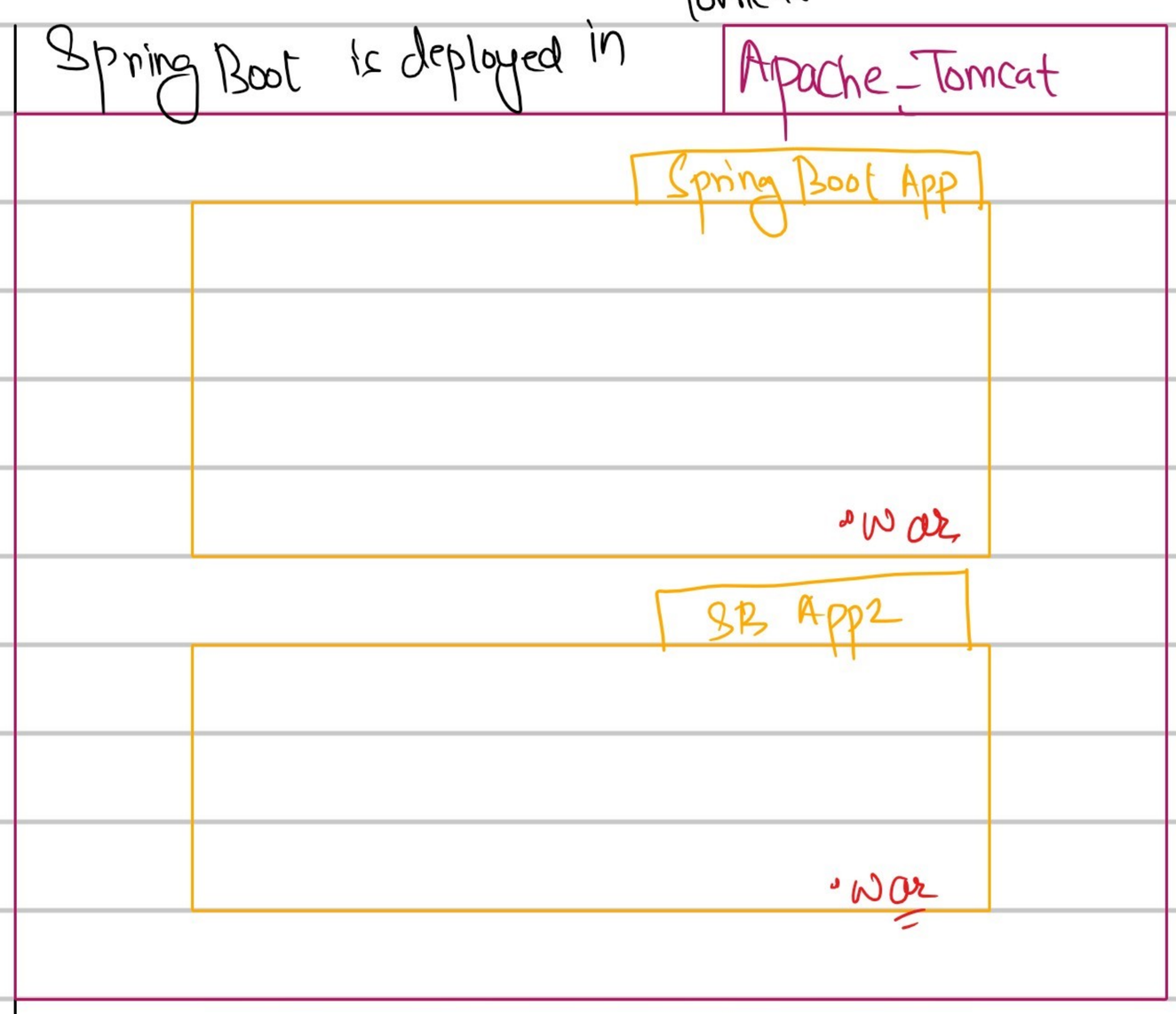
→ Spring Boot Application does not need a web-server to be deployed. It has it's own embedded web-server.

→ But on other hand; Spring Boot does have an option to export it's application in such a way that it can be deployed on an existing Web-Server

Spring Boot have embedded web-servers



Spring Boot is deployed in Tomcat



③ Opinionated

Spring Boot decides what defaults to use for configuration

- It also decides what packeages to be installed for a specific dependency.

Food Cort

French fries + Ketchup

* Annotations *

auto-configuration

Annotation is meta-data
↳ data about data

@

```
@Override
public String toString() {

}
```

tell the compiler that this particular method is overriden method.

→ If there is not such method in super-class, this annotation will throw a compile time error.

* How will annotations be used in Spring Boot ? *

MyController

Earlier in Spring Framework

configuration.xml

{ < >

</xml>

In SpringBoot

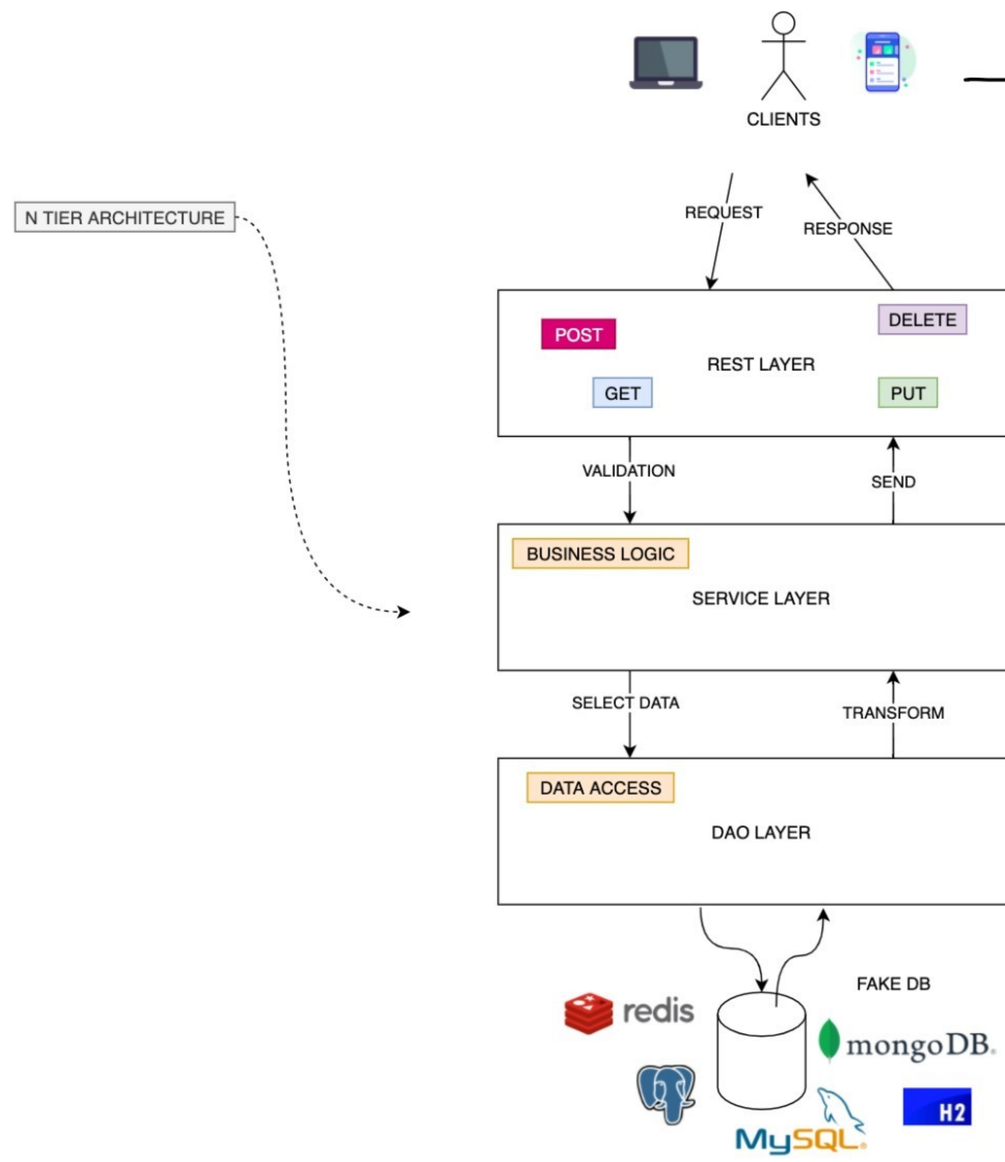
@RestController

MyController

annotation specifies that this class is a Controller

* Spring Boot offers a fast way to build applications. It looks at your classpath & at the beans you have configure ; makes reasonable assumption about what is missing & add those items *

With SpringBoot you can focus more on business logic & less on infrastructure of code



→ Front end → postman
react

→ Controller layer

Spring Boot Project Structure.

- springplayground [boot] [devtools]
  - src/main/java
    - com.scb.rpg.springplayground
      - SpringplaygroundApplication.java
  - src/main/resources
    - static → images
    - templates → html
    - application.properties
  - src/test/java
    - com.scb.rpg.springplayground
      - SpringplaygroundApplicationTests.java
  - JRE System Library [JavaSE-1.8]
  - Maven Dependencies
  - src
  - target
  - HELP.md
  - mvnw
  - mvnw.cmd
  - pom.xml

All the packages & java files go here.

Java file → main() → Runs the application

Property file → port, db connection, s.e.l

Test Cases goes here.

pom.xml → dependencies & build configuration

18. Using the @SpringBootApplication Annotation

all the classes & interfaces should be inside the base package

Many Spring Boot developers like their apps to use auto-configuration, component scan and be able to define extra configuration on their "application class". A single `@SpringBootApplication` annotation can be used to enable those three features, that is:

- `@EnableAutoConfiguration`: enable Spring Boot's auto-configuration mechanism
- `@ComponentScan`: enable `@Component` scan on the package where the application is located (see the best practices)
- `@Configuration`: allow to register extra beans in the context or import additional configuration classes

com.scb.rpg

The `@SpringBootApplication` annotation is equivalent to using `@Configuration`, `@EnableAutoConfiguration`, and `@ComponentScan` with their default attributes, as shown in the following example:

```
package com.example.myapplication;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication // same as @Configuration @EnableAutoConfiguration @ComponentScan
public class Application {

        public static void main(String[] args) {
                SpringApplication.run(Application.class, args);
        }

}
```

① Start Tomcat Server

② Run the Application

dao → data access object

Starting package : **com.scb.<classname>.teamname**

example com.scb.lambda.loan

Controller	com.scb.<classname>.teamname.controller
Service	com.scb.<classname>.teamname.service
Service implementation	com.scb.<classname>.teamname.service.impl
Data	com.scb.<classname>.teamname.dao
Data implementation	com.scb.<classname>.teamname.dao.Impl
entities /model	com.scb.<classname>.teamname.entities
Exception	com.scb.<classname>.teamname.exception
Common utils	com.scb.<classname>.teamname.utils

all controllers

interface for service

actual service

interface for dao

actual implementation Java

model/bean

exception

utilities

Whatever component (controllers/services/repositories) we write; it has to be bare-minimum marked as @Component

@Component → specify that class is Spring Bean

@Controller

@Service

@Repository

@RestController

Doctor

NeuroSurgeon

Cardiologist

Dentist

org.springframework.stereotype

Annotation Type Component

```
@Target(value=TYPE)
 @Retention(value=RUNTIME)
 @Documented
 @Indexed
public @interface Component
```

Indicates that an annotated class is a "component". Such classes are considered as candidates for auto-detection when using annotation-based configuration and classpath scanning.

Other class-level annotations may be considered as identifying a component as well, typically a special kind of component: e.g. the `@Repository` annotation or AspectJ's `@Aspect` annotation.

Since:

2.5

org.springframework.stereotype

Annotation Type Controller

Controller

```
@Target(value=TYPE)
 @Retention(value=RUNTIME)
 @Documented
 @Component
public @interface Controller
```

Indicates that an annotated class is a "Controller" (e.g. a web controller).

This annotation serves as a specialization of `@Component`, allowing for implementation classes to be autodetected through classpath scanning. It is typically used in combination with annotated handler methods based on the `RequestMapping` annotation.

A class marked as Controller/RestController; it can hold the methods which can act as RequestMapping

```
@Target(value={TYPE,METHOD})
 @Retention(value=RUNTIME)
 @Documented
public @interface RequestMapping
```

Annotation for mapping web requests onto methods in request-handling classes with flexible method signatures.

Modifier and Type	Optional Element and Description
`String[]`	**`consumes`** input Narrows the primary mapping by media types that can be consumed by the mapped handler.
`String[]`	**`headers`** The headers of the mapped request, narrowing the primary mapping.
`RequestMethod[]`	**`method`** The HTTP request methods to map to, narrowing the primary mapping: GET, POST, HEAD, OPTIONS, PUT, PATCH, DELETE, TRACE.
`String`	**`name`** Assign a name to this mapping.
`String[]`	**`params`** The parameters of the mapped request, narrowing the primary mapping.
`String[]`	**`path`** The path mapping URIs (e.g.
`String[]`	**`produces`** OUTPUT Narrows the primary mapping by media types that can be produced by the mapped handler.
`String[]`	**`value`** The primary mapping expressed by this annotation.

@RequestMapping

@GetMapping @PostMapping @PutMapping @DeleteMapping

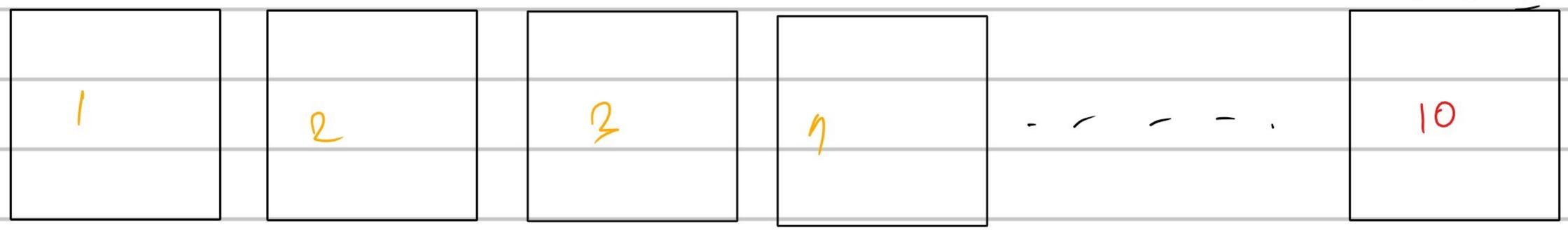
http://localhost:8080/customers/5 → Path Variable

http://localhost:8080/customers?name=Tom → Request Parameter

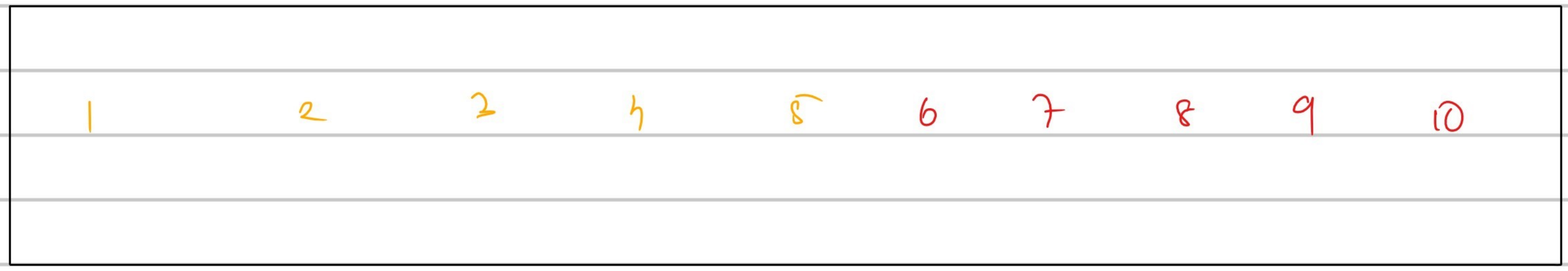
* How to Structure API's in Java Controller classes?

① One Class One API

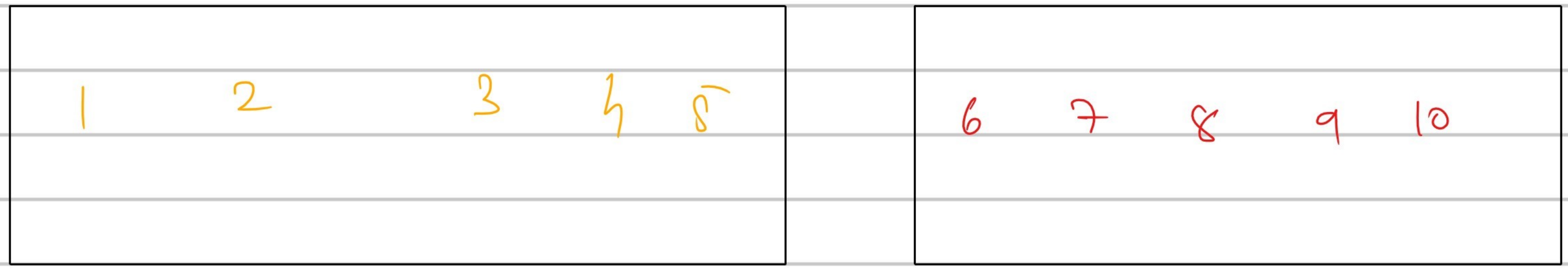
- Customers → 5
- Orders → 5



② One Class All API

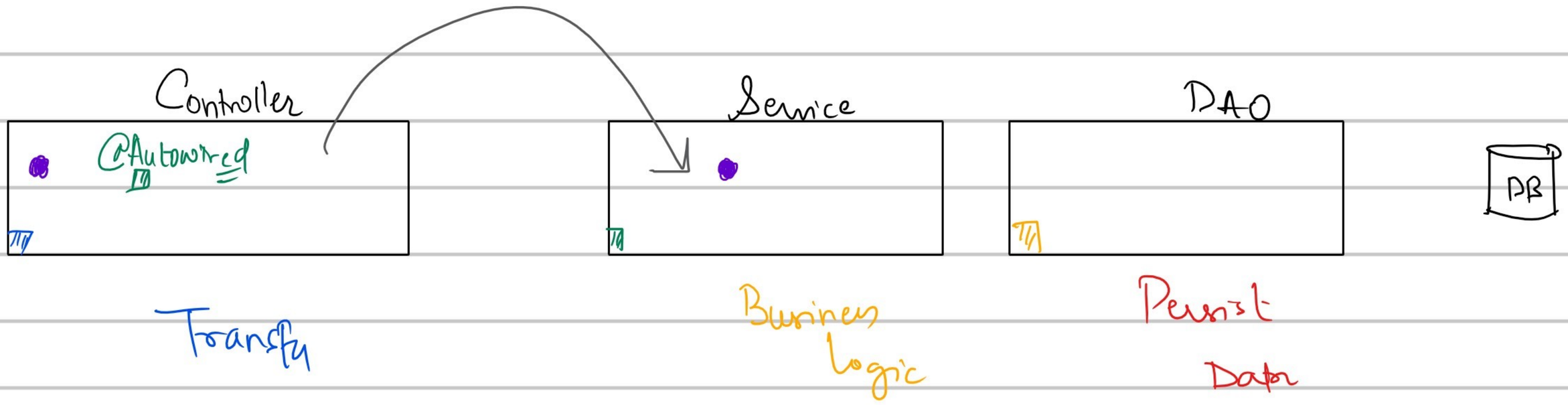


③ Some Class - Some API



* REST API NAMING CONVENTIONS *

Customers

* → body

HTTP VERB	API	Description
GET	/customers	Fetch all customers
GET	/customers/1023	Fetch Customer at id 1023
POST	/customers *	Create a new Customer
PUT	/customers/1025 *	Update a Customer at id 1025
DELETE	/customers/1026	Delete a Customer at id 1026