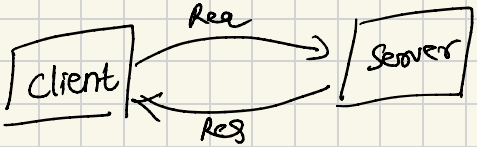
Sanket Singh

&

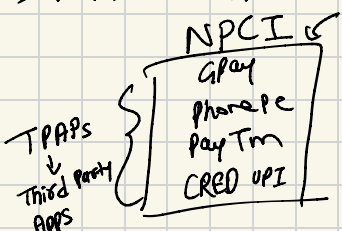
Ashwini upadhyay

NOTES

☆ REST API

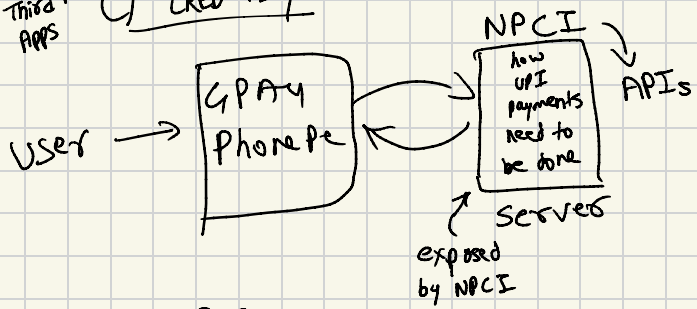


→ API – Application Programming interface



UPI ← payment mechanism that works in India

→ NPCI provide UPI api to TPAPs



REST API

↳ Rest is a set of recommendations to prepare modern web api's

↳ define kind of a style guide to make APIs

↳ If we follow Rest Api's convention then our apis become more consistent, readable, standard that a lot of developer use.

↳ design of API's is more important

↳ Rest alternative SOAP, gRPC

① Most widely accepted

② framework (Ruby, Django etc) heavily tied with Rest.

③ Easy to follow REST

Principles of REST

1) all the apis are designed around Resources

any kind of object, data, service or real life entity that client can access!

Ex: in an e-commerce app
↳ Product is a resource &
creation of Product is action.

localhost:3001/products/2
port route

↳ Your route should be based on resources.

/products /customers
/orders/2 /categories

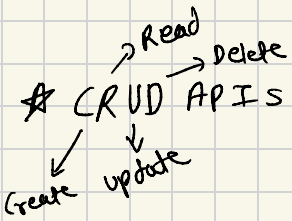
↳ If you want to send & receive data, use JSON format for it.
↳ Javascript object notation

Ex
```
{
  "name": "Aman"
  "company": "Google"
  "age": 30
}
```

② If you want to send data from client to server there are 3 ways

① Query Params → /route?key1=value1&key2=value2
② URL Params → path param /products/23 /todo/10 /todo/:id
③ Body Params → URL headers http method

```
{
  "name": "Aman"
  "company": "Google"
  "age": 30
}
```



HTTP methods

↳ GET – fetch data
POST – create data
PUT – update data
PATCH – partially update
DELETE – delete data

↳ Create Product API
/products → POST
↳ Body: {
name: ___
price: ___
}

↳ Get all Products
/products → GET

Routes are same but we can differentiate with help of HTTP methods POST, GET...

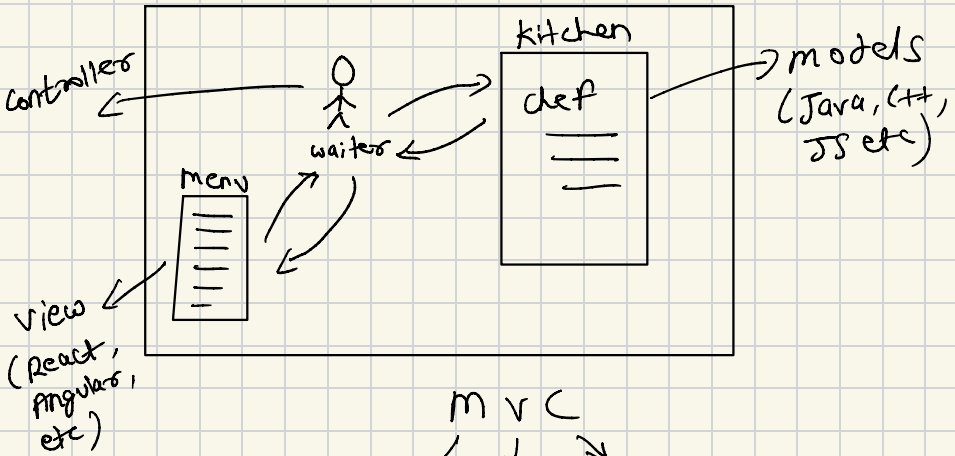
↳ Get a product
/products/:product_id → GET

↳ Delete a product
/products/:product_id → Delete

Note: HTML form in frontend only support GET & POST to use all method we use javascript -

MVC Models Repositories Services Controllers

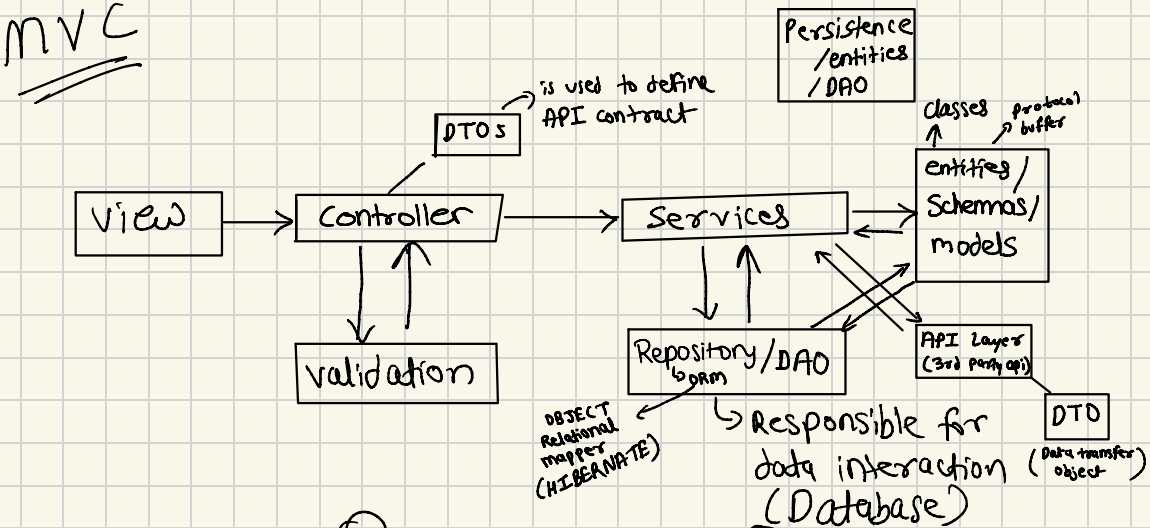
→ Separation of Concern

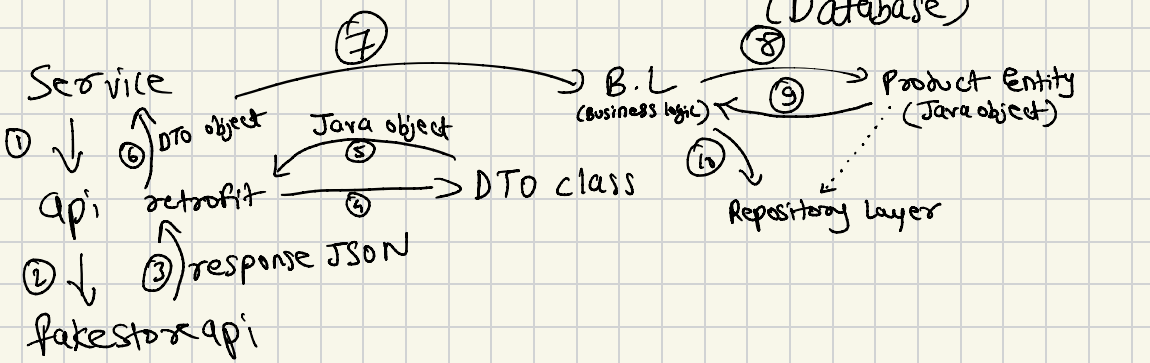


MVC

- **models** → part of your code where all the logic resides. (Business logic)
  - Algorithm (services)
  - Entities (Schema/models)
  - Data interaction (Repositories)
- **views** (frontend)
- **controllers** → that part of code which is responsible for collecting incoming user req & provide it to the models layer. Then whatever response models gives controllers take that & send it to the user.

MVC





MVP
↓
minimum viable product

HLD UBER → location based
→ Real time communication
→ database
and more

Booking some kind of cab ride

Function Requirements → Product manager

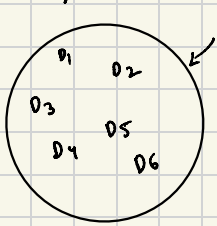
→ 3 type of users of our app
↳ Passengers ↳ Riders ↳ Admins

→ 1) Onboarding
↳ for onboarding of a passenger, we take there phone no., email, password
↳ for onboarding as a Rider, apart from phone no., email, password, additional documents, subject to approval from admins.

2) Booking A Ride
↳ passengers select the start point & drop location, then selects for type of ride.
↳ the request for the ride goes to all the nearby available Drivers who support that type or beyond.

type
↳ Uber go
↳ Uber premier
↳ Uber auto
↳ Uber bike

What driver see
Price of ride
Accept decline ignore



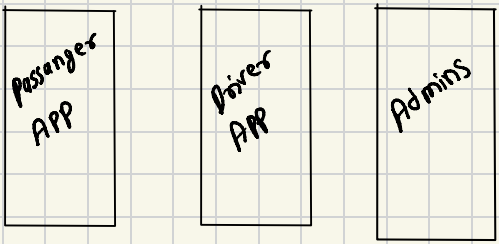
3) Payments
   ↳ Prepaid } 3rd party payment Gateway
   ↳ Postpaid }

4) Ride OTP

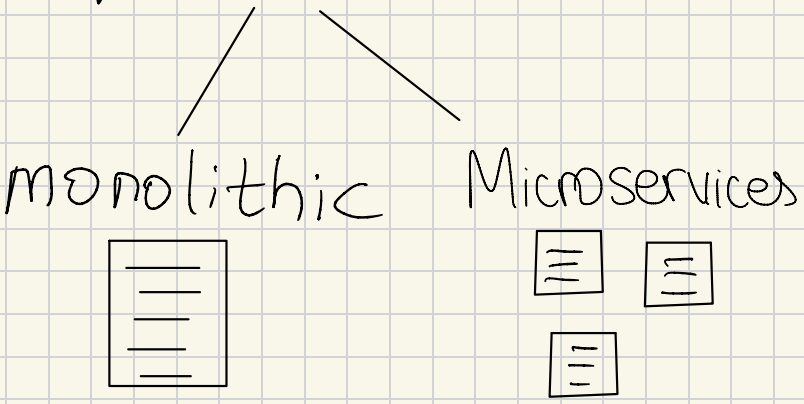
5) Price Calculation

6) Rating → Passenger
          ↳ Driver

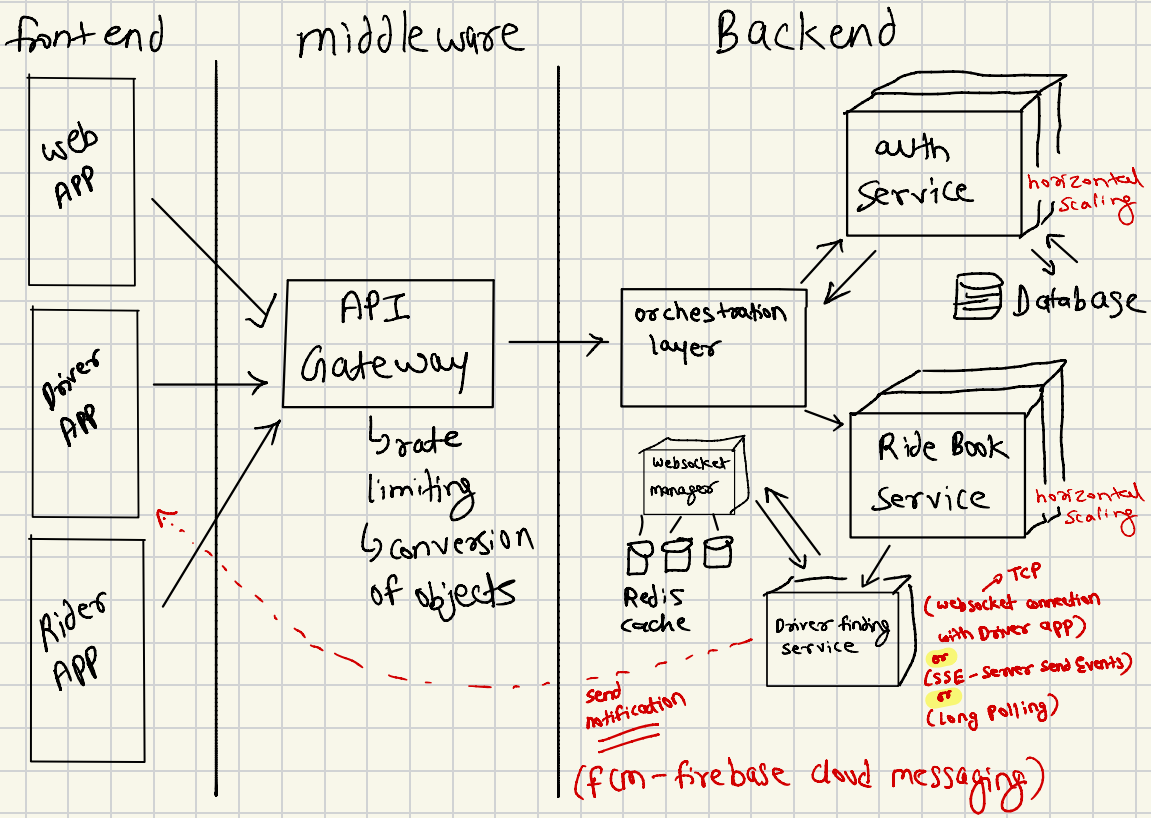
7) Passenger should be able to view their past rides & Driver should be able to see income of day, total income, income of last month & past rides



Architecture



(HLD)



JAVA → JDBC (Java database connectivity)

↳ helps in direct DB interaction we can directly use JDBC to execute queries to the DB. Query are written as strings, & then pass those in function exposed by JDBC.

Uses connection Drivers / Driver managers / Connection pool.

JPA (Java Persistence API)

↳ JPA is a specification for accessing, persisting & managing data between Java objects & Relational tables.

(Interfaces + Annotations)

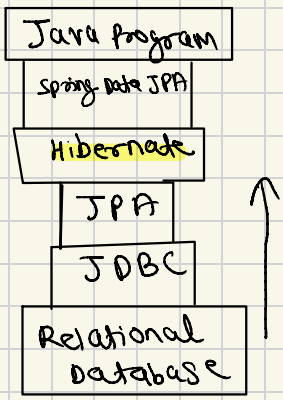
(Class objects methods) JPA → (abstraction layer) JDBC (Raw Query)

ORM (Object relational mapper)

↳ Hibernate

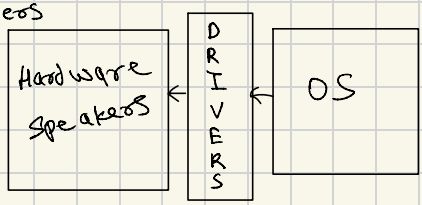
↳ open JPA

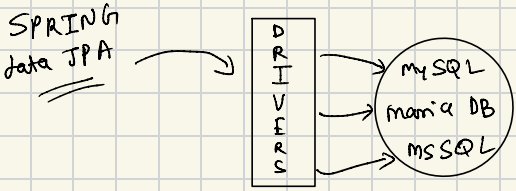
↳ Eclipse Link

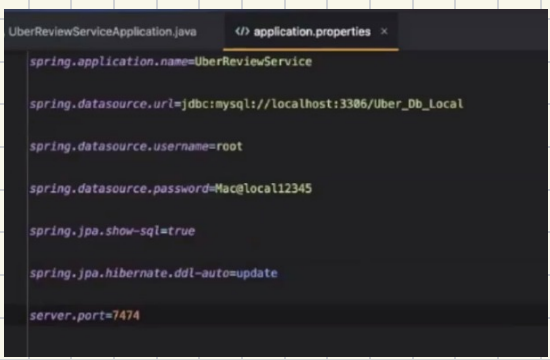


Lombok → this helps us to get annotation like @getter, @setter which are helpful for preparing classes

#Drivers







Doctor has many patients through appointments

Patients belongs to many doctors through appointments

} Third table ↓ Join table / through table

Doctor

primary key

d_id	name
1	Anand
2	Anurag

Patients

primary key

p_id	name
1	Aman
2	Sanket

appointments

Foreign key

a_id	time	d_id	p_id
1	...	1	2
2	...	2	2
3	...	2	1

pair_of (d_id, p_id)

(many to many) Relation

Mode

OneToOne → Default → Eager

OneToMany → Default → Lazy

ManyToMany → Default → Lazy

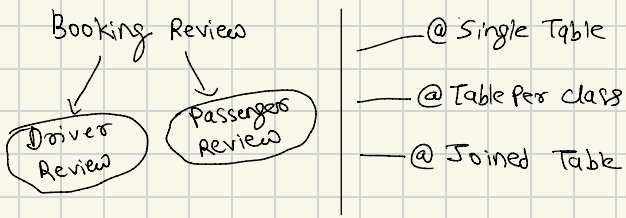
ManyToOne → Default → Eager

Handling Inheritance in Spring Data JPA

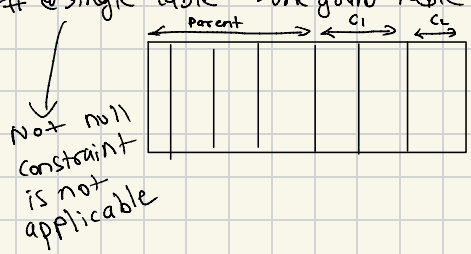
① @MappedSuperClass

↳ No table for the parent class

↳ One table for each child class having its own attributes & parent class attributes.



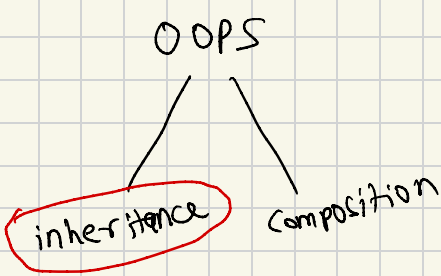
@Single Table → one grand table



@Table Per class

↳ Every thing is same as mapped super class except the fact that parent also has a dedicated table.

↳ child will have all properties of parent.



Mapped Super class

Table Per Class

Single Table

Joined Table

"is-a"

"has-a"

Composition

(Read CASCADE TYPE)

A driver has many reviews
↳ one to many relationship

```
class Driver {
    List<Reviews> reviews;
}
class Booking {
        Booking date;
        Booking time;
    Review Bookingreview;
}
```

- 1:1 one to one
- 1:n one to many
- n:1 many to one
- m:n many to many

→ 1 Booking has one review
& a review belongs to a booking;
1:1 A:B (any one table has id of another table)
(any table have the foreign key)

→ 1:n 1 : n (driver has many reviews & review belongs to a driver)
n:1 Driver Review
↳ have driver_id (FK) (many side have foreign key)

User → enum → Passenger
→ Driver

```
class User {
@one to many(mapped by="Passenger");
private List<Booking> booking;
@ one to many(mapped by="Driver");
private List<Booking> booking;
}
```

```
class Booking {
@many to one
private User driver;
@many to one
private User passenger;
}
```

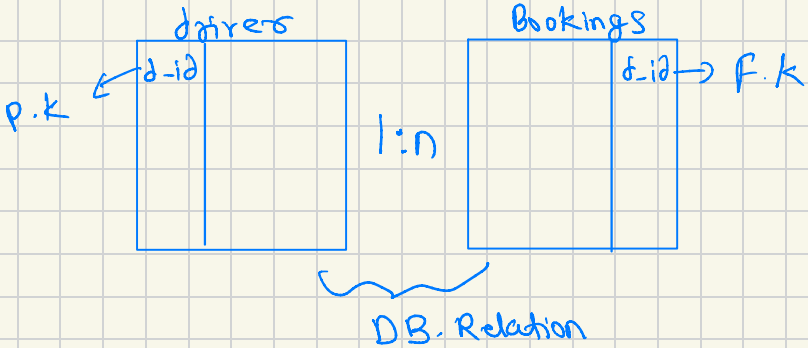
★ Imp

→ we have an api which looks something like this

GET → /api/v1/drivers/bookings

Req-body → {
driver_ids: [2, 3, 4, 5...]
}

expected output → we should be able to fetch driver & booking of each and every driver.



With respect to app performance, in driver model booking should be load lazily.

→ Driver findByID(Long id);

Booking → Lazy

Driver d = driverRepo.findByID(12);
d.getBookings()

↳ List<Driver> findAllByID in (List<Long> driverID);

Select * from Driver where id in [, ,];

List<Driver> drivers = driverRepo.findAllByIDin(——)

```
for (driver : drivers) {
        driver.getBooking();
}
```

← disaster

N driver

↓

N+1 problem

☆ Schema migration

→ RDBMS → Tables ↕ Relations } when the product features & requirements are huge their will be a lot of tables & lot of relations.

DB automatically update

- when we create a new Entity class
- when we update an old Entity class

☆ Database migration with flyway

↓

schema versioning

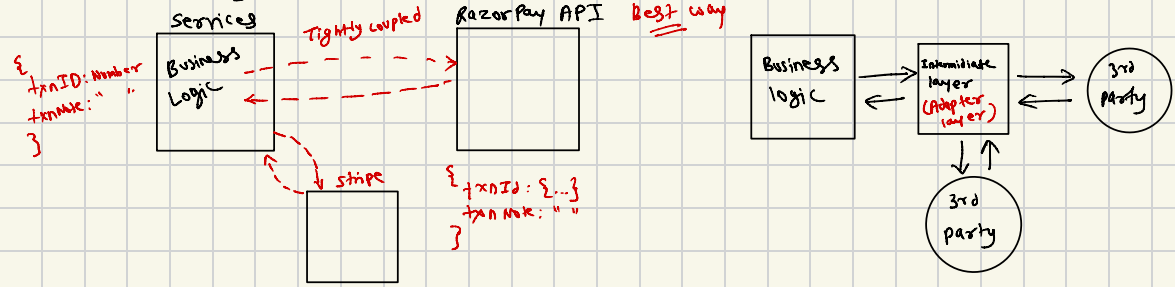
Adapter Design pattern

↳ 2 incompatible interfaces can work together

↳ Adapter pattern acts like a bridge btw 2 classes or Interfaces or objects that cannot directly work together due to different structures.

Adapter pattern ensures that our codebase remains easily maintainable when it is dependent on 3rd party libraries.

↳ Should your codebase having core business logic directly talk to the 3rd party : No



class → PaymentProcessor { ← concrete class

```
RazorPay rg = new RazorPay();
rg.makePayment();
rg.getStatus(p_id);
}
```

after making changes

```
class PaymentGatewayAdapter{
  RazorPay rg = new RazorPay
  initPayment(){
    rg.makePayment()
  }
  checkStatus(pid){
    rg.getStatus(pid)
  }
}
```

↑ Violating OCP

```
class PaymentProcessor{
  paymentGatewayAdapter pg = new P__();
  pg.initPayment();
}
```

☆ POSTMAN

1) Use environments as per the environments

Developer (env based variables)

↳ Local

http://localhost:3000/v1/api/problems

↳ UAT/Preview

http://{{uat-url}}/v1/api/problems

↳ Production

http://{{prod-url}}/v1/api/problems

↓

ac-problem-service.onrender.com

environments

local	URL	default	localhost:3000	localhost:3000
UAT	URL	default	ac-problem-service.onrender.com	ac-problem-service.onrender.com
prod	URL	default	ac-problem-service.onrender.com	ac-problem-service.onrender.com

2) Load/Performance Testing (API)

↳ SWE → sole consumer → single user

↳ 20 users → parallel users to the api

POSTMAN → Collection (click ...) → Run collections → Performance

↳ Performance test

load: fixed, virtual: 10, Test: 1 min

3) Test the api using scripts

↳ unit tests

postman

Params Auth Headers . . . Tests Settings

```
pm.test("Status code 200", function () {
    pm.response.to.have.status(200);
});
pm.test("Content-Type is application/json", function () {
    pm.expect(pm.response.headers.get('Content-Type')).to.include('application/json');
});
```

4) monitoring

↳ monitor health means continuously checking the health and performance of the APIs.

postman → collections → Algo (. . .) → monitor collection

Auth ⟶ Authentication (who are you?) (Register, Log in)
↳ Authorization (what you can do?)

Register
↳ mechanism to collect mandatory user details.
↳ Store these details in your DB.
Things to take care of
↳ Sensitive info. (password) ⇒ Encrypted (Bcrypt hashing algorithm)
↳ Handling uniqueness
↳ Email, Phone no...

Signup /api/v1/auth/signup

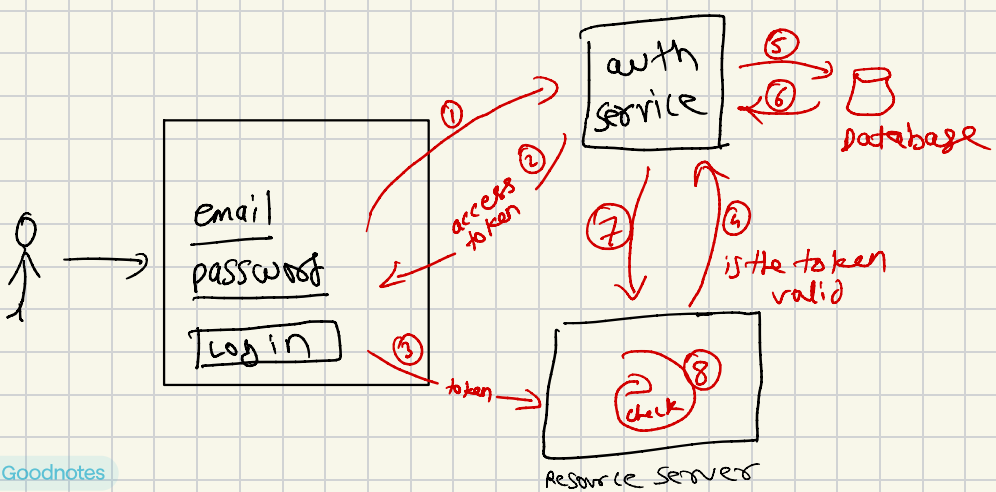
Post ↳ { } ← Req Body

Resource owner (user itself)

Client (Bookmyshow.com, leetcode.com)

Resource Server (servers of Bookmyshow, leetcode which is having user informations)

Auth Server (Google/fb/Github server)



Authentication

① Token based auth (Bookmyshow) (JWT)

② Session Based (Hotstar)

③ Hybrid (Token + session)

② Session ID → token → user

Hotstar (2 devices) → (session based auth)

token saved in DB

check → yes

logged in

(Note: token is also c/as session ID)

(session storage)

① Token based auth (Stateless) → info. of the user

↳ Generate special token which is not present in the DB.

(JWT)

User ⇄ auth server

token with expiry

(Note: here user need to save token i.e. local storage or else)

JWT ⇒ JSON web Token (String)

algo	data	base 64 encoding
Headers	payload	Signature

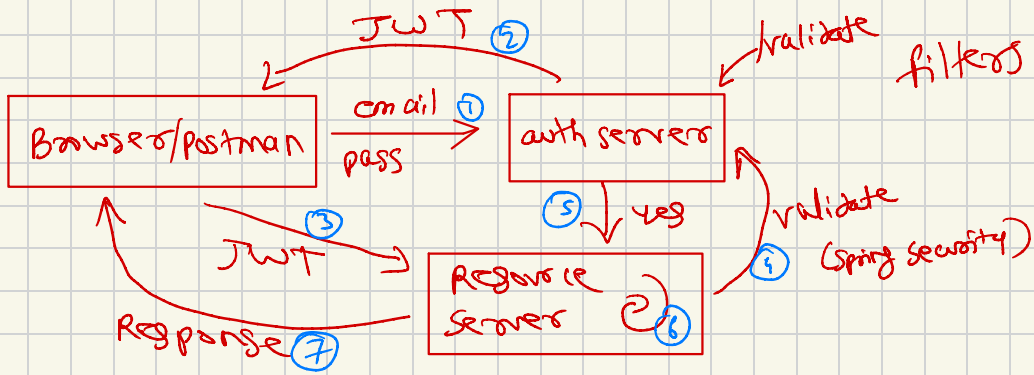
(JWT Token not save sensitive info like password)

/login

HTTPS → ReqBody { email password }

→ authserver → JWT Token { user email expiry }

Spring boot package (JJWT)



Testing

1) Testing
2) Unit Testing
3) Junit
4) Integrate UTs in our uber project

lec 1

Java

- 1995 (official)
- 1991 (developed) — OAK — Sun microsystem
  - ↳ Acquired by oracle S.M.

Java
- J2SE (Java Standard edition) eg: Calculator (standalone App)
- J2EE (Java Enterprise edition) eg: facebook (web apps)
- J2ME (Java Mobile edition) eg: android apps

Java features

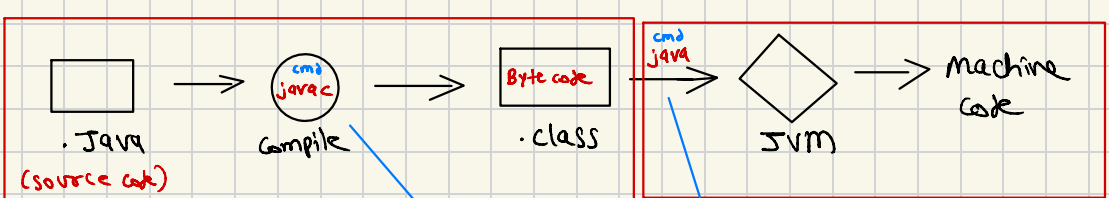
"write once run anywhere"

- ↳ Simple
- ↳ platform independent
- ↳ Robust (strong)
  - ↳ Automatic memory management
  - ↳ Exception Handling
- ↳ OOPS (Object Oriented Programming System)
- ↳ Secure

Env. Setup

- ↳ Dowload JDK & JRE
  - → JRE gives us platform to execute program and JVM is inside JRE
  - ↳ provide set of tools to develop program
- ↳ set path : system variable : C:\Program files\Java\jdk1.8.0_202
  user variable : %JAVA_HOME%\bin

OS is a software (platform) used to establish connection between user and computer.



File

Hello.java → javac filename.java → java className

javac Hello.java

java Hello

Java Program Structure

lec 2

1) package statement
2) import statement
3) class declaration
4) methods / functions
5) variables

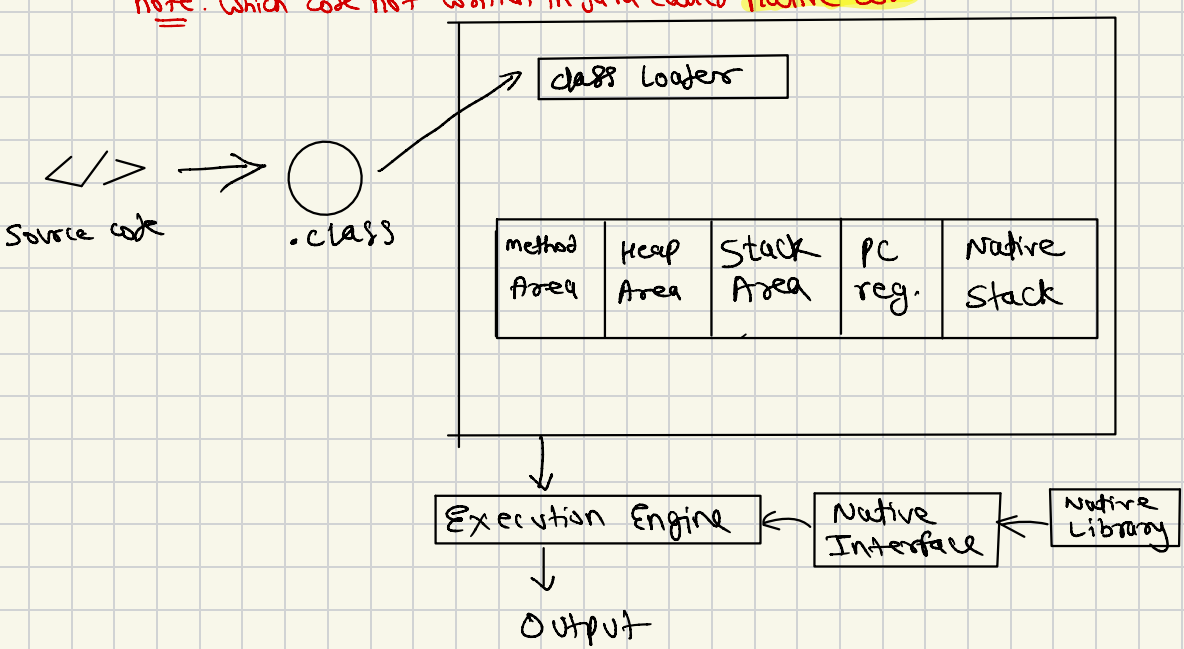
Interpreter: line by line (python)

Compiler: whole code (Java)

JVM architecture

1) classloader subsystem : It will load .class file into JVM
2) method Area : Class code will be stored here
3) Heap Area : object will be stored into heap area
4) stack Area : method execution information will stored here
5) PC register : It will maintain next line information to execute
6) Native stack : It will maintain non-java code execution information
7) Execution (Interpreter) : It is responsible to execute the program and provide output
8) Native Interface : It will load native lib. into JVM
9) Native Libraries : non java lib. which are required for native code execution

note: which code not written in java called native code



Data Types

Lec 3

Primitive

1) char → 'a' → 2 bytes (in Java) bec. ASCII + Spec. char.
2) boolean → true/false
3) byte → −128 to 127 → 1 bytes ⇒ 8 bits
4) Short → 2 bytes
5) int → 4 bytes
6) long → 8 bytes
7) float → 4 bytes
8) double → 8 bytes

Non-primitive

1) Array → int[] arr = [10, 20, 30] , string[] cars = {"Bmw", "Ford"}
2) String → String greet = "Hello";
3) etc.

Identifiers

↳ Identifiers are the names given to class method, interface.

rules for naming Identifiers

1) valid char : lowercase, uppercase, digit, _, $
2) start with : letter, _, $. It cannot start with digit
3) case sensitivity : car , Car, CAR (different)
4) Reserved keywords: (int, for, if, class) cannot be used

convention for naming

1) Camelcase : for class, method, variables eg : carName
2) meaningful : string cityName = "Agra";

valid
- age ✓
- _marks ✓
- $value ✓
- 8marks ✗
- class ✗
- my-name ✗
- @cityName ✗
- calculator ✓

int a = 10;
- int → Data type
- a → identifier
- 10 → literal

Note : Object is a instance of a class.

Lec 4

```
class A
{
    static int b;
    public static void main(String args[])
    {
        System.out.println("Aman");
        int a;
        a = 10;
        System.out.println(a);
        System.out.println(b);
    }
}
```

(static default value '0')

cmd

```
javac A.java
java A
Aman
10
0
```

Read Data from User → Scanner ✓ (for practise)
→ BufferedReader ✓ (for production)

```
import java.util.Scanner
class Sum
{
    public static void main (String [] args)
    {
        Scanner sc = new Scanner (System.in); // creating object
        System.out.println("Enter first Number");
        int a = sc.nextInt(); // 1st number
        System.out.println("Enter second number");
        int b = sc.nextInt(); // 2nd number
        int sum = a+b;
        System.out.println(sum);
    }
}
```

(used for object making; class; static variable)

cmd

```
javac sum.java
java sum
Enter first number
10
Enter second number
20
30
```

for String

```
Scanner sc = new Scanner(System.in)
String fname = sc.next();
```

for integer

```
int age = sc.nextInt();
```

for complete line

lec 5

```
Scanner sc = new Scanner(System.in)
String s1 = sc.next(); // Aman kumbhalwar
System.out.println(s1); // Aman
String s2 = sc.nextLine(); // Aman kumbhalwar
System.out.println(s2); // Aman kumbhalwar
```

Flow Control keyword

if, else : used for conditional branching in code execution.

switch, case, default : used for multi-branch decision making

while, do, for : used for loop control

break, continue : control loop execution flow

return : Returns a value from a method

Modifiers keywords :

public, private, protected : Specifies access levels.

static : Indicates a class method or variable that belongs to the class itself.

final : Represents an entity that cannot be changed.

abstract : Specifies that a class or method declaration is incomplete and must be implemented by subclass.

synchronized : Controls access to a block of code by multiple threads.

volatile : Indicates that a variable may be modified asynchronously.

transient : Specifies that a variable is not part of the persistent state of an object.

class Declaration keywords

class, interface, extends, implements : Used in defining classes and interfaces and their relationship.

Exception Handling keywords :

try, catch, finally : Used for handling exceptions

throw, throws : Used to throw exceptions explicitly and declare exceptions that a method might throw.

Lec 6

Object Creation and management keywords:

new: Creates a new instance of a class or array.
this: Refers to the current instance of the class.
super: Refers to the superclass constructor, variables or methods.

Miscellaneous keywords

package: Groups related classes and interfaces
import: Allows access to classes from other packages
enum: Defines a fixed set of constants.
instanceof: Checks if an object is an instance of a class or implements an interface.
native: Indicates that method is implemented in platform dependent code (often written in another language)

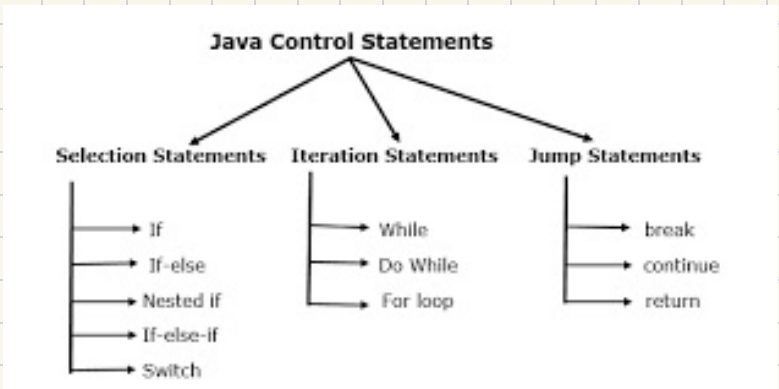
Operators and Control Statements

Operators

1) Arithmetic → +, -, *, /, % (→ remainder (modulo)), ++, --
2) Relational → ==, !=, >, <, >=, <=
3) logical → && (AND), || (OR), ! (NOT)
4) Assignment → =, +=, -=, *=, /=, %= → a = a % b ; a %= b
5) conditional (ternary operator) → Syntax: stm1 ? stm2 : stm3
6) instanceof →
```
String s1 = new String ("Aman"); // s1 → reference variable
boolean res = s1 instanceof String;
System.out.println(res); //true
```
7) new → for creating object and which stores in Heap Area.
8) Dot (.) operator → to access class members.

Control Statements

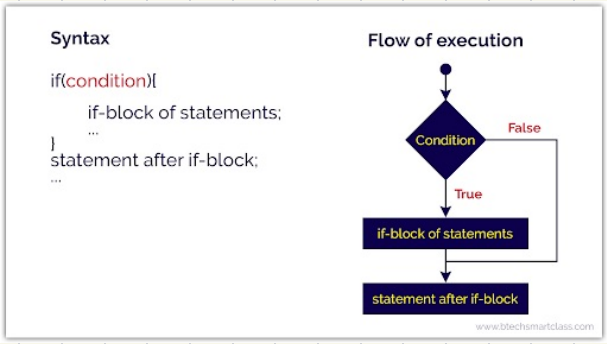
1) conditional
2) looping
3) Transfer



Selection statement

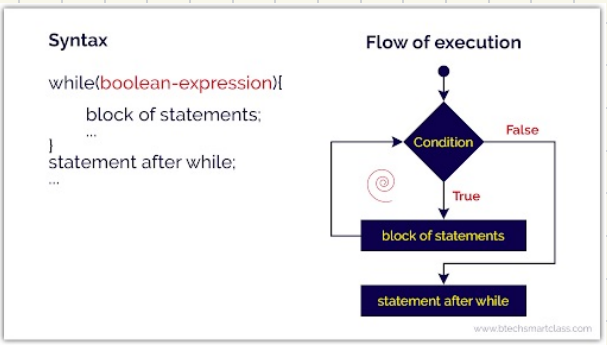
Lec 7

if
if else if
Switch



Iteration statement

while
do while
for



Jump statements

break
continue
return

```
while (testExpression) {
    // codes
    if (condition to break) {
        break;
    }
    // codes
}
```

```
do {
    // codes
    if (condition to break) {
        break;
    }
    // codes
}
while (testExpression);
```

```
for (init; testExpression; update) {
    // codes
    if (condition to break) {
        break;
    }
    // codes
}
```

Lec 8

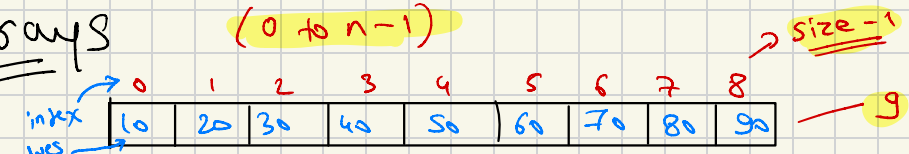
```
for (int i = 0; i <= 5; i++)
{
    ===
}
```
① ② ③ ④ ⑤

```
int rows = 3;
for (int i = 1; i <= rows; i++) {
    for (int j = 1; j <= i; j++) {
        System.out.print("*");
    }
    System.out.println();
}
```

output
```
*
**
***
```

Arrays

(0 to n-1)



1) Array is an object which contains element of similar data types
2) Array is a container that holds values of homogeneous (same type) type
3) Array are static in nature once declared remain fixed size.
4) Array are object that's why stored in Heap Area.

Syntax

```
int arr[] = new int[9]; //initialization & declaration
```

①
```
for (int i=0; i<9; i++)
{
    arr[i] = i; // arr[0] = 0, arr[1] = 1 ...
}
for (int a : arr)
{
    System.out.print(a)
}
```

output
0, 1, 2, 3, 4, 5, 6, 7, 8

```
int arr[] = {10, 20, 30, 40, 50};
```

②
```
for (int a : arr)
{
System.out.print(a);
}
```

// length
arr.length => 5

Output
10, 20, 30, 40, 50
(index: 0, 1, 2, 3, 4)

for changing
arr[2] = 100
(30 → 100)

Lec 9

① Sum

```
int arr[] = {10, 20, 30}
int sum = 0;
for (int i=0 ; i<arr.length; i++) {
    sum = sum + arr[i]
}
System.out.println(sum);
```

output : 60

② max

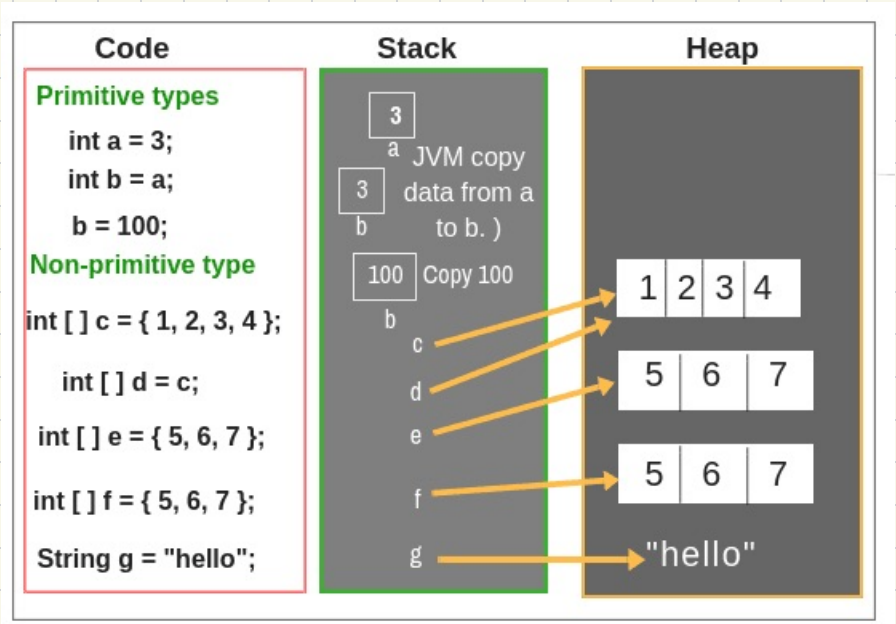
```
int arr[] = {8, 4, 5, 88, 44, 12, 3};
int max = arr[0];
for (int i=1 ; i< arr.length; i++)
{
  if (max < arr[i])
    {max = arr[i]
    }
}
System.out.println(max);
```

output
88

③ check

```
int arr[] = {1, 2, 18, 14, 15 }
int k = 18
boolean flag = false;
for (int i=0; i< arr.length; i++){
    if (arr[i] == k)
      { flag = true;
      }
}
System.out.println(flag);
```

output
true

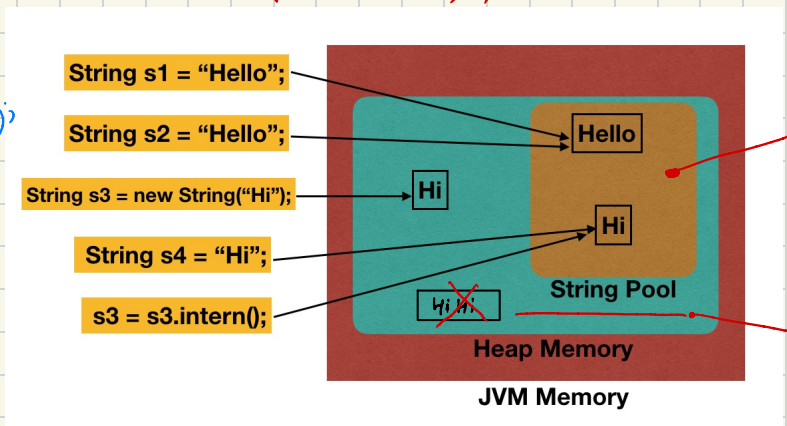


String

Lec 10

1) String is an object that represents sequence of character
   eg: "Aman"
2) In Java string represents by String class which is available in java.lang package.
3) String objects are immutable in nature → non-changeable
4) How to create string objects
   - by string literal → String res = "Aman"
     Note: String literal is always created in String constant Pool.
   - by new keyword
   - by converting character array into string.

```
char arr[] = {'a','b','c'};
String s = new String(arr);
System.out.println(s); // abc ✓
```



(s1.length());
5

Garbage collector don't access this area

Garbage collector access this area.

s1 == s2
true

```
System.out.println(s1.hashCode());
System.out.println(s2.hashCode());
```

Output: -714415749 ✓
-714415749 ✓ } Same

Note: This is not memory address. encoded value

Note: When we create string with the help of new keyword it stores in heap memory and String pool area but always point to heap memory. JVM create extra string in String pool for future purpose. String s3 = "Hi" (discarded)

```
String s = "Aman";
char arr[] = s.toCharArray();
System.out.println("to char array = " + arr)
```

output : to char array = [c@1b6d3586
↳ character array
↳ single dimension

```
System.out.println(Arrays.toString(arr));
```
output : ('A', 'm', 'a', 'n')

```
System.out.println(s.charAt(1));
```
output : m

```
String s1 = "hello";
String s2 = "hi";
int res = s1.compareTo(s2)
System.out.println(res)
```
output: -4

s1 = "hello"
s2 = "hello"
s1.compareTo(s2)
↓
0

```
//equals or ==
System.out.println(s1.equals(s2));
```
output true

```
String s = "Hello, Aman";
System.out.println(s.contains("Aman"));
```
output true

```
System.out.println(s.indexOf('a'));
```
output 2

```
String res = s.replace("Aman", "world");
System.out.println(res);
```
output Hello, world

```
String s1 = "hello world aman";
String res = s1.substring(0,7);
```
(0, n-1)
0 1 2 3 4 5 6
output hello w

StringBuffer

Lec 11

1) StringBuffer class is mutable → change allowed
2) It is similar to the string
3) If we want to lot of modification than we can go with StringBuffer
4) StringBuffer class is thread safe.

```
StringBuffer sb = new StringBuffer("Hello");
sb.append("world");
System.out.println(sb);
```
output : Helloworld

Note: String → original remain same
StringBuffer → original changed

```
//String
String s1 = "hello";
String s2 = s1.concat("world");
System.out.println(s1); //hello
System.out.println(s2); //helloworld
```

```
StringBuffer sb = new StringBuffer("Archi");
sb.insert(2,123);
System.out.println(sb); // Ar123chijain

sb.reverse();
System.out.println(sb); // niajihc321rA

sb.replace(6,11,"hello"); → (6 to n-1)
System.out.println(sb); // niajihhelloA

StringBuffer sb1 = new StringBuffer();
System.out.println(sb1.capacity()); // 16 (default)

StringBuffer sb1 = new StringBuffer("hello"); → length = 5
System.out.println(sb1.capacity()); // 16 + 5 = 21
```

StringBuilder thread safety → multiple thread not allowed

1) When we want mutable string without thread safety then StringBuilder will be used.
2) When we want a mutable string with thread safety then StringBuffer will be used.
3) When we want a immutable object then String should be used.

```
StringBuilder obj = new StringBuilder("welcome to");
obj.append("genei ashwani");
System.out.println(obj); //welcome to genei ashwani
```

Command Line Arguments

```
public class Command {
    public static void main(String[] args) {
        int a = Integer.parseInt(args[0]);
        System.out.println("a ="+a);
        int b = Integer.parseInt(args[1]);
        System.out.println("b ="+b);
        int c = Integer.parseInt(args[2]);
        System.out.println("c ="+c);
```

open cmd

```
javac Command.java
java Command 10 20 30
a = 10
b = 20
c = 30
```

Programming lang. are divided into 2 types :-

1) Procedure lang. → C, cobal, Pascal
2) object oriented lang. → Java, C#, Python

→ In Procedure lang. we will develop function and procedure. if we want to add more functionality then we need to develop more function there is no security.

→ If we want to develop a project using OOP lang. than we have to use classes and object.
  ↳ Blueprint ↳ Instance of a class

→ OOPS lang. are secure

→ the main adv. is code reusability.

OOPS principles

1) Encapsulation (Data hiding)
   ↳ is a used to combine our variable and method as a single entity.
   ↳ we can achieve Encapsulation by using classes
   ↳ class Demo {
     // variable
     // method
     }

2) Abstraction (eg: ATM)
   ↳ means hiding unnecessary data and providing only required data.
   ↳ we can achieve abstraction using interface and abstract class.
   ex: we will not bother about how laptop is working internally.

3) Polymorphism
   ↳ Exhibiting multiple behaviors based on situation is c/es Polymorphism
   ↳ There are two types of Polymorphism

4) Inheritance
   ↳ means creating new classes based on existing ones.
   ↳ acquires all the properties and behaviors of a parent class.

OOPS

lec 1