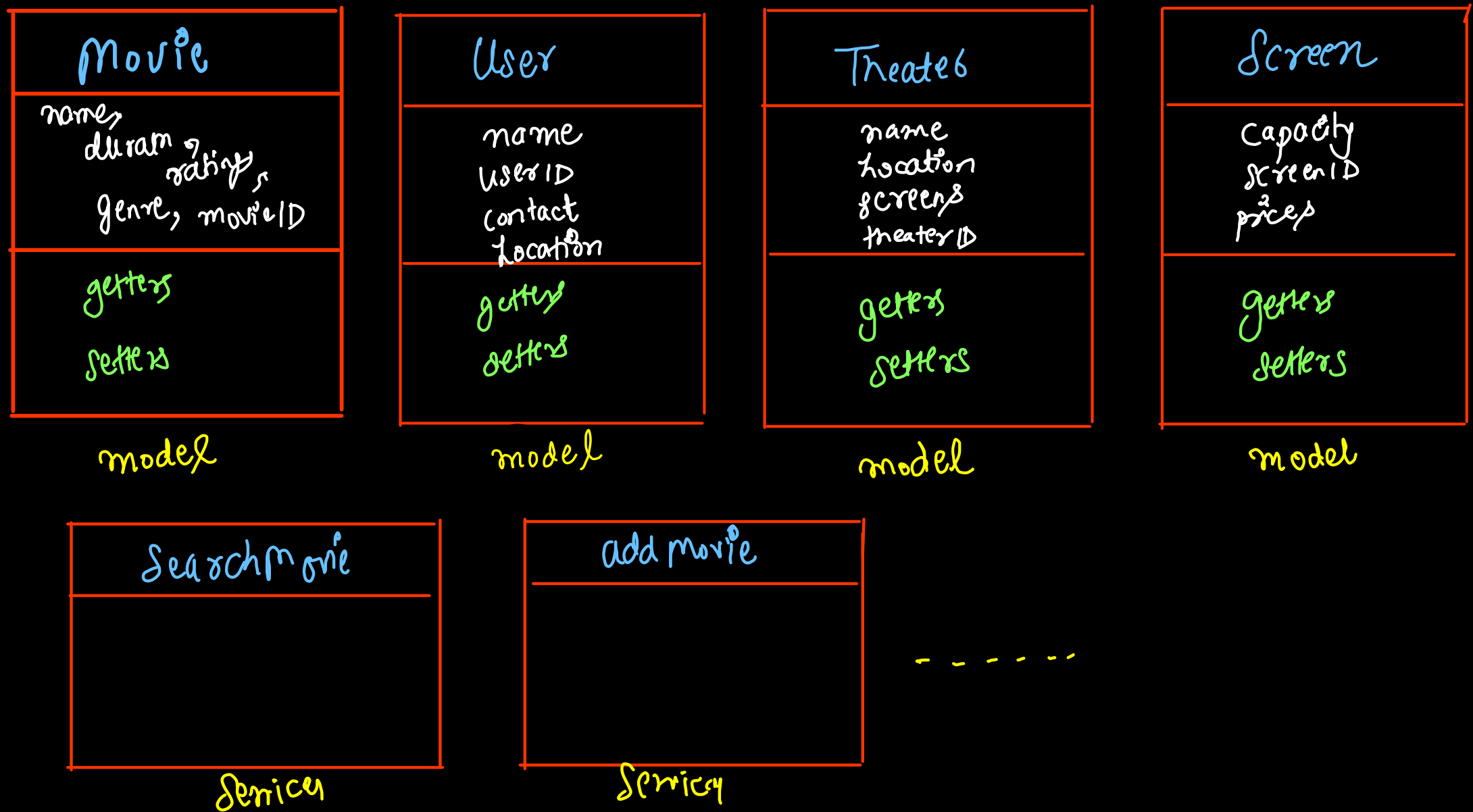
Design BookMyShow



Java Compilation & Execution using Terminal

Compilation :→ "javac filename.java"

↳ It will make .class files for each class in .java file

Execution :→ "java classname"

↳ class should have public static void main(String [] args) method, otherwise no main method found runtime error

↳ To run a program using a particular class, it is not necessary that class should be public.

Important

Public class must have same name as the .java filename. because JVM will look for an entry point, ie. class having main() method, So we are telling JVM that class which is public will be entry point.

Interview Questions

Q) Can classes or Interfaces be "private"?

A) Only inner classes/interfaces can be made private. But outermost (top level) class/interface are not allowed to be private, because if allowed, "no one" will be able to access them & it will become useless.

Q) Can we have no (zero) public classes in .java file?

A) Yes, it is possible but in that case, .java filename should not be same as any classname. Also to run the program, you should use "java classname" containing main method.

Q) Can we have more than one (>1) classes in same .java file?

Yes, we can have more than one classes in same .java file. But the constraint is there: atmost one class can be public. Rest all classes should be protected / default.

Q) Is it possible that no class have the main method?

There will be no compilation error, but there will be "no main method found" run-time error. (whichever class is used as entry point (classname used in "java classname") must have the main method in it.

Q) Is it possible that more than 1 classes have main method defined?

Yes, it is possible that multiple classes can have main method. But again, class used as entry point must have the main method defined.

Packages → used to implement "Data Hiding"

Advantages

- used to group related classes
- used to avoid name space conflicts
- It improves readability, maintainability & easy debugging.
- It is similar to "folder or directory" of Java classes.

Types of Package

- Built in Packages
  - java.lang → String, StringBuffer, Object, System, Math, Integer, etc
  - java.util → Collection frameworks (Arraylist, Hashmap, linkedlist, etc), Scanner
  - java.io → Input/Output (BufferedReader, BufferedWriter, File, InputStream, OutputStream, Serializable, etc)
- User Defined Packages
  - "package" keyword
  - "import" keyword

User Defined Packages

Note → We can have subpackages (nested directories)

→ Folder name is equivalent to package name

→ All java files in the package must have first line as "package packagename;"

→ All .class files (byte code) must be saved in the corresponding package directory / folder.

→ But java files (source code) can be saved anywhere, not necessarily within the same folder/package.

Compilation → "javac full path or relative path / javafilename.java"

Execution → "java fully qualified classname"

↳ packagename.subpackagename.classname

Three ways to import Packages

(1) Using Fully classified classname

eg `java.util.Scanner = new java.util.Scanner(System.in);`

(2) `import packagename.subpackagename.*;`

eg `import java.util.*;`

↳ Import everything from that subpackage

(3) `import packagename.subpackagename.classname.methodname;`

↳ optional (Import specific subpackage/class/method)

eg `import java.util.Scanner;`

↓ to import static method use "`import static ....`";

Note # If package is within the same working directory from where it is being used/imported, then we do not need to set class path environment variables

Class Paths (Required when driver code file & package dependency are in different directories)

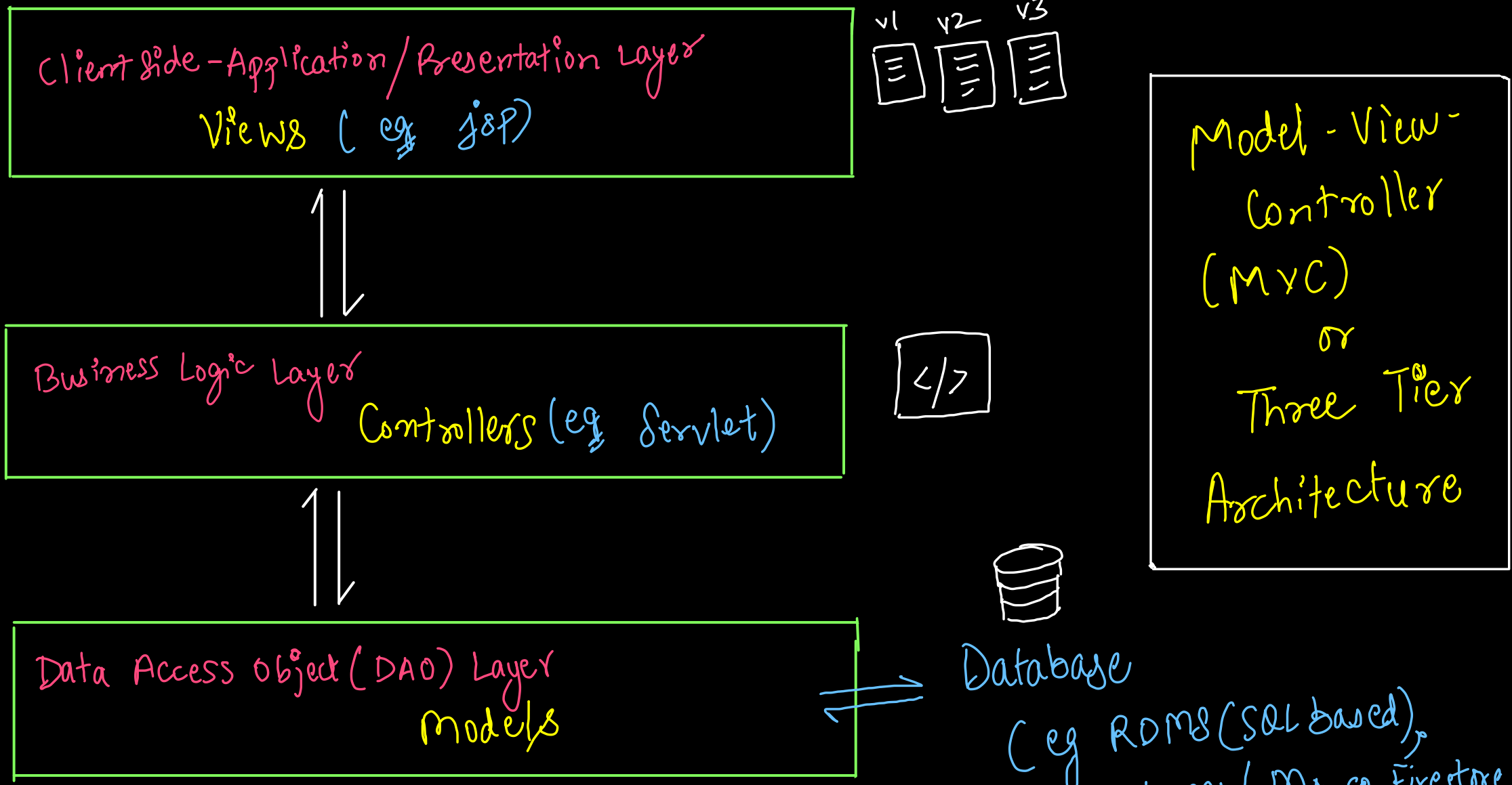
- Temporary Class Path Environment Variables
- Permanent class Path Environment Variables

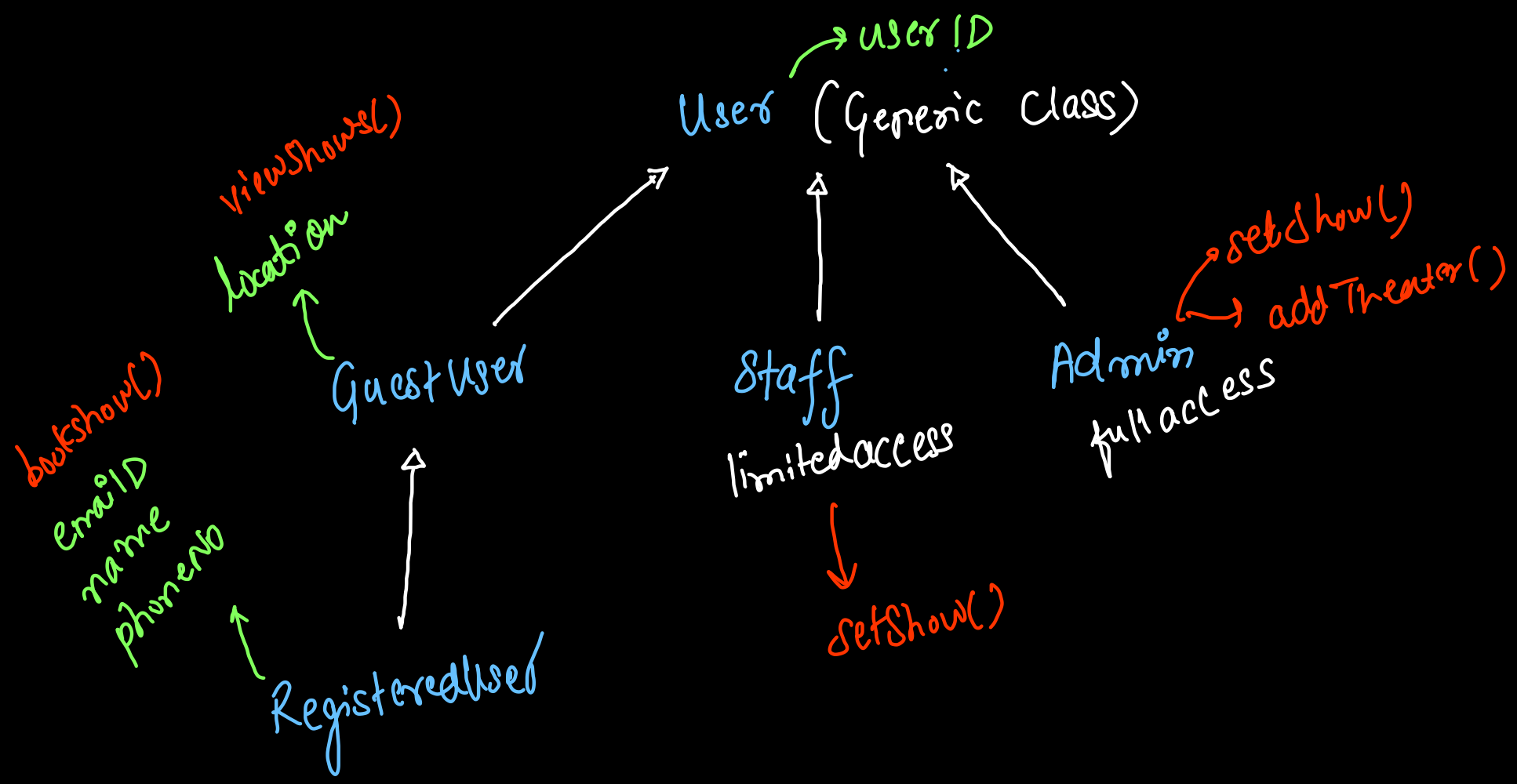
] Explore Yourself.

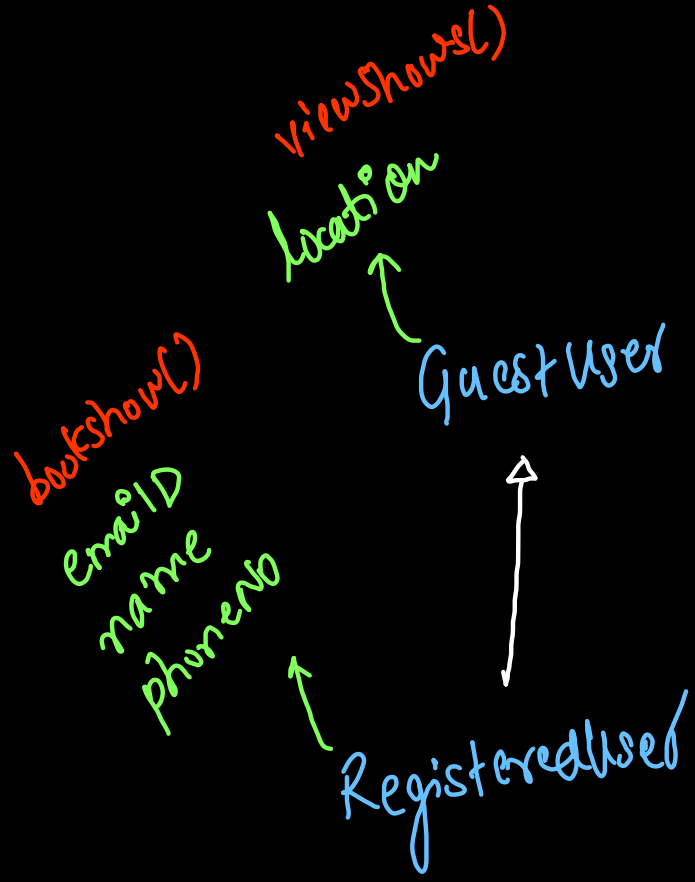
Jar file → Executable file for Distribution of Java programs

Scope of Different Access Specifiers

Access Specifier/ Modifier	Within Same class	Outside class within same package	Different Package Subclass	Different Package non-subclass	
public	Yes	Yes	Yes	Yes	everywhere
protected	Yes	Yes	Yes	No	everywhere except diff package non-subclass
default	Yes	Yes	No	No	same package
private	Yes	No	No	No	within class only


Client side - Application / Presentation Layer
Views (eg jsp)
v1
v2
v3
Model - View - Controller (MVC) or Three Tier Architecture
Business Logic Layer
Controllers (eg Servlet)
</>
Data Access Object (DAO) Layer
Models
Database
(eg RDMS (SQL based), NonSQL (Mongo, Firestore, etc)


User
(Generic Class)
userID
viewShows()
location
GuestUser
bookShow()
emailID
name
phoneNo
RegisteredUser
Staff
limitedaccess
setShow()
Admin
fullaccess
setShow()
addTheater()



RUser User = new RUser();

child of parent class object

child RUser() call ① Execute emailID name phoneNo

super() 1st line

parent GUser() call ② Execute locatn = 'India'

back track

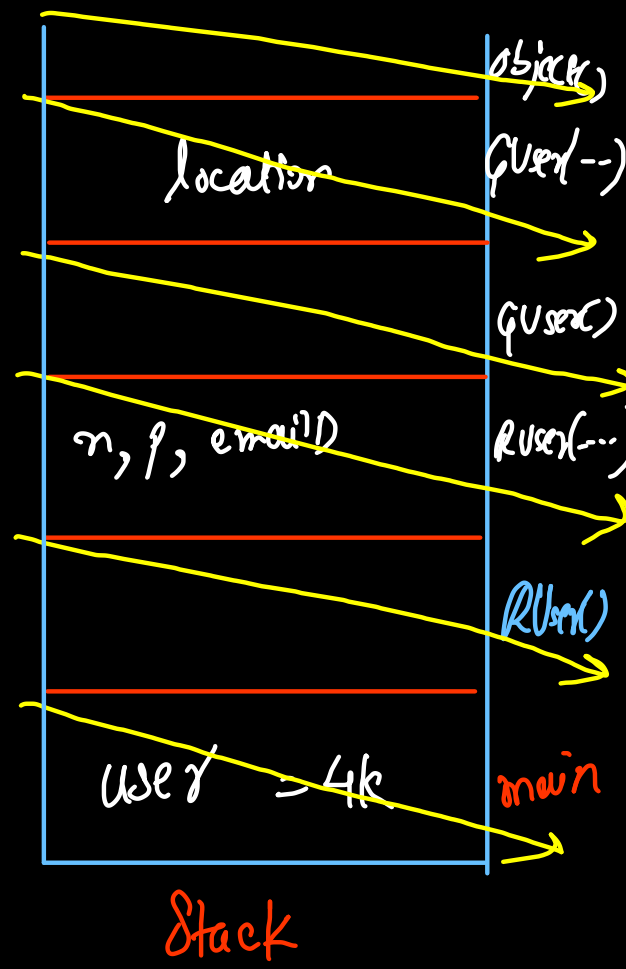
RUser() → RUser(...) → super() → super(...) → Super() object

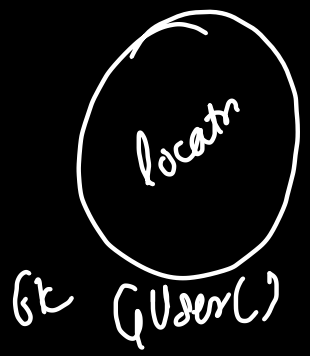
GU() GU(..)

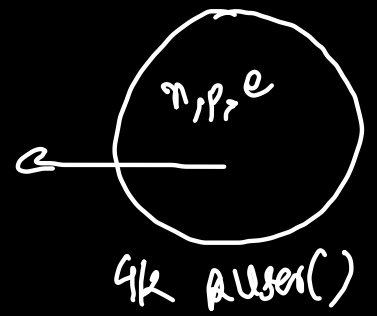
```java
public class RegisteredUser extends GuestUser{
    public String name;
    public long phoneNo;
    public String emailID;

    public RegisteredUser() {
        this(name: "Anonymous", phoneNo: 0l,
        emailID: "anonymous@gmail.com");

        System.out.println(x: "RUser Empty Constructor");
    }

    RegisteredUser(String name, long phoneNo, String emailID) {
        super();
        System.out.println(x: "RUser Parameter Constructor");
        this.name = name;
        this.phoneNo = phoneNo;
        this.emailID = emailID;
    }

    public void bookShow() {…
}
```

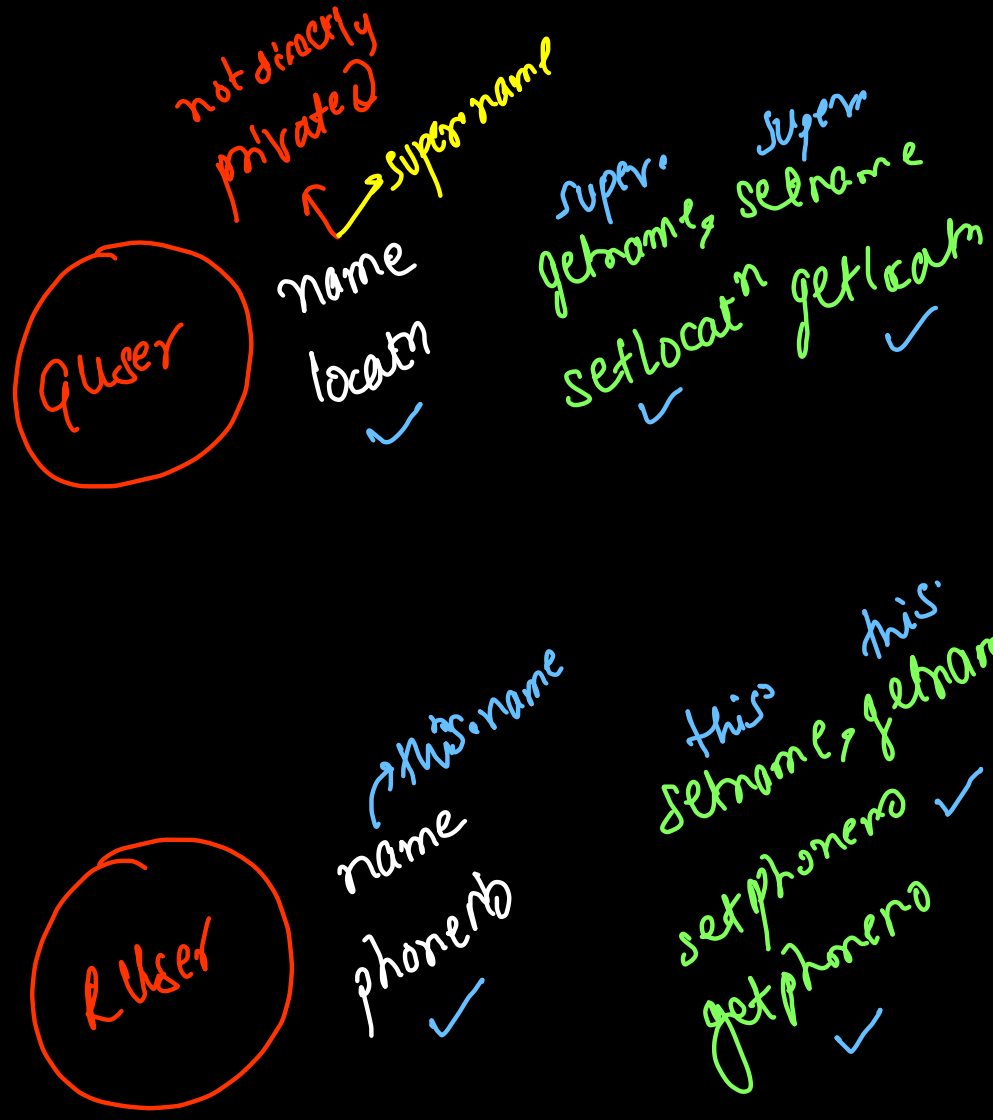
```java
package BookMyShow.users;

public class GuestUser {
    public String location;

    GuestUser() {
        this(location: "India");
        System.out.println(x: "GUser Empty
        Constructor");
    }

    GuestUser(String location) {
        this.location = location;
        System.out.println(x: "GUser Parameter
        Constructor");
    }

    public void viewShow() {…
}
```

```java
public class App {
    Run | Debug
    public static void main(String[] args) {
        RegisteredUser user = new RegisteredUser();
```





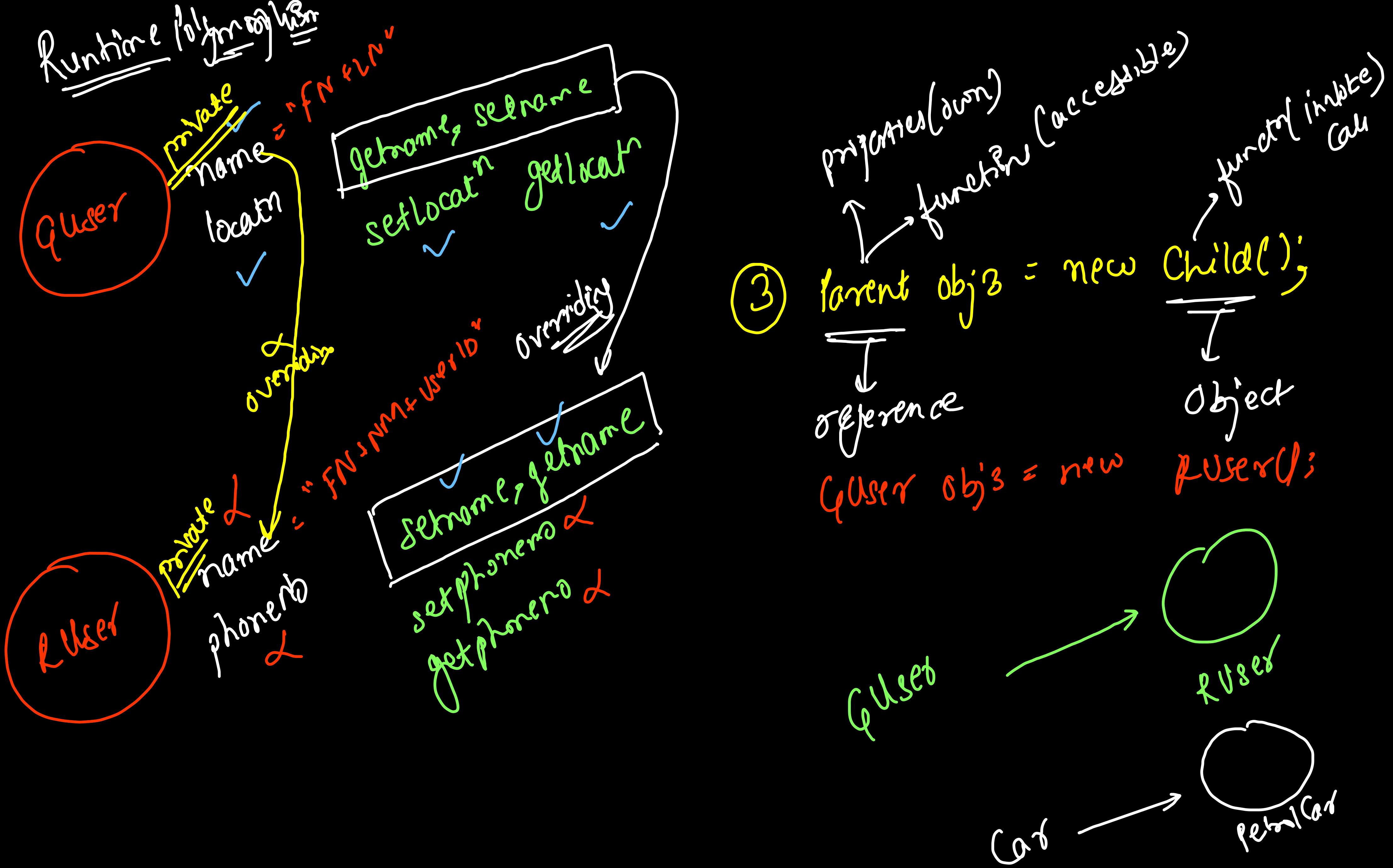




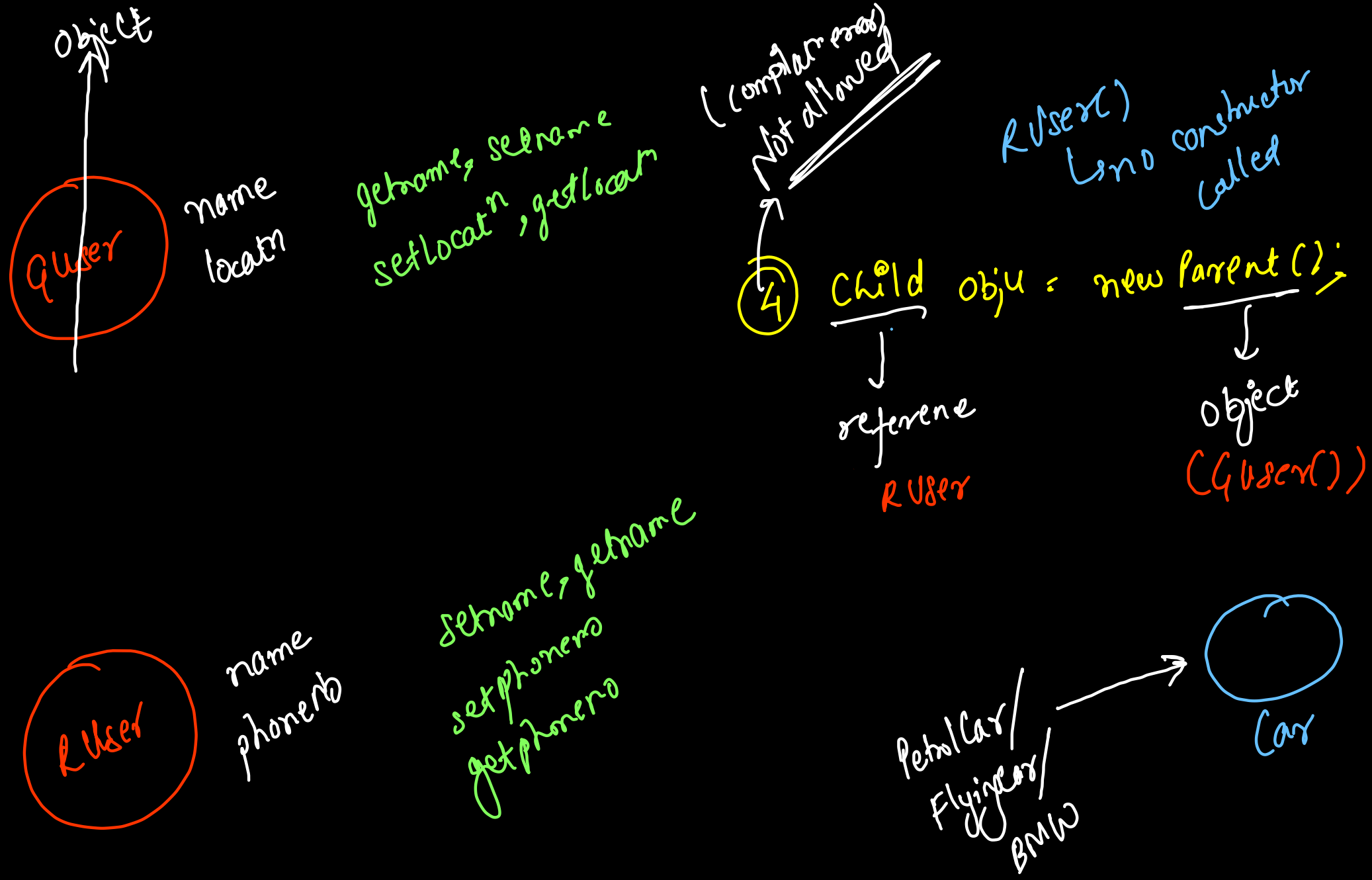
(2) Child obj2 = new Child();

reference

object



Runtime polymorphism
GUser
private
name = "FN + LN"
locatn
getname, setname
setlocatn
getlocatn
overriding
RUser
private
name = "FN + LN + UserID"
phoneno
setname, getname
setphoneno
getphoneno
overriding
propaties (down)
function (accessible)
function (invoke call)
3
Parent obj3 = new Child();
reference
Object
GUser obj3 = new RUser();
GUser
RUser
Car
PetrolCar


Object
GUser
name
locatn
getname, setname
setlocatn, getlocatn
((compiler error)
Not allowed
4
Child obju = new Parent();
reference
RUser
RUser()
Lno constructor
called
Object
(GUser())
RUser
name
phoneNo
setname, getname
setphoneno
getphoneno
PetrolCar/
Flyingcar/
BMW
Car

Interview Question

```java
public class Class1 {

    public void method(Object obj){
        System.out.println("Object");
    }

    public void method(String str){
        System.out.println("String");
    }

    public static void main(String... arg){
        new Class1().method(null);
    }

}
```

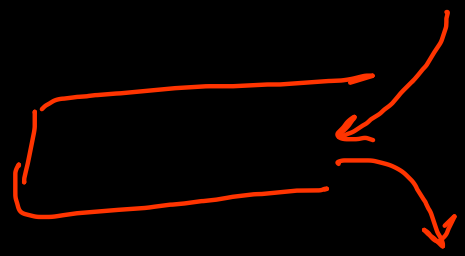
generic

specific

answer

"String" → specific class

Stack (LIFO)



ArrayList

addLast()

removeLast()

LinkedList

addFirst()

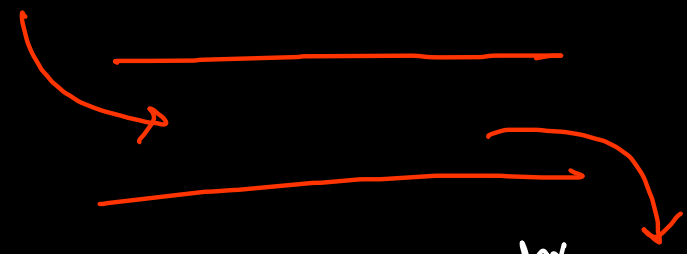
removeFirst()

Queue (FIFO)

LinkedList

addLast()

removeFirst()

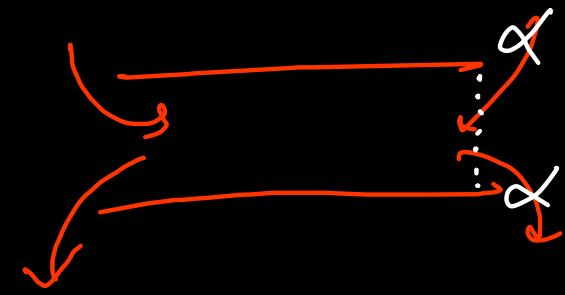


addFirst()

removeLast()

Circular Array

Deque



addFirst()

removeFirst()

addLast()

removeLast()

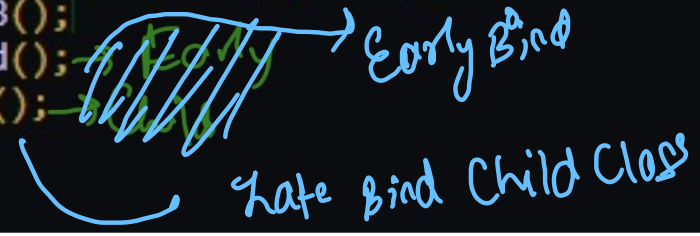
```java
static class Stack{
    ArrayDeque<Integer> q = new ArrayDeque<>();

    public void addLast(int data){
        System.out.println("Add Last of Stack");
        q.addLast(data);
    }

    public int removeLast(){
        System.out.println("Remove Last of Stack");
        return q.removeLast();
    }
}
```

```java
static class Queue{
    ArrayDeque<Integer> q = new ArrayDeque<>();

    public void addFirst(int data){
        System.out.println("Add First of Queue");
        q.addFirst(data);
    }

    public int removeLast(){
        System.out.println("Remove Last of Queue");
        return q.removeLast();
    }
}
```

```java
static class Deque extends Queue{
    ArrayDeque<Integer> q = new ArrayDeque<>();

    public void addLast(int data){
        System.out.println("Add Last of Deque");
        q.addLast(data);
    }
    public void addFirst(int data){
        System.out.println("Add First of Deque");
        q.addFirst(data);
    }
    public int removeLast(){
        System.out.println("Remove Last of Deque");
        return q.removeLast();
    }
    public int removeFirst(){
        System.out.println("Remove First of Deque");
        return q.removeFirst();
    }
}
```

```java
public static void main(String[] args){
    Queue q = new Deque();
    q.addFirst(10); // Allowed
    // q.addLast(20); Not Allowed
    System.out.println(q.removeLast()); // Allowed
    // System.out.println(q.removeFirst()); // Not Allowed

}
```

```java
class A{
    public void earlyBind(){
        System.out.println("Early Bind");
    }

    public void lateBind(){
        System.out.println("Late Bind in Parent Class");
    }
}

class B extends A{
    @Override
    public void lateBind(){
        System.out.println("Late Bind in Child Class");
    }
}

public class Main{
    public static void main(String[] args){
        A obj = new B();
        obj.earlyBind();
        obj.lateBind();
    }
}
```



Overloaded

↳ Same class a functions

Overided class

Parent - child class

↓ ref ↓ object

Abstract Datatype

abstract

Stack, Queue

af rf af rl

Priority queue

add remove

Concrete Data Structure

Implementation

Deque

Binary Heap

```java
static abstract class Stack{
    ArrayDeque<Integer> q = new ArrayDeque<>();

    public void addLast(int data){
        System.out.println("Add Last of Stack");
        q.addLast(data);
    }

    public int removeLast(){
        System.out.println("Remove Last of Stack");
        return q.removeLast();
    }
}
```

```java
public static void main(String[] args){
    Stack stk = new Stack();
}
```

```
Finished in N/A
Line 54: error: Stack is abstract; cannot be instantiated
[in Main.java]
        Stack stk = new Stack();
                    ^
```

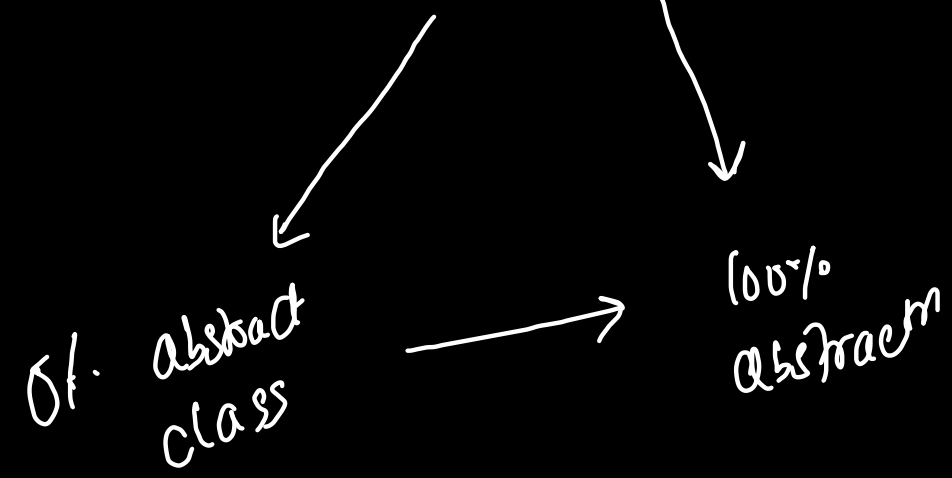
```java
static abstract class Stack{
    public abstract void addLast(int data);
    public abstract int removeLast();
}
```

```java
static class Deque extends Stack{
    static ArrayDeque<Integer> q = new ArrayDeque<>();

    public void addLast(int data){
        System.out.println("Add Last of Deque");
        q.addLast(data);
    }
    public void addFirst(int data){
        System.out.println("Add First of Deque");
        q.addFirst(data);
    }
```

```java
public static void main(String[] args){
    Deque q = new Deque();
    q.removeLast();
}
```

```
Finished in N/A
Line 23: error: Deque is not abstract and does not override
abstract method removeLast() in Stack [in Main.java]
    static class Deque extends Stack{
           ^
```

Abstract class



1 abstract method (no body) → mandatory → class should be abstract !!

```java
static abstract class Stack{
    public abstract void addLast(int data);
    public abstract int removeLast();
}
```

```java
public static void main(String[] args){
    Stack stk = new Deque();
    stk.addLast(10);
    stk.removeLast();
}
```

```
Finished in 74 ms
Add Last of Deque
Remove Last of Deque
```

```java
static class Deque extends Stack{
    static ArrayDeque<Integer> q = new ArrayDeque<>();

    public void addLast(int data){
        System.out.println("Add Last of Deque");
        q.addLast(data);
    }
    public void addFirst(int data){
        System.out.println("Add First of Deque");
        q.addFirst(data);
    }

    @Override
    public int removeLast(){
        System.out.println("Remove Last of Deque");
        return q.removeLast();
    }

    public int removeFirst(){
        System.out.println("Remove First of Deque");
        return q.removeFirst();
    }
}
```

```java
static abstract class Stack{
    int nonStaticData = 0;
    Stack(){
        System.out.println("Abstract Class Constructor");
        nonStaticData = 100;
        System.out.println(this.nonStaticData);
    }
    public abstract void addLast(int data);
    public abstract int removeLast();
}
```

```
Finished in 89 ms
Abstract Class Constructor
100
Add Last of Deque
Remove Last of Deque
```

Abstract class will have

↓

1) Constructors Yes

2) Non static data (Object Data members)

3) This pointer

Constructor — No

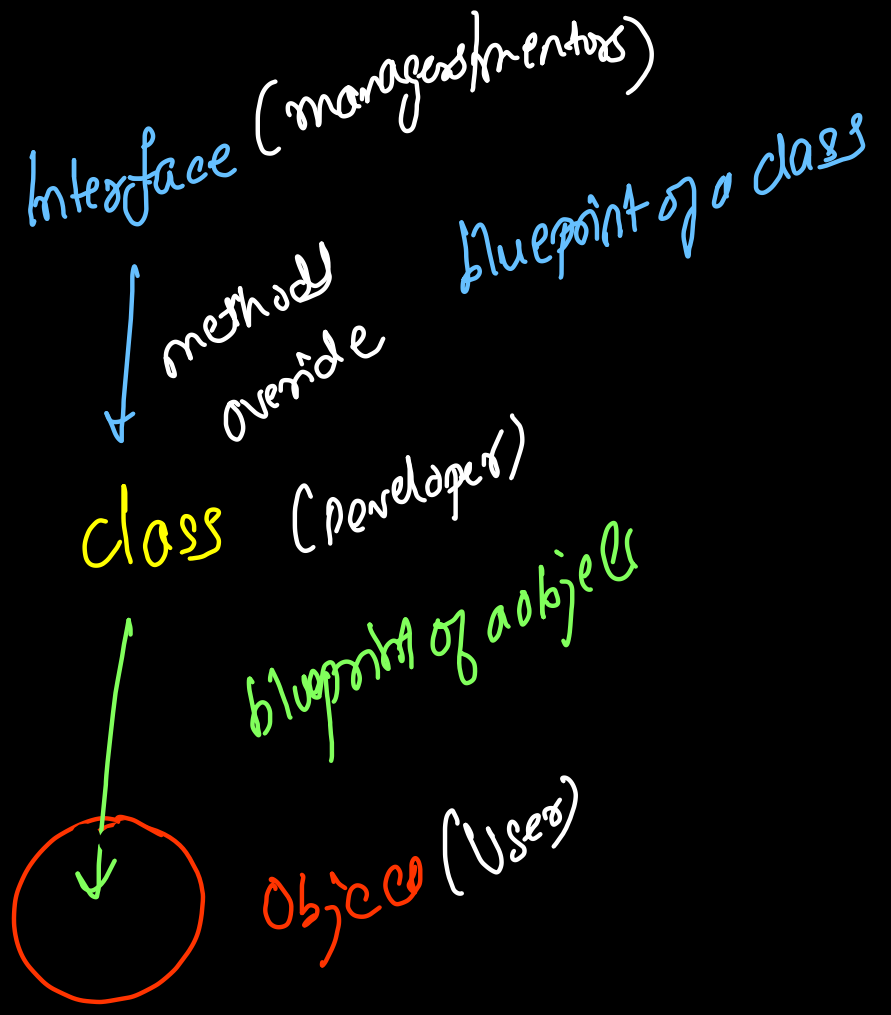
1) super()
2) override

Static methods — No

abstract

Static methods cannot be overrided.

& Abstract method must be overrided



Interface

1. not in class hierarchy
2. Constructor ✗
3. non static data ✗
4. this ✗
5. multiple interfaces can be implemented

```java
static interface Stack{
    void addLast(int data);
    int removeLast();
}

static interface Queue{
    void addFirst(int data);
    int removeLast();
}
```

```java
public static void main(String[] args){
    Stack stk = new Deque();
    stk.addLast(10);
    stk.removeLast();

    Queue q = new Deque();
    q.addFirst(10);
    q.removeLast();
}
```

```java
static class Deque implements Stack, Queue{
    ArrayDeque<Integer> q = new ArrayDeque<>();

    public void addLast(int data){
        System.out.println("Add Last of Deque");
        q.addLast(data);
    }
    public void addFirst(int data){
        System.out.println("Add First of Deque");
        q.addFirst(data);
    }
    public int removeLast(){
        System.out.println("Remove Last of Deque");
        return q.removeLast();
    }
    public int removeFirst(){
        System.out.println("Remove First of Deque");
        return q.removeFirst();
    }
}
```

```
Finished in 94 ms
Add Last of Deque
Remove Last of Deque
Add First of Deque
Remove Last of Deque
```

final keyword

1. Data Members → constant (can be static as well as non-static)
2. Reference Variables → Reassignment not allowed, data can be changed
3. Classes → cannot be extended / inherited! = all final methods!
4. methods → cannot be overided! = early Binded methods!

① Change in return type

```java
class Parent {
    int parentData;

    public void getObject() {
        System.out.println(x: "Parent getObject");
        // return this;
    }
}

class Child extends Parent {
    int childData;

    public int getObject() {
        System.out.println(x: "Overrided");
        return 0;
    }
}
```

same return type

compiler error ⇒ Ambiguous function!

Overriding is not possible with different return type!

Covariant type!

```java
class Parent {
    int parentData;

    public Parent getObject() {
        System.out.println(x: "Parent getObject ");
        return this;
    }
}

class Child extends Parent {
    int childData;

    public Child getObject() {
        System.out.println(x: "Child getObject ");
        return this;
    }
}

public class App {
    Run | Debug
    public static void main(String[] args) throws Exception {
        Parent obj1 = new Parent();
        System.out.println(obj1.getObject());

        Child obj2 = new Child();
        System.out.println(obj2.getObject());

        Parent obj3 = new Child();
        System.out.println(obj3.getObject());
    }
}
```

Overriding

↳ Parent getObject

↳ Child getObject

↳ Child getObject

```java
interface IParent {
    public IParent getObject();
}

class Child implements IParent {
    int childData;

    public Child getObject() {
        System.out.println(x: "Child getObject ");
        return this;
    }
}

public class App {
    Run | Debug
    public static void main(String[] args) throws Exception {
        IParent obj = new Child();
        System.out.println(obj.getObject());
    }
}
```

Interface Returning an Object!