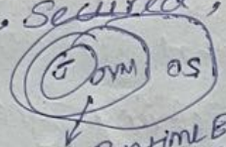
Java

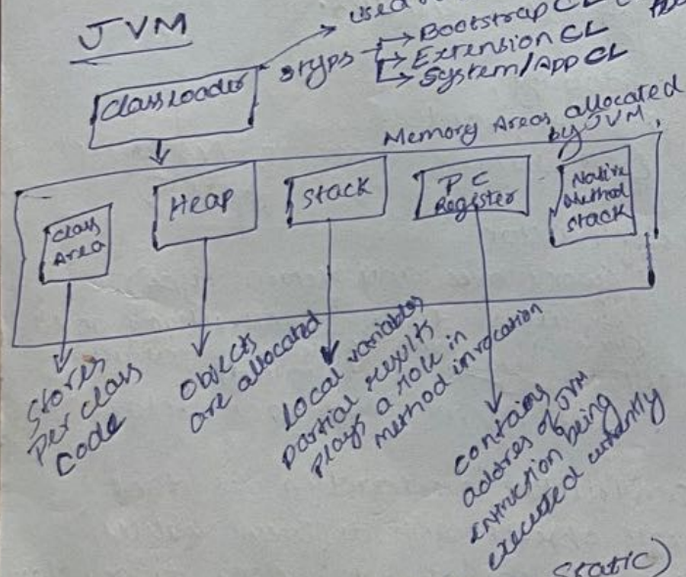
Features of Java
+Simple, OOPlg, Portable, Platform indipendent, Secured, Multithreaded



Valid Java main method signatures
1. psvm(String[] args)
2. psvm(String []args)
3. psvm(String args[])
4. psvm(String... args)
5. static public void main("")
6. ps final vm("s")
7. f psvm("")
8. f strictfp psvm(String[] args)

JVM → Loads code verifies, Provides RTE
JRE → It is a set of software Tools which JVM uses at run time.
JDK → JRE + Development tools.

JVM



Variables (Local, Instance, Static)
→ A local variable cannot be defined with static keyword
→ Memory allocation of static variable happen

Operators.

Unary → ++ -- ~ !

a=10
b=-10
~a o/p -11
~b o/p 9

Left shift
10<<2 10 * (2^2) ⇒ o/p = 40
10<<3 10 * (2^3)

Right Shift
(20>>3)
20/2^3 =

Java Keywords

enum: Java enum keyword is used to define fixed set of constants. Enum constructors are always private or default

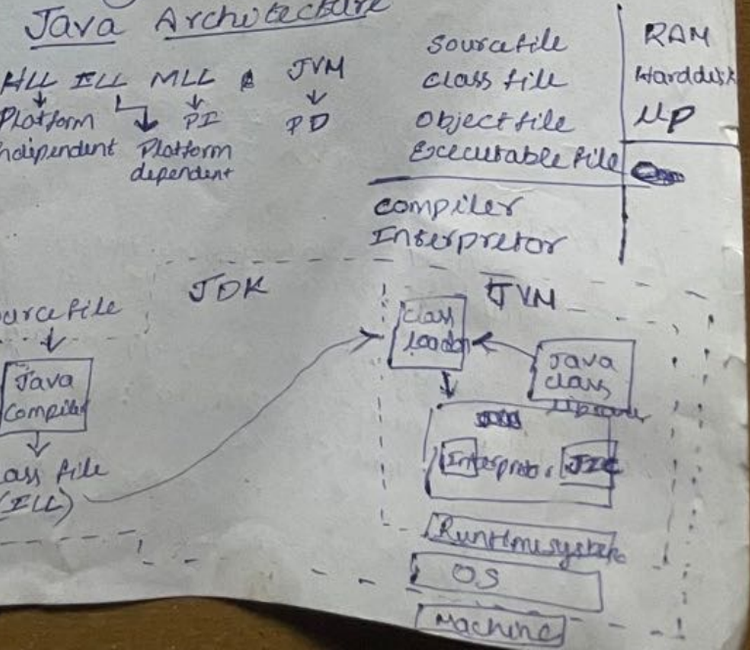
native: Java native keyword is used to specify that a method is implemented in native code using JNI

transient: used in serialization If you define any data member as transient, it will not be serialized.

volatile: Java volatile keyword is used to indicate that a variable may change asynchronously.

Java Architecture

HLL ILL MLL → JVM
Platform Indipendent → Platform dependent → PI → PD



OOPS vs Object based pgmg lg.

→ Object based programming language follows all the features of OOPs except Inheritance Javascript & VB script are examples of object based programming languages.

Data Types

byte (1) short (2) Int (4) Long (8) → Base 2 $(-2^{n-1}$ to $+2^{n-1}-1)$

char (2) → UNI code

Float (4) Double (8) → IEEE single & double precision format

Boolean → 1 bit

Int → 0, 0x, 0b (0 → zero)

octal, hexa decimal, binary.

→ All decimal nos are considered double by Java so prefix (float) or suffix 'f' when storing into a float variable.

Strings in Java

Heap

CP / NCP (new) without using New operator

Codespace
Static mem
Heap mem
Stack mem

→ mutable strings: if String class is used then

→ immutable strings: if String Buffer or Builder is used

S Buffer	S Builder
• Synchronized	• non synchronized
• best for Multi threaded apps	• Not for
• slow	• fast

Variables

$ & _ → only 2 special symbols allowed in Java.

Literals

_ → only 1 special symbol that too should be used in btw the literal numbers

Primitive Data types & wrapper classes

Integer a = new Integer(10);

↓ int short byte

→ wrapper classes.

Adv of wrapC → Pure OOPL

Disadv → slow execution.

→ Type casting Implicit & Explicit

→ Local variables
- stored in stack
- deallocated as soon as ctrl leaves method

Instance variables
- stored in heap
- object becomes garbage deallocated.

→ Value type & Reference type Assignment

→ Method Overloading. (Virtual Polymorphism)
- function call is resolved based on
  1. Number of Parameters
  2. Data types of the Parameters
  3. order of Parameters.

OOPS

1. Encapsulation
2. Inheritance
3. Polymorphism
4. Abstraction.

1. Encapsulation.

→ Declaring most imp component of an object by declaring it as private & giving controlled access to this private members by using setters & getters.

a) shadowing.

b) This keyword (it always points to the current object)

c) Constructor
- (doesn't have any return type)
- (executes during construction of object
- *** (the very 1st statement inside every constructor should be super() method)

d) super
- calls Parent class constructor

→ constructor overloading is done to create objects with different values where all the objects belong to same class

→ constructor chaining
- super()
- calling parent class constructor.

Local chaining
- this()
- calling same class constructor

Static Keyword

→ allocated memory on static space
→ allocated only once (irrespective of no of objects created)
→ class variables they are called.
(since they remain same for all classes)
→ Static Variables

→ Static blocks
- They execute only once irrespective of number of objects created to the class.
- static block can never be inside static method
(common sense all stuff is static why defining again static)

- any non static stuff (var, blocks, methods) cannot be accessed from static stuff.

| Non static blocks

Inheritance

→ reusability of code, reduce time taken for development, increase profitability of company.
- extends
- private members dont take part in Inheritance
- constructors also dnt take part in Inheritance they execute due to super()
- Multiple & Cyclic Inheritance is not possible.
- Multilevel is possible.

UML → Unified modelling language

→ Parent reference is always given to child objects
(this helps in overriding, polymorphism, Loose coupling)
→ Parent reference cannot be used to access specialized methods of the child.
(By performing down casting we can use specialized child methods)

Upcasting

→ Parent p = (Parent) new Child();

```
P{                C{
  add()             add()
}  o/p 3          }  o/p 4
```

p.add o/p 4

So Parent p = new Child(); is same as 1st object creation so there is no much need of upcasting in Java.

Down Casting

```
Child c = new Child();
Parent p = new Child();

P{                C{
  add()             add();
}                   sub();
                  }
p.sub(); //error

Child c = (Child) p;
c.sub(); ✓
```

Polymorphism

Compile Time	Run Time
method overloading	Based on the reference passed at Run time User illusion is True also called

Animal
↑
Deer | Monkey | Tiger

Forrest
allow(Animal ref)
{
}

Animal App {
main method() {
}
}

final Keyword

→ If used with class, the class cant be inherited or extended

→ If used on method, the method cannot be over ridden.

→ If used on variable, the value becomes constant.

Delegation Model → for "has-a" relationship

→ composed Objects (obj which gets destroyed once main obj is destroyed)

→ Aggregate objects. (obj which still exist)

- composed objects should be created inside the class of main object at the instance level.
- All aggregate objects should be created seperately & passed as the parameter to one of the methods present inside the class of the main object.

Eg: OS charger Mobile Testmain

Rules of method overriding

① ~~Child has to maintain same access specifier as that of the parent~~

② Child can retain the same access specifier or only assign public or loose access specifiers.

when extends keyword shld not be used.

③ Return types should be same or Co-variant

Eg: Co-variant

```
Animal {
}

Parent {
public Animal activate() {
return A
}
}

Tiger extends Animal {
}

Child extends Parent {
public Tiger activate() {
return T
}
}
```

Access Modifiers / Access Specifiers.

	within the same class	within the same package	within a different package & subclass	within a different package & unrelated classes
public	✓	✓	✓	✓
protected	✓	✓	✓	✗
default	✓	✓	✗	✗
private	✓	✗	✗	✗

Abstract Classes

→ Abstraction is the process of hiding the implementation details & showing only functionality to the user.

- abstract key coord is used to declare class & methods as abstract.

- Abstract class can have both abstract & non-abstract methods
- It cannot be instantiated.
- It can have constructors & static methods also.

Exceptions in Java.

→ Ducking → when called method does not handle the exception, then automatically the exception object will be thrown to its caller.

→ Rethrowing → Even though called method is handling the exception, informing the caller with throw keyword is known as Rethrowing.

→ try block can have multiple catch blocks.

```
try{
}
catch(ArithmeticException e)
{
}
catch(ArrayIndexOutOfBounds
      Exception e)
{
}
catch(Exception e)
{
}
finally
{
}
```

→ finally block gets executed irrespective of whether there is exception or no exception, irrespective of throw or return keyword.

```
Class Exception
{
  public String getMessage()
  {}
  Public String printStackTrace()
  {
  }
}
```

Steps for creating custom Exception.

① Create a custom exception object

② write a class for custom exception object which extend Exception class.

③ Override the getMessage() in the class of the custom exception object.

Use of printStackTrace()

→ In order to know where the problem is in the program we must use PRINTSTACKTRACE()

→ this method will print the sequence of function calls.

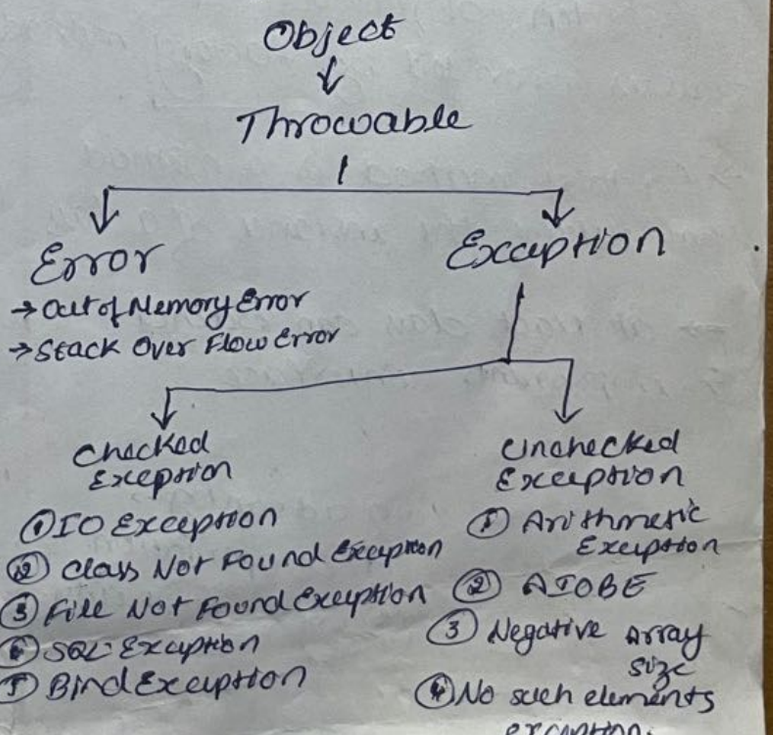
Types of Exceptions.

① Checked Exceptions.

→ Not handled by the JVM

② Unchecked Exception

→ Even if pgrmr doesn't handle JVM itself will handle.



- It can have final methods which will force the subclass not to change the method.
- If extending abstract class either provide implem for all abstract methods or declare the child class as abstract.

Eg:

```
abstract class Bank {
    abstract int getRateOfInterest();
}
class SBI extends Bank {
    int getRateOfInterest() {
        return 7;
    }
}
class main {
    psvm {
        Bank b = new SBI();
        SOP b.getROI();
    }
}
```

→ Implementation class will be hidden, object of implementation class is provided by factory method

→ factory method is a method that returns the instance of a class

→ abstract class can extend & implements interface

Two advantages:
① Encapsulation
② Code reusability

Interface

→ Interface represents connection between programming language & DBMS.

① Inside interface the methods are by default public & abstract.

② Object of interface cannot be created.

③ From Java8, → default & static methods can be present in Interface.

- default methods → To provide backward compatibility.

Static methods → using the interface name u can call it

④ From Java 9 → private methods can be present in Interface

⑤ All variables inside an interface are by default public static & final

can be done with abstract classes as well, but they will be having constructors, state & behaviours

Java9 private methods because →

```
public interface Food {
    default void bar() {
        jazz();
    }
    private void jazz() {
        SOP("");
    }
}
```

```
public interface Good {
    default void bar() {
        jazz();
    }
    private static void jazz() {
        SOP("");
    }
}
```

```
psvm {
① Food f = new CustomFood();
    f.bar();
② Good.bar();
}
```

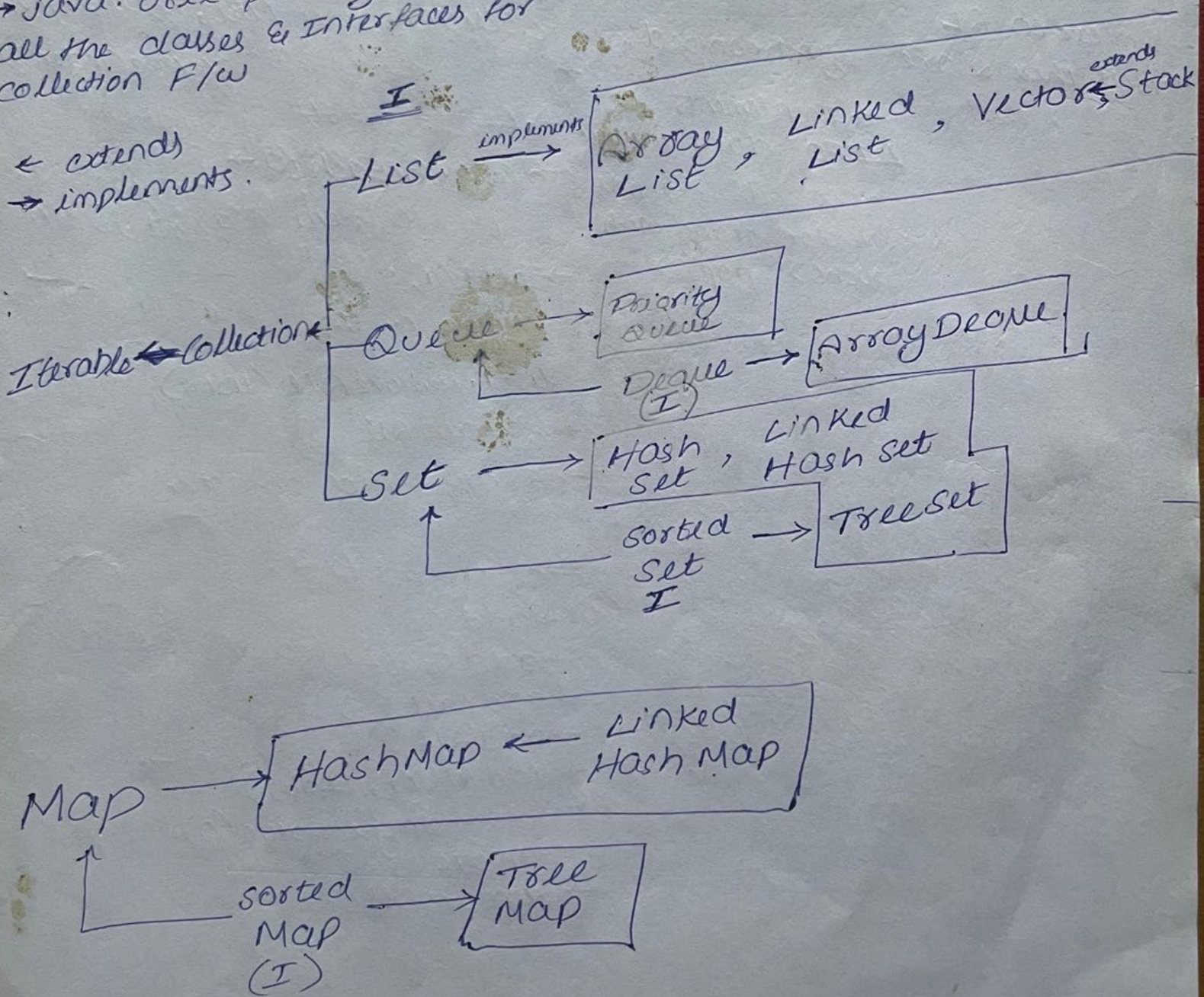
Collections Framework

→ Collection in Java is a f/w that provides an architecture to store & manipulate group of objects.

→ Can acheive all operations that you perform on data such as searching, sorting, insertion manipulation & deletion.

→ java.util package contains all the classes & Interfaces for collection F/w

← extends
→ implements.



Array List Linked List

→ maintain insertion order
→ Both are Not Synchronized so not thread safe.
→ LL, Manipulation is fast as no shifting is required.
→ Acts as both List & Queue
⇒ Duplicates can be stored

→ for (int i=0; i<list.size(); i++)
  SOP(list.get(i))
  set(i)
→ for (Integer i : list)
  SOP(i)
→ Iterator itr = list.iterator()
  while(itr.hasNext())
  SOP(itr.next())

→ Collections.sort(list)
  Collections.reverse(list)
  Collections.sort(list, Collections.reverseOrder())

Vector ← extends STACK.

→ Similar to AL
→ But these 2 are synchronized.

→ Implements LIFO data structure
→ has all methods of vector along with push() pop()

⇒ ~~Linked list~~
→ list.forEach(a → {
  System.o.(a);});
→ add(1, "")
  remove("")
  list1.removeAll(list2)
  .retainAll
→ Array list of objects

⇒ Linked List
→ add( )
→ addFirst( )
→ addLast( )
→ addAll
→ addAll(1, LLobj)
→ descendingIterator(0)
→ remove( )
→ removeAll( )

Hash Set

→ allows null value
→ non synchronized
→ unique elements only.
→ Insertion order not saved, all elements are arranged on the basis of Hashcode

Hash set	Array	sorted Array	Binary ST
Insert → O(1)	O(1)	O(n)	O(h) O(n)
Search (O(1))	O(n)	O(logn)	O(n) Ologn)
Delete O(1)	O(n)	O(n)	O(n)

→ Best

Values (24, 52, 91, 67, 48, 83)

Hash Function (Kmod10, Kmodn, Midsqr Folding Method)

where to store 24
Kmod 10
24 mod 10 = 4
Hashvalue = 4

where is 67
67 mod 10 = 7

Now 62 mod 10 = 2
But 2nd place is already full Here comes the concept of Collision

this was using Kmod 10 fn
There are different fns like Kmodn, Midsqr Folding method

	0
91	1
52	2
83	3
24	4
	5
	6
67	7
48	8
	9

Collision Resolving Techniques

- Chaining
- Open Addressing closed Hashing
  1. Linear Probing
  2. Quadratic Probing
  3. Double Hashing

→ Open Hashing Tech banake usi place ko attach kar do (24, 19, 32, 44)

0	24
1	19
2	32 → 44 → 58
3	44
4	
5	

Linked list
Kmod 6
→ Linear Probing.

→ Linear Probing
- First fill the space available bas.
- Instead of creating linked list we add it to the next available space.

→ Quadratic Probing
R + i^2 mod n.
Eg 30 → shld be placed in 0th position
- 0 + 1^2 mod 6 = 1
- 0 + 2^2 mod 6 = 4

but place is filled so u got to next place increasing i value to 1 but R + i^2 mod n gives you 1st place again it's filled you go to 2nd place increasing i value you get 4 & place is free u place it at that position

→ Double Hashing
- Using 2 fns to resolve collision.

Hash set methods

→ add
→ remove
→ set. addAll (set 1)
→ removeAll

Linked Hash Set

→ maintains insertion order
→ allows null elements
→ unique elements only.
→ non synchronized.

(methods remain same)

Tree Set

→ Elements are always stored in ascending order.
→ unique elements only.
→ does not allow null elements.
→ Treeset class maintains ascending order.

```
Treeset <String> ts = new Treeset <String> ();
ts.descendingSet();
ts.headSet("", true)
```
↖ Till this element from start it prints.

```
ts.tailSet("", false)
```
↳ Till this element from tail it prints

```
ts.subSet("A", false, "E", true)
```
From A to E excluding A it prints.

HS	LHS	TS
Null ✓	null ✓	null X
N Sync	N Sync	N Sync
Unq Ele only	Unq Ele only	Unq Ele only.
Insertion order X	Insertion order ✓	Ascending order ✓

Queue Interface.

FIFO → first in first out

when elements needs to be processed on priority not just by FIFO we have

Priority queue.

```
PriorityQueue <String> queue
        = new PriorityQueue <String>();
queue.add("")
SOP(queue.element());
SOP(queue.peak());
```
} o/p's the First element

iterator → same as previous List

Deque Interface.

LIFO

To implement Deque we have a class.

Array Deque

→ Unlike queue, we can add or remove elements from both sides.
→ Null elements are not allowed.
→ non synchronized.
→ Array Deque is faster than LL or stack

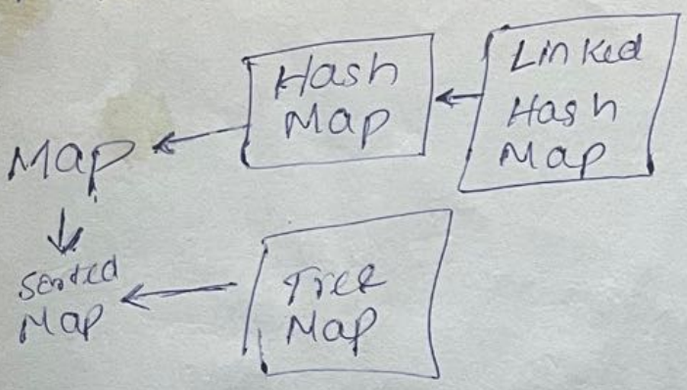
```
Deque <String> dq = new
        ArrayDeque <String> ();
```

→ dq.add("")
  .offer("")
} insert element → .offerFirst() → .offerLast()

→ dq.remove()
  .poll()
} removes the head of the dq.

→ .element() } retrieve
→ .peek()
  .peekFirst() } peekLast

Map Interface.



→ Key & Values → Map can have duplicate values but not duplicate keys.

→ Map can't be traversed convert to set using keySet() or entrySet()

```
Map<Integer, String> map =
    new HashMap<Integer, String>();
```

- map.put(K, "V")
- putAll(map obj)
- putIfAbsent(K, V)
- remove(K)
- get(K) → getKey() / getValue()
- containsKey
- containsValue

Traversing Map

①
```
Set set = map.entrySet();
Iterator itr = set.iterator();
while(itr.hasNext()){
    Map.Entry entry = (Map.Entry)
                itr.next();
    entry.getKey();
    entry.getValue();
}
```

②
```
for(Map.Entry m : map.entrySet())
{
    SOP(m.getKey()  m.getValue)
}
```

// ordering map

③
```
Map.entrySet()
 .stream()
 .sorted(Map.Entry.
        comparingByKey())
 .forEach(System.out::println);
```
o/p ascending order Key.

// Descending Order

④
```
map.entrySet()
 .stream()
 .sorted(Map.Entry.comparingByKey(
        Comparator.reverseOrder()))
 .forEach(SOP)
```

⑤
```
.comparingByValue(
    Comparator.reverseOrder())
```

HashMap

→ HashMap can have 1 null key & multiple null values.

→ HashMap is non-synchronized.

→ HashMap maintains no order.

Methods

```
hm.replace(102, "Gaurav")
    or
hm.replace(102, "Ajay", "Gaura")
hm.replaceAll((K,V) → "Ajay")
```

storing objects

```
Book b1 = new Book("  ");
    b2
    b3
map.put(1,b1)
        2, b2
        3, b3
```

```
for(Map.Entry<Integer, Book>
    entry : map.entrySet())
{ int key = entry.getKey();
  Book b = entry.getValue();
  SOP(key + "Details:");
} SOP(b.id+   b.na);
```

LinkedHashMap

→ contains unique elements.
→ may have 1 null key & multiple null values
→ Non synchronized
→ maintains insertion order.

same methods follow.

TreeMap

→ contains unque elements.
→ cannot have a null key but can have multiple null values.
→ Non synch
→ maintains ascending order.

methods
map.descendingMap()
map.headMap(102, true)
// till 102
.tailMap(102, true)
//
.subMap(100, false, 102, true)

HashTable

→ Unique elements only.
→ doesnt allow null key or value.
→ synchronized.
→ legacy class
→ slow compared to HM.

HM	HT
inherits Abstract Map class	inherits Dictionary class.

JDBC → Java Database Connectivity

→ Java API to connect & execute the query with the database.

→ API uses drivers to connect with the database.

- JDBC-ODBC Bridge Driver (Removed in Java 8)
- Native Driver
- N/w protocol Driver
- Thin Driver

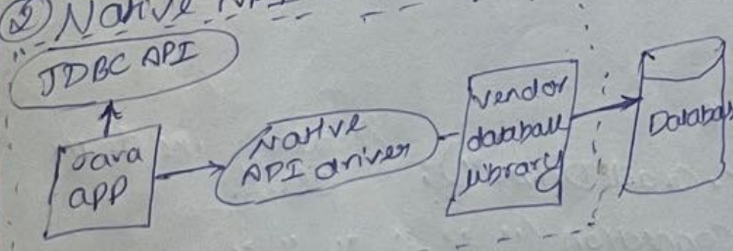
→ java.sql package contains classes & interfaces of JDBC API

Interfaces	Class
Driver Interface	DriverManager class
Connection "	Blob class
Statement	Clob class
PreparedStatement	Types class
CallableStatement	
ResultSet	
ResultSetMetaData	
DatabaseMetaData	
RowSet	

① JDBC-ODBC Bridge Driver

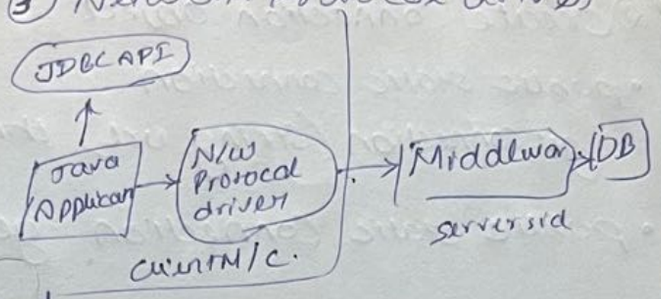
JDBC API calls or methods are converted to ODBC calls.

② Native API driver.



→ The driver converts JDBC method calls into Native calls of the database API.

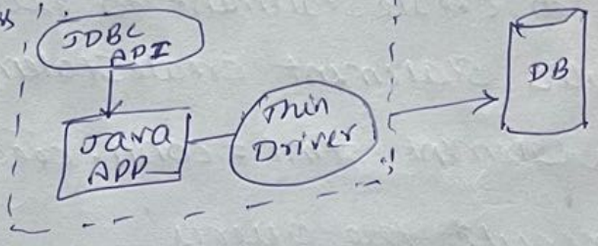
③ Network Protocol driver



④ Thin Driver

→ Thin Driver converts JDBC calls directly into the vendor-specific database protocol.

→ Drivers depend on Database.



Java DataBase Connectivity

5 Steps

→ Register the Driver class
→ Create connection
→ Create statement
→ Execute queries
→ Close connection.

① Register the Driver Class

From JDBC 4.0 explicitly registering the driver is optional. Instead put vendors Jar in classpath, & then JDBC driver manager can detect & load the driver automatically.

2) Create Connection object

- public static Connection getConnection(String url) throws SQLException
- public static Connection getConnection(String url, String name String password) throws SQLException.

Connection conn = DriverManager.getConnection("","","")

3) Create Statement object

- The createStatement() method of Connection interface is used to create Statement to execute queries on the database.

→ public Statement createStatement() throws SQLException

→ Statement stmt = conn.createStatement();

4) Execute the query

- The executeQuery() method of Statement interface is used to execute queries to the database.

→ public ResultSet executeQuery(String sql) throws SQLException

→ ResultSet rs = stmt.executeQuery("select * f");

```
while (rs.next()){
    SOP(rs.getInt(1) + " " + rs.getString(2));
}
```

5) close the connection object

→ public void close() throws SQLException

→ con.close();

Oracle DB eg. import java.sql.*

```
try{
→ Class.forName("oracle.jdbc.driver.OracleDriver"); //loading the driver class.
→ Connection con = DriverManager.getConnection("","","");
                                  ↓ jdbc:oracle:thin:@localhost
→ Statement stmt = conn.createStatement()
→ ResultSet rs = stmt.executeQuery("");
→ while (rs.next()){
    SOP (rs.getInt(1) + " " + )
con.close } catch(Exception e) {SOP(e);}
```

Connectivity with MySQL

```
try{
→ Class.forName("com.mysql.jdbc.Driver")
→ Connection con = DriverManager
        .getConnection("","","");
→ Statement stmt = con.createStatement();
→ ResultSet rs = stmt.executeQuery
        ("");
→ while(rs.next())
    { SOP(rs.getInt }
  con.close()
} catch (exception e) {SOP(e);}
```

Loading the MySqlconnector.jar file

2 ways

1) Paste the mysqlconnector.jar file in the jre/lib/ext folder
2) Set classpath

Driver Manager class.

1. public static synchronized void registerDriver(Driver driver)
2. public static synchronized void deregisterDriver(Driver driver)
3. public static Connection getConnection(String url) throws SQLException.
4. " url userName pwd.
5. public static Driver getDriver (String url)
6. public static int getLoginTimeout()
7. public static void setLoginTimeout (int sec)
8. public static Connection getConnection(String url, Properties prop) ← together user & pwd

Connection Interface

→ public Statement createStatement().
→ public void setAutoCommit(boolean status)
→ public void commit()
→ public void rollback()
→ public void close()

Statement Interface

→ public ResultSet executeQuery(String sql)
→ public int executeUpdate(String sql)
→ boolean execute(String sql)
→ public int[] executeBatch()

ResultSet Interface.

→ public boolean next()
→ public boolean previous()
    //moves cursor to 1 row previous
→ first()
→ last()
→ absolute(int row)
→ getInt(int columnIndex)
→ getInt(String columnName)
→ getString ''

Prepared Statement

→ public void setInt(int parameterIndex, int value)
    setFloat
    setString
    setDouble
→ public int executeUpdate();
→ public ResultSet executeQuery();

Batch Processing

Eg 1

```
→ class.forName(
→ Connection con = DM.getConnection(
→ con.setAutoCommit(false);
→ Statement stmt = con.createStatement();
→ stmt.addBatch("sql insert qry
  stmt.addBatch("");
→ stmt.executeBatch();
→ con.commit();
→ con.close();
```

Eg 2 try

```
Class.forName
Connection con =
PreparedStatement PS = con.prepare
        Statement("insert into
        user420 values(?,?,?)");

BufferedReader br =
  new BufferedReader(new Input
        StreamReader(System.in));

while(true){
  SOP("enter id")
  String s1 = br.readLine();
  int id = Integer.parseInt(s1);
  SOP("enter
  =
  PS.setInt(1, id)
        String(2, name)
        Int(3, salary)
PS.addBatch();

}
PS.executeBatch();

  catch()
```

JDBC RowSet Interface

```
JdbcRowSet rowset = RowSetProvider.newFactory().
        createJdbcRowSet();
rowSet.setUrl("  ")
rowSet.setUsername("  ")
    .setPassword("  ")
    .setCommand("select * from emp")
    .execute();
while(rowSet.next()){
        rowset.getString(1)
                "  (2)
                (3)
}
```

RowSet with Event Handling

```
① public void cursorMoved
    (RowSetEvent event);
② public void rowChanged
    (RowSetEvent event)
③   rowSetChanged("  ")
```

→ add 1 extra statement here

```
rowset.addRowSetListener
    (new MyListener());
class MyListener implements
        RowSetListener {

  override all the above
    3 methods
}
```

check the o/p its awesome.