public void moveToInsertRow()

Ex° rs.moveToInsertRow();

Step③: Insert new record data temporarily in resultset object.

To achieve this we have to use the following method.

public void updateXxx(int columnNum, xxx value)

where xxx may be byte, short, int ---

Ex: rs.updateInt(1,111);
rs.updateString(2,"aaa");
rs.updateFloat(3,5000.04);

Step④: Make temporary insertion as permanent insertion in updatable ResultSet object, as well as in database table. To achieve this we have to use the following method from ResultSet.

public void insertRow()

Ex: rs.insertRow();

- The following example demonstrates how to insert no. of records onto the database table through updatable resultset object.

Before database

```
import java.sql.*;
import java.util.*;
import sun.jdbc.odbc.*;
public class JdbcApp16
{
    public static void main(String[] args)
                                throws Exception
    {
DriverManager.registerDriver( new JdbcOdbc
                                Driver());
Connection con = DriverManager.
    getConnection("jdbc:odbc:nag", "system",
                                "durga");
Statement st = con.createStatement
(ResultSet.TYPE_SCROLL_SENSITIVE,
 ResultSet.CONCUR_UPDATABLE);
```

```
ResultSet rs = st.executeQuery("select * from emp1");
Scanner sc = new Scanner (System.in);
rs.moveToInsertRow();
while(true)
{
System.out.println("enter employee number");
int eno = sc.nextInt();
SOPrint("Employee name--");
String ename = sc.next();
S.O.println("enter employee salary:");
float esal = sc.nextFloat();
rs.updateInt(1,eno);
rs.updateString(2,ename);
rs.updateFloat(3,esal);
rs.insertRow();
SOP("employee inserted successfully");
SOP("one more record?.[y/n]");
String option = sc.next();
if(option.equals("n"))
break;
}//while
con.close();
}
}
```

8/8/17

01 > javac JdbcApp16.java

warning:

Expl: Load of driver is optional here. Just neglect that warning.

>javac Jdbc16.j

enter employee number:

enter employee

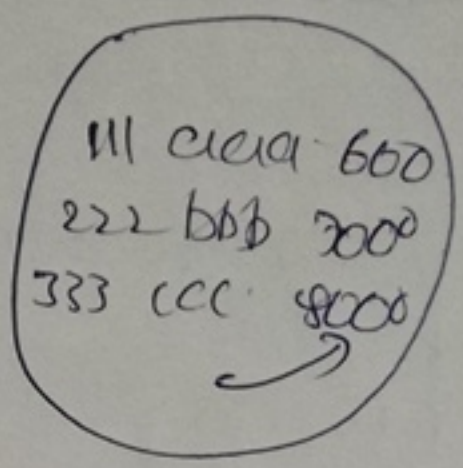
Notes In the above JDBC application, using moveToInsertRow() method is optional.
execute the above pgm once again without the stmt

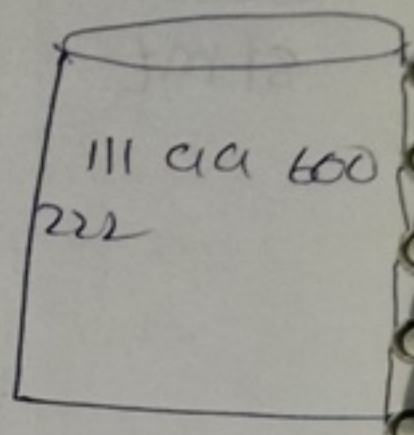
Suppose we have to use that stmt

— If we want to perform updations on database table through updatable resultset object then we have to use the following steps.

Step①: ~~[illegible]~~

```
while(rs.next())
{ float fsal=rs.get
                Float(3));
```





Create updatable resultset object

```
Statement st=con.createStatement(ResultSet.TYPE
SCROLL_SENSITIVE, ResultSet.CONCUR_UPDATABLE)
```

```
ResultSet rs=st.execute Query("select *
                              from emp(1);
```

Step②: Provide updations temporarily on updatable resultset object.

To perform updations on resultset object temporarily we have to use the following method from resultset.

```
public void updateXxx(int column_num,
                      xxx value)
```

where xxx may be byte, short, int, ———

Step③: make the temporary updations as permanent updations in resultset object to database table

— To achieve this we have to use the following method from ResultSet

public void updateRow()

else rs.updateRow() ✓

— The following example demonstrates how to perform operations on database table through updatable resultset object.

```
import java.sql.*;
public class JdbcApp17
{
  p s v m(String[] args) throws Exception
  {
Connection con=DriverManager.getConnection
("jdbc:odbc:nag","System","durga");
Statement st=con.createStatement(ResultSet.
TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_
                                UPDATABLE);
ResultSet rs=st.executeQuery(
        "select * from emp");
  while(rs.next())
    {
      float sal=rs.getFloat(3);
        if(sal<1000)
        {
float newSal=sal+500;
  rs.updateFloat(3,newSal);
    rs.updateRow();
  }//if
  }//while
  con.close();
  }
}
```

- Note: Similarly, to perform delete operation on database table we have to use the following method:

```
public void deleteRow()
```

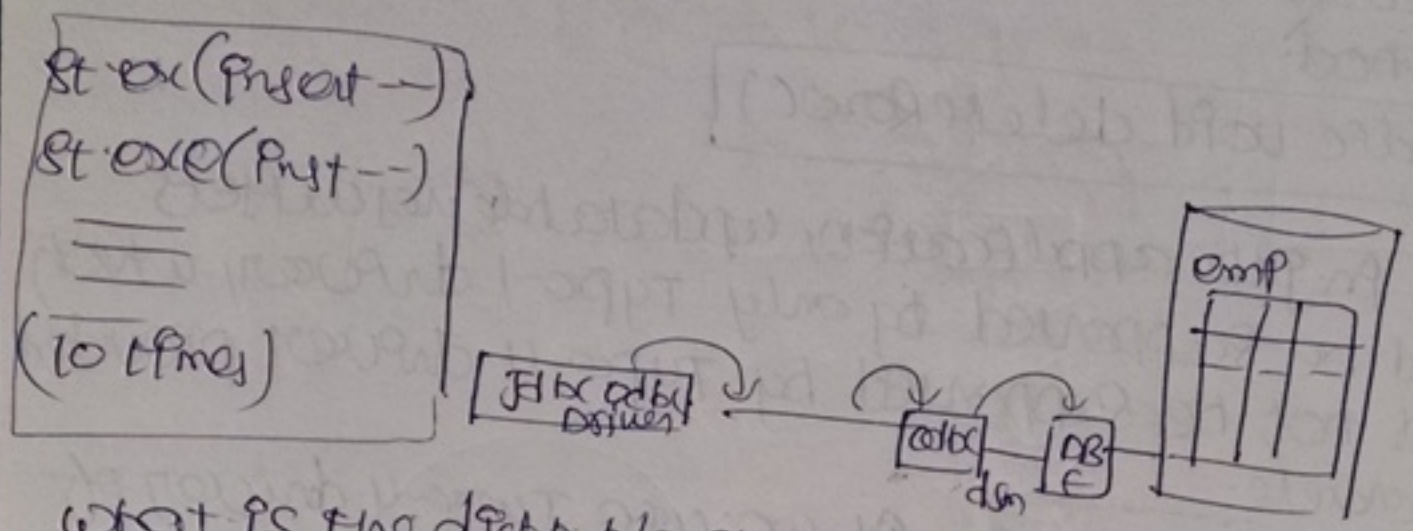
Note: In Java application, updatable resultsets could be supported by only Type-1 drivers, which could not be supported by Type-4 drivers provided by oracle.

- In the above pgm if we use Type-4 drivers of oracle

What will be o/p? while by Prog

Date: 11th May, 2012 Fri

prepared Statement:-



what is the diff b/w statement and prepared statement?

In jdbc applications, when we submit an sql query to the database engine then database engine will execute that sql query by performing query tokenization, query parsing, query optimization and query execution [query processing]

— In jdbc applications we will use Statement, when we have a requirement to execute independent sql queries.

— In jdbc applications, when we have a requirement to execute the same sql query in the next sequence where to improve the performance of jdbc appln we will use preparedStatement,

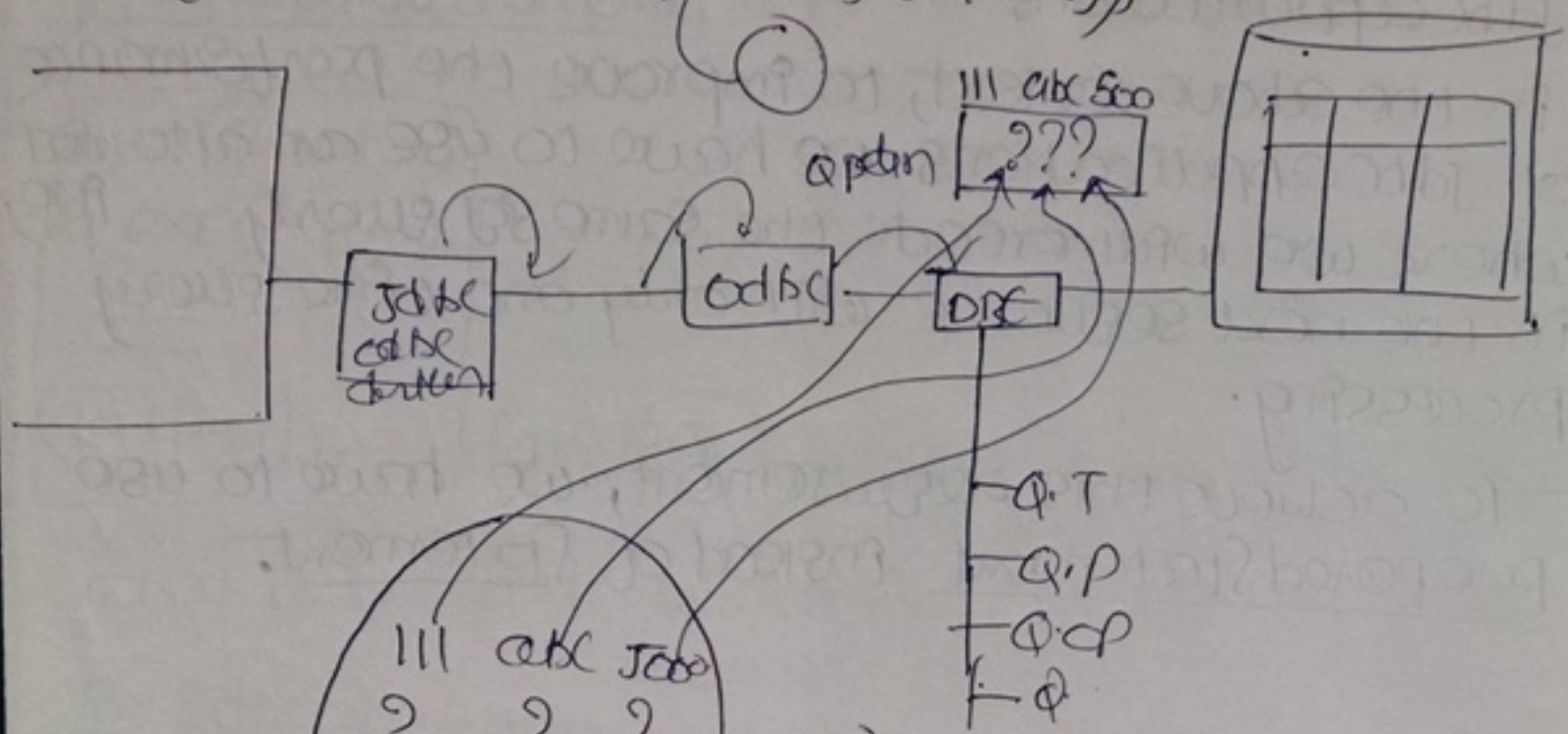
— For the above requirement if we use Statement then Database engine will perform query processing every time execution of the same sql query without having variation from one time to another time.

This approach will reduce the performance of JdbC applications.

In the above context, to improve the performance of JdbC applications we have to use an alternat where we will create the same SQl query five, in the next sequence with only one time query processing.

— To achieve this requirement, we have to use <u>preparedStatement</u> instead of <u>Statement</u>.

④ preparedStatement pst = con.prepareStatement ("insert into emp values (?,?,?)");



- If we want to use prepared statement in our jdbc applications then we have to use the following steps:

step①: create prepared Statement object with generalized SQL query format.

To create preparedStatement object we have to use the following method from Connection.

public preparedStatement prepareStatement (String query format) throws SQL Exception.

eg: preparedStatement pst = con.prepareStatement
("insert into empl values(?,?,?)");

— when JVM encounter the above instruction then JVM will pickup the SQL query format, send to database engine, where database engine will process query format by performing query processing and prepare a buffer with the positional parameters, called as queryplan, as a result preparedStatement object will be created at java application with the same positional parameters.

steps: set values to the positional parameters as per the requirement.

— To set values to the positional parameters we will use the following method from prepared Statement.

public void setXxx(int paramIndex, xxx value)

where paramIndex will start from 1.
where xxx may be byte, short, int, --

eg: pst.setInt(1, 111);
pst.setString(2, "aaa");
pst.setString(3, 5000.0f);

When JVM encounters the above instruction, JVM will set the specified parameter values to the respective positional parameters, here the provided parameters will be transferred to the positional parameters available in Query plan.

Step③: make database engine to pickup the values from Query plan and to perform the respective database operation as per the query format

- To achieve this requirement we have to use the following method

① If the SQL query format belongs to selection group then we have to use the following method

```
public ResultSet executeQuery() throws
                                SQLException.
```

Ex:
```
ResultSet rs = pst.executeQuery();
```

- If we provided SQL query format belongs to update or group then we have to use the following method

```
public int executeUpdate() throws SQLException
```

Ex: `int rowCount = pst.executeUpdate();`

- The following example demonstrates; how to insert numbers records on to the database table through prepared statement.

SQL> select * from product;

SQL> drop table product;
> commit;
> create table product (pid varchar2(5), pname varchar2(5));

psm: import java.sql.*;
import java.util.*;
public class JdbcAppl
{
public static void main(String[] args) throws Exception
{
Class.forName("oracle.jdbc.driver.OracleDriver");
Connection con = DriverManager.getConnection ("jdbc:oracle:thin:@localhost:1521:xe", "system", "durga");
preparedStatement pst = con.
prepareStatement("insert into product values (?,?,?)");
Scanner s = new Scanner(System.in);
while(true)
{
System.out.println("enter product id");
String pid = s.next();
sop("enter product name");
String pname = s.next();

```
sop("enter product cost");
  int pcost = s.nextInt();
 pst.setString(1, pid);
 pst.setString(2, pname);
 pst.setInt(3, pcost);
  pst.executeUpdate();
  s.o.p("product inserted successfully");
so.p("one more record [y]n]");
String option = s.next();
 if(option.equals("n"))
     break;
  }
}con.close();
}
```

the
....
are
are
update
e.......
e()
are
.....
in
prep.
stat

→ The following example demonstrates how to update database table through preparedStatement-

```
import java.sql.*;
import java.io.*;
public class JdbcApp19
    {
      p s v m(S[] args) throws Exception
       {
Class.forName("oracle.jdbc.driver.OracleDriver");
Connection con = DriverManager.getConnection
("jdbc:oracle:thin:@localhost:1521:xe", "system",
"durga");
preparedStatement pst = con.prepareStatement
("update product set pcost=pcost+? where
    pcost<?");
      pst.setInt(1,50);
      pst.setInt(2,1000);
int rowCount = pst.executeUpdate();
 sop(" Records updated "+rowCount);
        con.close();
    }
}
```

O/P

— The following example demonstrates how to fetch the data from database table through prepared Statement.

```
import java.sql.*;
public class JdbcApp20
  { p s v m(S[] args) throws Exception
    {
Class.forName("oracle.jdbc.driver.OracleDriver");
Connection Con = DriverManager.getConnection
("jdbc:odbc:thin:@localhost:1521:xe", "system",
                                    "durga");
preparedStatement pst = Con.
  prepareStatement("select * from student where
    smarks<?");
      pst.setInt(1, 90);
```

```
ResultSet rs = pst.executeQuery();
S.O.P("SID SNAM SMARKS");
S.O.P("_____");
while(rs.next())
{
S.O.P(rs.getString(1)+"  "+rs.getString(2)+"  "+
                    rs.getFloat(3));
}
con.close();
}
}
```

Callable Statement:

Stored procedures and functions:- In general in database applications, we often use normal SQL queries to perform the database operations.

- In database applications, SQL queries are sufficient to implement simple database logic.
- To implement complex application requirements SQL queries are not sufficient, where we have to use stored procedures and functions provided by PL-SQL.

What is the difference b/w stored procedures and stored functions?

Ans: Stored procedure is a block of instructions to represent a particular action, available at database and it will not use return statement to return a value.

```
Create or replace procedure procedure_Name
([param-list]) as
        == global variables
BEGIN
        == DB logic
END procedure_Name;
```

To save procedure SQL> /↵

- Stored function is a block of instructions want at database to represent a particular action and it will use return statement to return value.

```
create or replace function function_Name
      ([Param-List]) return datatype
as
== Global declarations
BEGIN
== DB logic
return value;
 END function-name;
```

To save function SQL> /↵

— From the above syntaxes, we need to provide parameters to stored procedures and functions there are three types of parameters

① IN type parameter.
② OUT type
③

IN parameter:- It is a parameter, it will get value from procedure call and function call and it will provide that value throughout the procedure body and function body.

Ex: Syntax: | var IN datatype |

Ex: eno IN number.

OUT type parameter: It is a parameter, it will get the value from procedure body and functional body and it will send that value to the respective procedure call or functional call.

| var OUT datatype |

eg: esal OUT number.

INOUT type parameter: It is a parameter acting as both IN type parameter and OUT type parameter

Syntax: [var INOUT datatype]

eg: eaddr INOUT varchar2(10)

Conclusion

① If jdbc applications if we want to access stored procedures and functions available at database from java applications then we have to use callable statement.

— If we want to access stored procedure or function available at database from java application by using callable statement then we have to use the following steps.

Date: 12th May, 2012 Sat

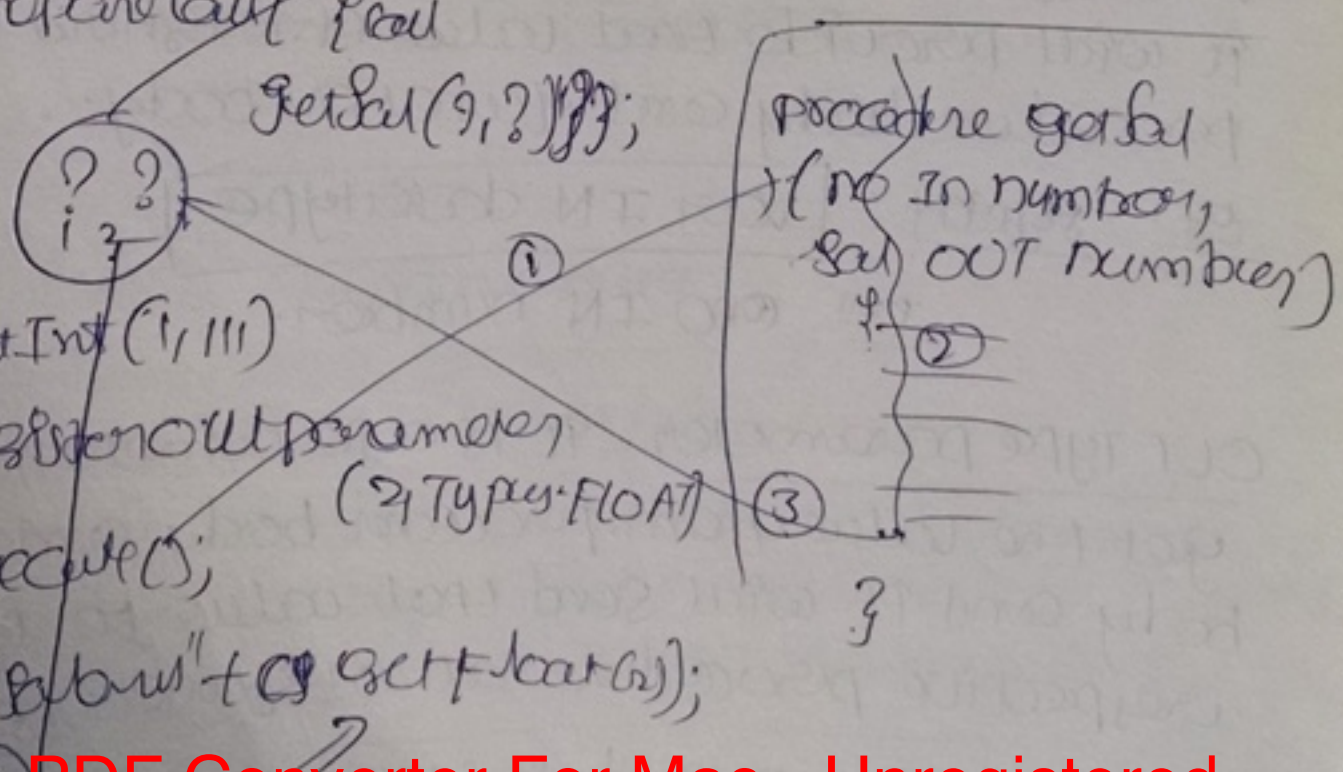
① CallableStatement st = Con.prepareCall("{call getSal(?,?)}");

② cst.setInt(1,111)

③ cst.registerOutParameter(2,Types.FLOAT)

④ cst.execute();

⑤ sop("Salary"+cst.getFloat(2));



Step①: Get callable Statement Object.

- To get callable statement object, we have to use the following method from Connection

public CallableStatement prepareCall(String call_Fmt) throws SQLException

eg: CallableStatement st = Con.prepareCall ("{call getSal(?,?)}");

when JVM encounters the above instruction, JVM will pickup procedure call, send to database engine, where database engine will process that procedure call.

As a result callableStatement object will be created, at Java application.

Step②: In a callable Statement object if we have IN type parameters then set values to IN type parameters.

To set values to IN type parameters, we have to use the following method from CallableStatement

public void setXxx(Int param_index, xxx value)

where xxx may be
bytes, short, Int - —

Eg: cs.setInt(1, 111);

Step③: In callable Statement object if we have OUT type parameters then register OUT type parameters with a particular data type.

— To register OUT type parameters, we will use the following method

public void registerOutparameter(int paramIndex, int type)

— where type may be constants like BYTE, SHORT, INTEGER, FLOAT — from Types class

Eg: cst.registerOutparameter(3, Types.FLOAT)

Step④: Execute the respective Stored procedure or function.

To access the stored procedure or function with the procedure call represented by CallableStatement object, we have to use the following method

public void execute() throws SQLException

Eg: cst.execute();

Step⑤: get the values from OUT type parameters.

— After executing the respective procedure or function, respective values will be stored in OUT type parameters in callable Statement object from stored procedure or function.

— To access the OUT type parameter values, we have to use the following method.

public xxxx getXxxx(int paramIndex)

where xxx may be byte, short, int, ---
ex: float sal = cst.getfloat(1);

— The following example demonstrates how to create a procedure at database and how to access it from java application.

edit in notepad

```
create or replace procedure getSal(no IN number,
sal OUT number)
as
BEGIN
select esal into sal from empl where eno=no;
END getSal;
/
```

↳ copy and paste on the sql command prompt.

— pgm:

```
{
Class.forName("oracle.jdbc.driver.OracleDriver");
Connection con = DriverManager.getConnection
("jdbc:odbc:thin:@localhost:1521:xe", "system",
                                    "durga");
CallableStatement cst =
  con.prepareCall("{call getSal(?,?)}");
    setInt(1, 111);
cst.registerOutparameter(2, Types.FLOAT);
    cst.execute();
```

- The following example demonstrates how to create a function at database and how to access into the java application.

```
create or replace function getAvg(no1 IN
number, no2 IN number) return number
as
sal1 number;
sal2 number;
BEGIN
select esal into sal1 from emp1 where eno=no;
select esal into sal2 from emp2 where eno=no2;
return (sal1+sal2)/2;
END getAVG;
```

```
import java.sql.*;
public class JdbcApp22
{ public static void main(String[] a1) throws Exception
{
Class.forName("oracle.jdbc.driver.OracleDriver");
Connection con=DriverManager.getConnection
("jdbc:oracle:thin@localhost:1521:xe", "System",
"durga");
CallableStatement cst=
con.prepareCall("{?=call getAvg(?,?)}")
```

① ②

↑ out type. / in type

```
cst.setInt(2,222);
cst.setInt(3,333);
cst.registerOutParameter(1,Types.FLOAT);
cst.execute();
```

```
SOP("Avg Salary -" + cst.getFloat(1));
Con.close()
}
}
```

O/P

— In general in stored procedures and functions, we will use the data types like number, varchar and so on; to represent a number of a string.
— As per the application requirements, we need to return all the records of a particular table from the procedures and functions, to achieve this oracle provided data types number, varchar and so on -- are not sufficient; we have to use Cursor types.
— Cursor is a variable, it will create an execution environment in database applications to send out bulk of data.
— To achieve the above requirements, we will use a predefined cursor that is SYS_REFCURSOR
— If we want to use cursor type variables in stored procedures and functions then we have to declare a variable as cursor type and we have to open it before the respective SQL query.

```
(OUT));
```

```
create or replace procedure
                    getproduct(pds OUT
                    SYS_REFCURSOR, cost IN number)
as
BEGIN
  open pds for
  select * from product where pcost < cost;
  END getproduct;
```

— If we want to use cursor type as parameter to

— In general in stored procedures and functions we will use the data types like number, varchar and so on; to represent a number or string.

— As per the application requirements, we need to return all the records of a particular table from the procedures and functions, to achieve this oracle provided data types number, varchar and so on -- are not sufficient; we have to use Cursor types.

— Cursor is a variable, it will create an execution environment in database applications to send out bulk of data.

— To achieve the above requirement, we will use a predefined cursor that is SYS_REFCURSOR

— If we want to use Cursor type variables in stored procedures and functions then we have to declare a variable as Cursor type and we have to open it before the respective SQL query.

Ex: create or replace procedure getproduct (pds OUT SYS_REFCURSOR, cost IN number)
as
BEGIN
open pds for
select * from product where pcost < cost;
END getproduct;

— If we want to use Cursor type as parameter to

Ex:2 create or replace function getproduct
(cost IN number) return SYS_REFCURSOR
as
product SYS_REFCURSOR
BEGIN
open product for
Select * from product where pcost <= cost;
return product;
END getproduct;

— If we access the above specified procedure or function with the cursor from a Jdbc application then the respective procedure or function will be executed and retrieve multiple number of records from database table represented by cursor type parameter and return that bulk of data into java application.

— At java application, the returned data will be stored in the form of resultset object.

— To register CURSOR type parameter, we have to use the following data type provided by Ojdbc6.jar file.

oracle.jdbc.OracleTypes.CURSOR

Ex: cst.registerOutParameter(1,
oracle.jdbc.OracleTypes.CURSOR)

- To get the generated ResultSet object, we have to use the following method from CallableStatement.

`public Object getObject(int paramIndex)`

Ex:

```
ResultSet rs=(ResultSet)st.getObject(1);
```

- The following example demonstrates how to create a procedure with cursor type and how to access it from java application.

```
create or replace procedure getEmps(emps
OUT SYS_REFCURSOR, sal IN number)
as
BEGIN
open emps for
select * from emp1 where esal<sal;
   END getEmps;
/
```

```
import java.sql.*;
public class JdbcApp23
    {  public static void main(String a[]) throws
                                        Exception
       {
Class.forName("oracle.jdbc.driver.OracleDriver");
Connection con=DriverManager.getConnection
("jdbc:oracle:thin:@localhost:1521:xe", "system",
CallableStatement cst=                  "durga");
  con.prepareCall("{call getEmps(?,?)}");
    cst.setFloat(2,1000.0f);
```

```
cst.registerOutParameter(1, oracle.jdbc.
OracleTypes.CURSOR);
        cst.execute();
ResultSet rs=(ResultSet) cst.getObject(1);
 sop("ENO ENAME SAL");
 sop("-------------------");
    while(rs.next())
     {
      sop(rs.getInt(1)+"  "+rs.getString(2)+
                  "  "+rs.getFloat(3));
     }
    con.close();
  }
}
```

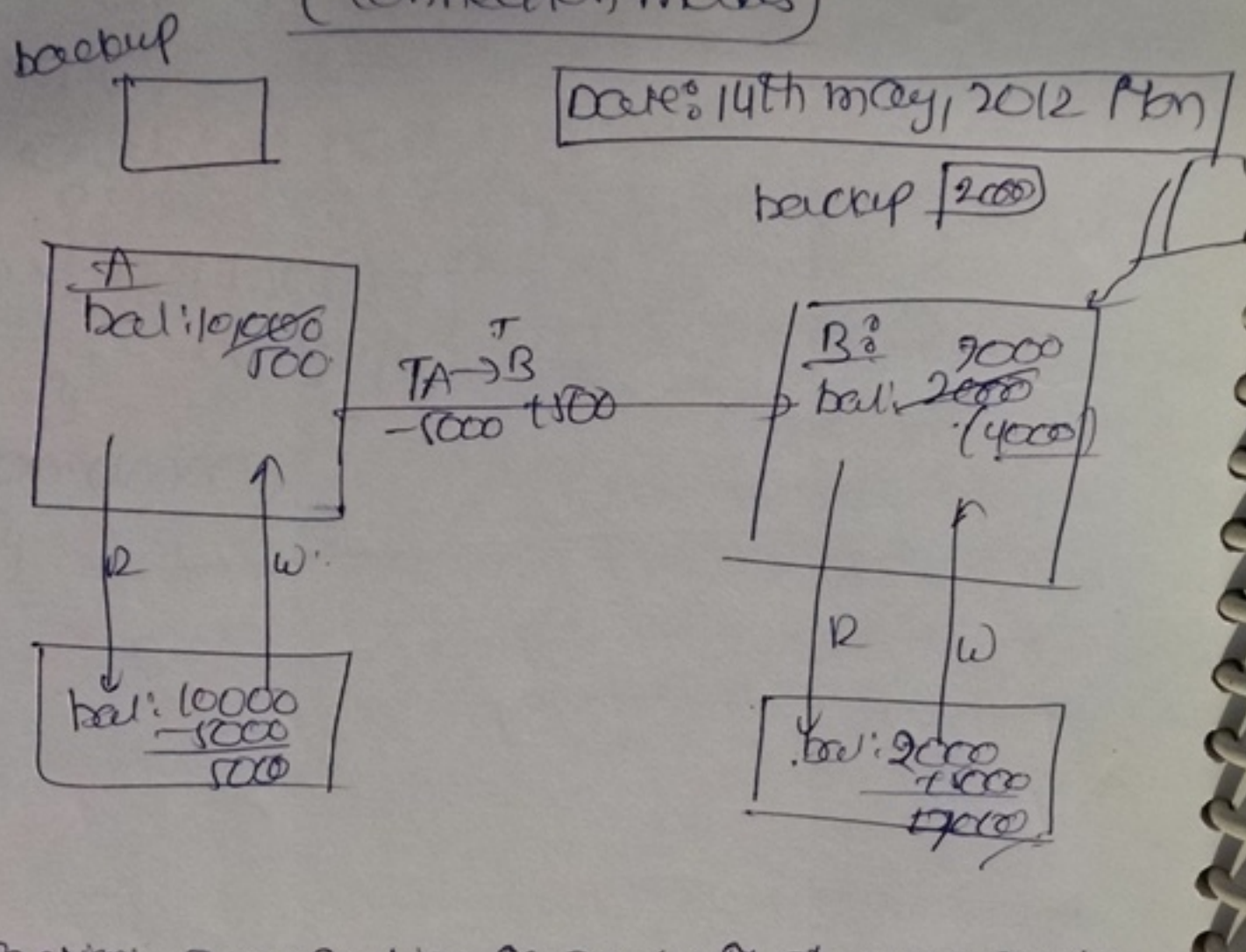
- The following example demonstrates how to prepare a function which return cursor type and how to access it from java application.

```
create or replace function getEmployees(sal
IN number) return SYS_REFCURSOR
 as
employees SYS_REFCURSOR;
 BEGIN
open employees for
select * from empl where esal<sal;
 return employees;
 END getEmployees;
 /
```

```
import java.sql.*;
public class JdbcApp24
    { public static void main(String[] args)
                throws exception
      {
Class.forName("oracle.jdbc.driver.OracleDriver");
Connection con=DriverManager.getConnection
("jdbc:oracle:thin:@localhost:1521:xe", "System",
                                    "durga");
CallableStatement cst=
con.prepareCall("{?=call getEmployees(?)}");
    cst.setFloat(2,10000.00);
cst.registerOutParameter(1,oracle.jdbc.OracleTypes.
                                    CURSOR);
cst.execute();
```

```
ResultSet rs = (ResultSet) cst.getObject(1);
    SOP("EMP ENAME SAL");
    SOP("-------------------");
    while(rs.next())
    {
SOP(rs.getInt(1)+"  "+rs.getString(2)+"  "+
            rs.getFloat(3));
    }
    con.close();
  }
}
```

Transaction Management
(connection modes)



Transaction: Transaction is an unit of work performed by front end application or backend system.

e.g. ① deposit some amount in an account.
② withdraw some amount from account
③ transfer some amount from one account to another account.

— In database applications, every transaction must satisfy the following four properties.

① Atomicity
②
③
④

Atomicity: In general in any transaction, we can perform one or more number of operations as per the requirements.

— In transactions, either perform all the operations or perform none operations is called as atomicity property.

— If we perform all the operations successfully in a transaction then the state of the transaction is success.

— If we perform none operations in any transaction then the state of the transaction is failure.

— While performing a transaction if we have any exceptions then we should not perform commit operation; where we should perform rollback operation to get previous instance of the database in order to preserve atomicity property.

Consistency:-

- The above property of the transaction is called as consistency, it may eliminate wrong results in transactions.

before after

Isolation: In general in database applications, if more than one transaction is acting on the same data item then that transactions are called as concurrent transactions.

The process of executing more than one transaction on the same data item is called as Transactions for while executing concurrent Concurrency. transactions, one transaction execution should not affect to another transaction execution, This nature of the transaction is called as isolation.

- To overcome transactions concurrency problems or to preserve transactions isolation property, we will use the transactions concurrency control mechanisms like locking mechanism. defining isolation methods over the data items and so on —.

Durability: After committing the transaction if we have any catastrophic failures then after the system all the modifications what we performed on database during the respective transaction must be preserved; this nature of the transaction is called as Durability.

≡ If we want to manage transactions in jdbc applications, we have to satisfy all the above transaction properties.

— In jdbc applications when we establish jdbc connection then automatically connection will be in auto commit mode that is in default mode

— In connection auto commit mode, when we submit an SQL query connection will pickup that SQL query, transfer to that database engine and make the DBE to execute that SQL query and store the results on database permanently.

- The above auto commit nature of connection may violate transactions atomicity property.
- To preserve transactions atomicity property in jdbc applications, we have to change the auto commit mode.

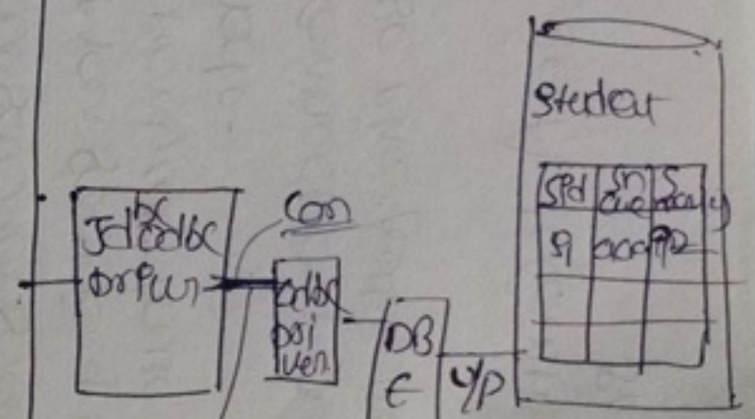
To eliminate connections auto commit mode, we have to use the following method from connection.

public void setAutoCommit(boolean b) throws SQLException

Sid	Sname	Smarks
1	aaa	72
S2	bbb	65
S3	ccc	92

Add

```
try
{
st.execUp("insert into stu values('S1',
                                  'aaa',72);
st.executeUpdate("insert
                  into student values
                  ('S2','bbb',65);
st.execute Update("insert into student
                   values('S3','ccc',92);
    Con.Commit();
}
catch(Exception e)
{
    Con.rollback();
}
```



If b == true then the connection will be in auto commit mode.

If b == false then the connection will not be in auto commit mode.

Ex: con.setAutoCommit(false);

2→ If we use non-auto commit mode to the connection in JDBC applications then we have to perform either commit or rollback operations to complete the transactions.

— To perform commit and rollback operations we have to use the following methods from Connection.

```
public void commit() throws SQLException.
public void rollback() throws SQLException.
```

— The following example demonstrates how to achieve transactions atomicity property by changing connections autoCommit mode from a JDBC application.

```
import java.sql.*;
public class JdbcApp25
{ public static void main(String args[])
  { Connection con=null;
    try
    {
      Class.forName("oracle.jdbc.driver.OracleDriver");
      con=DriverManager.getConnection
      ("jdbc:oracle:thin:@localhost:1521:xe",
      "system","durga");
      con.setAutoCommit(false);
```

```
Statement st = con.createStatement();
    st.executeUpdate("insert into empl
values(555,'eee',6000)");
st.executeUpdate("insert into empl values(
    666,'fff',7000)");
st.executeUpdate("insert into empl values(
                                777,'ggg',8000)");
    con.commit();
}//try-1
catch(Exception e)
    {
        try
        {
            con.rollback();
            e.printStackTrace();
        }
        catch(Exception e)
            {
                e.printStackTrace();
            }//catch-2
    }//catch-1
    }
}
```

Concurrent Transactions:-

If we allow more than one transaction on a single data item then that transactions are called as concurrent transactions.

— In general in database applications, if we provide concurrent transactions then we are able to get the following problems.

1. Lost update
2. Dirty read problem
3. Non-repeatable
4. phantom read.

— To resolve the above problems in concurrent Transactions we need to use isolation levels.

To represent isolation levels Connection interface has provided the following constants.

```
public static final int TRANSACTION_NONE=0
public static final int TRANSACTION_READ_
                        UNCOMMITTED=1
public static final int TRASACTION_READ_
                        COMMITTED=2
public static final int TRANSACTION_REPEATABLE
                        =4;
public static final int TRANSACTION_
                        SERIALIZABLE=8;
```

2) where TRASANCTION_NONE constant will represent all the above problems.

TRANSACTION_SERIALIZATION constant is able to resolve lost update problem.

- TRANSACTION_READ_COMMITTED Isolation level is able to resolve lost update and dirty read problem.
- TRANSACTION_REPEATABLE_READ Constant is able to resolve lost update, dirty read, non-repeatable read problems.
- TRANSACTION_SERIALIZABLE Constant is able to resolve all the above problems.

2 → To set a particular Isolation level in our JDBC applications we will use the following method from Connection.

```
public void setTransactionIsolation(int
                                    isolation level)
```

2 → To get the specified Isolation level; we have to use the following method from Connection.

```
public int getTransactionIsolation()
```

The default Isolation level in JDBC application is TRANSACTION_READ_COMMITTED. That is (2)

Lost update problem

Date: 15th May, 2012 Tue

In database applications, Transaction t_1 may perform updations on a particular data item, before performing commit operation by transaction t_2 perform any updations and perform commit operations then the updations performed by Transaction t_1 are not perceived (lost)

t_1	t_2
Read(bal)	
bal = bal + 500;	
write(bal);	
	Read(bal)
	bal = bal + 500;
	write(bal);
	commit();
commit();	

(bal = 5000)

To overcome this problem we have to use TRANSACTION_READ_UNCOMMITTED Isolation level.

Con.setTransactionIsolation(Connection.TRANSACTION_READ_UNCOMMITTED);

② Dirty read problem: In a database application Transaction t_1 may perform updations on a particular data item, before performing rollback or commit operations by t_2 transaction perform read operation over the modified data then by the transaction t_1 perform rollback operation over its modifications then the read operation performed by Transaction (t_2)

Ps dirty read

bal=5000

T_1	T_2
Read(bal);	
bal=bal+500;	
write(bal);	
	Read(bal);
	bal=bal+500
	write(bal);
rollback();	
.	commit();

To resolve lost update and dirty read problems we have to use TRANSACTION-READ COMMITTED

Ans Con.setTransaction Isolation(Connection. TRANSACTION-READ COMMITTED);

2) Non repeatable read:

In database applications, as per the applica requirement transaction T_1 may perform the sequence of read operations, where transaction T_1 may expect same result at each and every read operation. In the above context if a transaction T_2 perform updations over the same region of data (read by transaction T_1's

In the next read operation performed by transaction T_1, if transaction T_2 modify updated data, not same data retrieved in the previous read operations by transaction T_1 then this problem is called as non-repeatable read

T_1	T_2
to read(-)	
to read(-)	
to(updated) read(-)	write(-)
to(updated) read(-)	write(-)
=	
=	
read(-)	

To resolve dirty update, dirty read and non-repeatable read problems we have to use TRANSACTION_REPEATABLE_READ Isolation level.

@ Con.setTransaction Isolation(Connection. TANSACTION_REPEATABLE_READ);

2 → phantom read problem :-

In database application, transaction t_1 may to perform a sequence of read operations as per the application requirements.

— In the above context, transaction t_1 may expect same number of records at each and every read operation.

While performing sequence of read operations by transaction t_1 if any transaction t_2 inserts new records in the same region of data retrieved by transaction t_1 then transaction t_1 may identify some extra records along with original records in the next read operation.

Here the extra records identified by transaction t_1 are called as phantom records and read operation performed by transaction t_1 with the phantom records

is called as phantom read.

	T_1	T_2
t0	Read(-)	
t0	Read(-)	
t4	Read(-)	write(-)
t8	Read(-)	write(-)
	≡	
t0	Read(-)	

To resolve all the concurrency problems connection interface has provided TRANSACTION_SERIALIZABLE isolation level

Eg: con.setTransaction Isolation(Connection. TRANSACTION_SERIALIZABLE)

Note: In enterprise applications it is not at all suggestible to use Jdbc provided transaction support, it is always suggestible to use the underlying application server provided JTA (Java Transaction Api) middleware service.

Savepoint :- Savepoint is a new concept introduced in jdbc 3.0 version. The main purpose of the Savepoint is to block a set of instructions result in the transaction commit() operations.

(Capital, small)

— To prepare a save point we have to use the following method from Connection

```
public Savepoint setSavepoint().
```

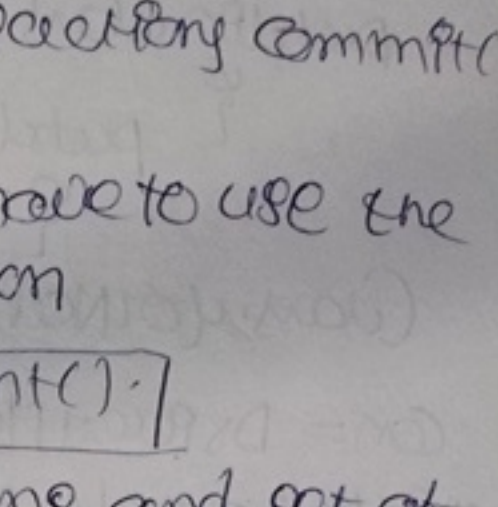
— To perform rollback operations and set of instructions execution with respect to a particular save point we have to use the following method from connection.

```
public void rollback(Savepoint sp)
```

— To release Savepoint, we will use the following method from Connection.

```
public void releaseSavepoint(Savepoint sp)
```

— In jdbc applications, Savepoint concept should be provided by Type4 driver provided by oracle, but not supported by Type1 driver ✓



— Type4 driver is able to provide Savepoint concept × capto for Savepoint() and rollback() method; not

releaseSavepoint() method

example

```
import java.sql.*;
public class JdbcApp26
    { public static void main(String a) throws
        { try{
Class.forName("oracle.jdbc.driver.OracleDriver");
Con= DriverManager.getConnection
("jdbc:oracle:thin:@localhost:1521:xe","System",
                                    "durga");
Con.setAutoCommit(false);
Statement st =Con.createStatement();
   st.executeUpdate("insert into student
                        values('s',1000,78)");
Savepoint sp=con.setSavepoint();
 st.executeUpdate("insert into student
        values('s3','ccc',80)");
(I) Con.rollback(sp);
 st.executeUpdate("insert into Student values
                        ('s4','ddd',89)");
        Con.commit();
    }
catch(Exception e)
        { try
            { con.rollback();
            }
                    { e.printStackTrace();
            }
```

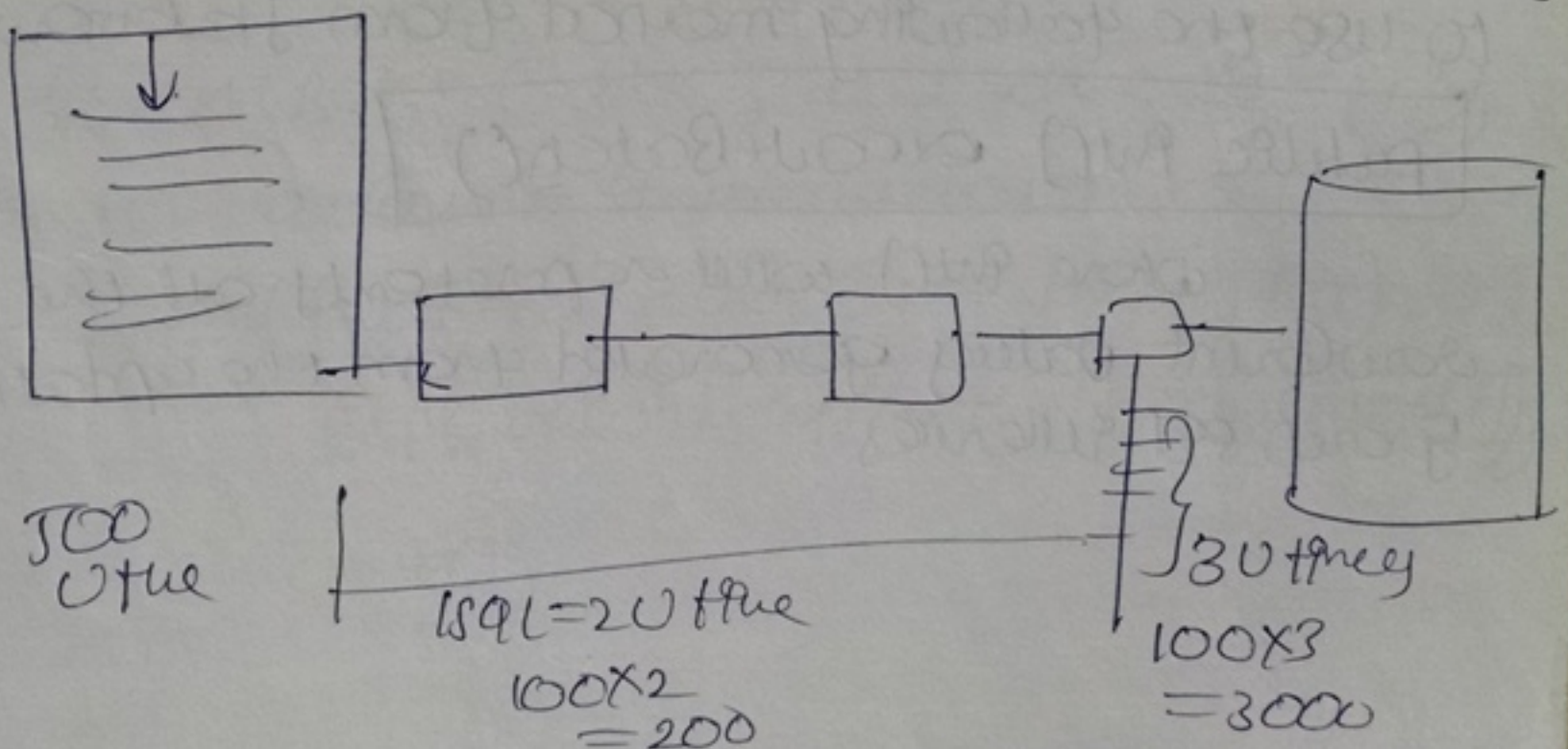
```
}
}
}
```

```
D:\JAVACAPP> javac
             > java
```

Batch updations

—In general in JDBC applications, it is required to provide number of SQL queries according to application requirement.

—When the above if we execute a JDBC application then JVM will send all the SQL queries to the database in sequential manner. If we use the above convention to execute SQL queries in JDBC applications, we have to spend lot of time only to carry or transfer SQL queries from Java application to database, this approach will reduce the performance of JDBC applications.



In the above context, to improve the performance of JDBC applications, we have to use batch updations.

In batch updations, we will gather of collection of the updation group SQL queries as a single unit called as batch and we will send batch of updation group SQL queries

at a time from java application to database
- At database, DBE may execute all the SQL queries and generate the respective rowCount() values in the form of an array to java application.
- To add an SQL query to a batch, we have to use the following method from Statement

```
public void addBatch(String query)
```

- To send a batch of updation group SQL queries from java application to database, and to make the database engine to execute all the batch of updation group SQL queries we have to use the following method from Statement.

```
public int[] executeBatch()
```

where int[] will represent all the rowCount values generated from the updation group SQL queries.

Oracle
DCM

```
Class.forName("oracle.jdbc.
Connection con = DriverManager.getConnection
("jdbc:oracle:thin:@localhost:1521:xe", "System",
                                        "durga");
Statement st = con.createStatement();
st.addBatch("insert into Student values
                    ('S3', 'CCC', 777)");
st.addBatch("update student set smarks=smarks+5
        where smarks<90");
st.addBatch("delete Student where sid='S'");
int[] rowCounts= st.executeBatch();
    for(int i=0; i<rowCounts.length; i++)
      { sop("records updated--"+rowCounts[i]);
      }
      con.close();
    }
  }
```

o/p: records updated --1
records updated --4
records updated --1

Notes If we include selection group sql query in the batch then JVM will raise an exception like java.sql.BatchUpdateException : Invalid batch Command : Invalid select batch Command

Metadata:—

Date: 16th may 2012 wed

Data about the data is called as metadata.

—In jdbc two types of metadata are available

1. Database metadata
2. Resultset Metadata.

Database metadata:— Data about the database is called as Database Metadata.

To represent database metadata, jdbc api has provided a predefined interface, that is java.sql.

DatabaseMetaData

—To get DatabaseMetaData object, we have to use the following method from Connection.

```
public DatabaseMetaData getMetaData()
```

2→ To get the name of the database we have to use the following method.

```
public String getDatabaseProductName()
```

2→ To get the database version name, we have to use the following method.

```
public String getDatabaseProductVersion()
```

2→ To get the database driver, major and minor versions, we have to use the following methods

```
public String getDriverMajorVersion()
public String getDriverMinorVersion()
```

2→ To get the underlying database supported SQL Key words, we have to use the following method

`public String getSQLKeywords()`

2→ To get all the string functions which are supported by the underlying database then we have to use the following method

`public String getStringFunctions()`

2→ To get all the numeric functions which are supported by the underlying database

`public String getNumericFunctions()`

2→ To check whether the underlying database supports batch updations or not, we have to use the following method

`public boolean supportsBatchUpdates()`

2→ To check whether the underlying database support stored procedures or not, we have to use the following method

`public boolean supportsStoredProcedures()`

expln
pgm

```
import java.sql.*;
public class JdbcApp28
{
public static void main(String[] args) throws
{                                    Exception
Class.forName("oracle.jdbc.driver.OracleDriver");
Connection con = DriverManager.getConnection
("jdbc:oracle:thin:@localhost:1521:xe","System",
                                    "durga");
DatabaseMetaData dbmd=con.getMetaData();
sop(dbmd.getDatabaseProductName());
        sop();
sop(dbmd.getDatabaseProductVersion());
        sop();
sop(dbmd.getDriverMinorVersion());
        sop();
sop(dbmd.getDriverMajorVersion());
        sop();
sop(dbmd.getSQLKeywords());
        sop();
sop(dbmd.getStringFunctions());
        sop();
sop(dbmd.getNumericFunctions());
        sop();
sop(dbmd.getTimeDateFunctions());
        sop();
```

sop(dbmd.supportsStoredprocedures());

}

}

D:\jdbc10>javac jdbcApp28.java

>java jdbcApp28 ↵

ResultSetMetaData:- Data about the data available in resultset object is called as ResultSetMetaData.

2→ In jdbc applications, ResultSetMetaData includes the number of columns which are available in resultset object, names of each and every column, datatypes of each and every column and so on--

* To get resultsetmetadata, we have to use the following method from ResultSet

[public ResultSetMetaData getMetaData()]

NOTE: observe we the same method is available in Connection. But it can be used to get DatabaseMetaData.

2→ To get the number of columns, which are available in ResultSet object, we have to use the following method

[Public int getColumnCount()]

2→ To get the name of a particular column available in ResultSet object, we have to use the following method

[public String getColumnName(int columnIndex)]

2→ To get the datatype of a particular column we have to use the following method

```
public String getColumnTypeName(int column index)
```

2→ To get the display size of a particular column, we have to use the following method

```
public int getColumnDisplaySize(int column index)
```

2→ Expl

```
import java.sql.*;
public class JdbcApp29
{
public static void main(String[] args) throws Exception
{
Connection con = DriverManager.getConnection
("jdbc:oracle:thin:@localhost:1521:xe","system","durga");
Statement st = con.create Statement();
ResultSet rs = st.executeQuery("select * from emp1");
ResultSetMetaData rsmd = rs.getMetaData();
int count = rsmd.getColumnCount();
for(int i=0; i<=count; i++)
{ sop("column name = "+rsmd.getColumn Name(i));
```

```
sop("DataType-"+rsmd.getColumnTypeName(i));
sop("columnsize-"+rsmd.getColumnDisplay
                                    Size(i));
        sop("-------------");
    }//for loop
}//main method
}//class
```

D:\JDBC> javac JDBCApp29.java

java JDBCApp29

2→ The following example demonstrates how to retrieve the data from a particular database table along with column names and how to display columns data and the table body on the command prompt.

```
import java.sql.*;
 public class JdbcAPP30
   {
   public static void main(String[] args) throws
                                          Exception
     {
Connection con = DriverManager.getConnection
("jdbc:oracle:thin:@localhost:1521:xe", "system",
                                    "tenga");
 Statement st = con.createStatement();
 ResultSet rs = st.executeQuery("select * from emp1");
 ResultSetMetaData md = rs.getMetaData();
    int count = md.getColumnCount();
```

In JDBC column count starts with '1' Here we are not use '0'

```
    for(int i=1; i<=count; i++)
      {
      SOP(rs.getString(md.getColumn.
      Name(i)+"|");
      }
    SOP();
SOP("----------");
```

```
while(rs.next())
    {
for(int i=1; i<=count; i++)
    {
SOP(rs.getString(i)+"|");
    }
    SOP();
}//while
  }
}
```

—D8

Connection pooling

Date: 17th May, 2012 Thur

—In general in JDBC application, when we have a requirement to perform database operations we will establish a connection with the database from a java application, at the end of the application we will close the connection. That is destroying connection object.

—In JDBC applications, every time establish the connection and closing the connection may increase burden to the JDBC applications; it will reduce the performance of the JDBC applications.

In the above context, to improve the performance of JDBC applications; we will use an alternative called as Connection pooling

—In connection pooling at the time of application startup we will prepare a fixed number of connection objects and we will keep them in a separate base object called as pool object, in JDBC applications, when we have a requirement to interact with the database then we will get the connection object from pool object, we will assign it to the respective client application.

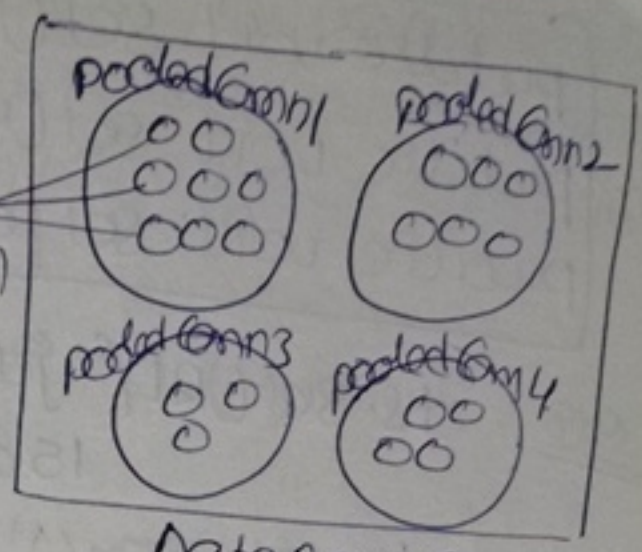
—At the end of the JDBC application, we will keep the same connection object in the respective pool object without destroying.

The above mechanism to improve the performance of the application is called as Connection pooling.

— If we want to implement Connection pooling in JDBC applications then we have to use the following steps.

Step1: prepare DataSource object

DataSource is an object, it able to manage all the jdbc parameters which are required to establish the connections.



- To represent DataSource object java api has provided a predefined interface, that is javax.sql.DataSource (observe it is not java.sql.--)
- DataSource is an interface provided by jdbc api but whose implementation classes are provided by all the database vendors. With the above convention oracle has provided an implementation class to data source interface in ojdbc6.jar file. that is oracle.jdbc.pool.oracleConnectionpoolDataSource

Ex: oracleConnectionpoolDataSource ds = new oracleConnectionpoolDataSource();

Step2: set the required jdbc parameters to DataSource object.

To set the jdbc parameters like driver url, database username and password to the dataSource object we have to use following methods

```
public void setURL(String driver_url)
public void setUser(String user_name)
public void setPassword(String password)
```

Eg: ds.setURL("jdbc:oracle:thin:@localhost:1521:xe");
ds.setUser("system");
ds.setPassword("durga");

Step③: get pooled connection object:

PooledConnection is an object provided by data source, it can be used to manage number of connection objects.

To represent pooled Connection object, JDBC Api has provided a predefined interface, that is javax.sql.PooledConnection.

— To get pooledConnection object we have to use the following method from DataSource.

```
public PooledConnection getPooledConnection()
```

Eg: PooledConnection pc = ds.getPooledConnection();

Step④: get Connection object from pooledConnection

To get Connection object from pooledConnection we have to use the following method

Eg: Connection con = pc.getConnection();

```
public Connection getConnection()
```

Step(5) : After getting connection prepare Statement object or preparedStatement or callableStatement and perform the respective database operations.

```
Statement st = con.createStatement()
```

=

example pgm

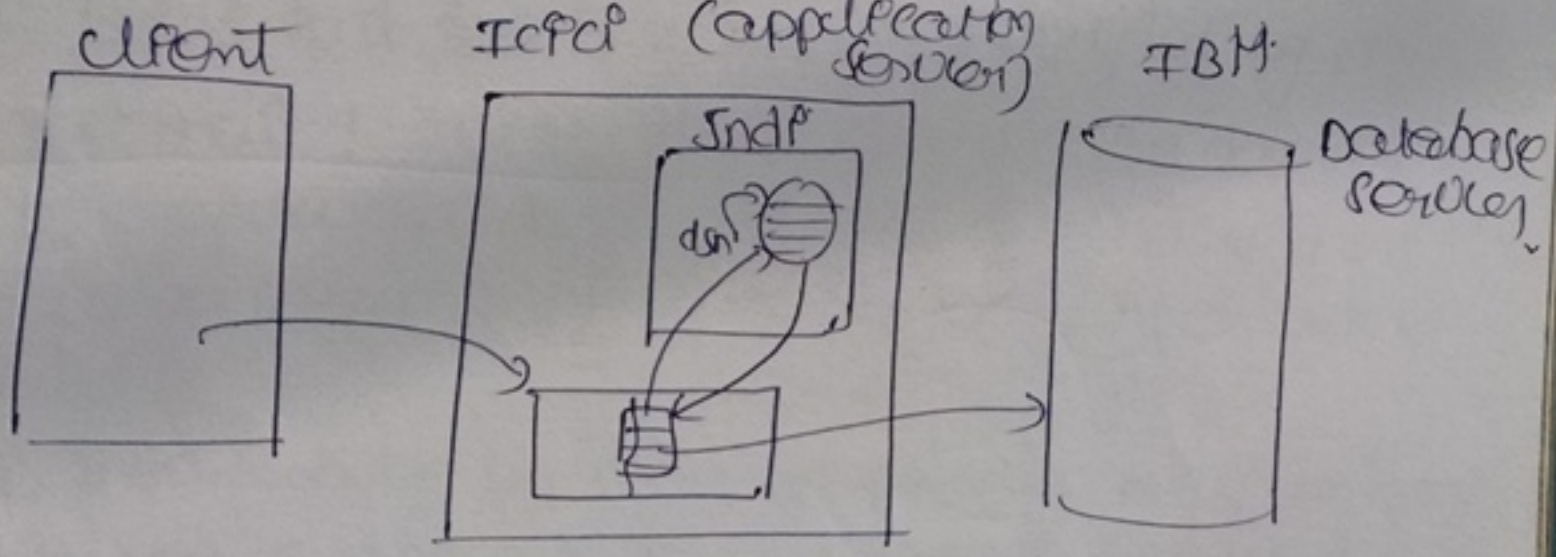
```
import java.sql.*;
import java.sql.*;
import oracle.jdbc.pool.*;
public class JDBCApp31
{
public static void main(String[] args) throws Exception
{
OracleConnectionPoolDataSource ds=new
  oracleConnectionPoolDataSource();
ds.setURL("jdbc:oracle:thin:@localhost:1521:XE");
ds.setUser("system");
ds.setPassword("durga");
PooledConnection pc = ds.getPooledConnection();
Connection con = pc.getConnection();
Statement st = con.createStatement();
ResultSet rs = st.executeQuery("select * from
                                emp");
while("EMP ENAME SAL");
System.out.println("-----");
```

```
while(rs.next())
  {
Sop(rs.getInt(1)+"  "+rs.getString(2)+"  "+
    rs.getFloat(3));
 }//while.

}//main

}//class
```

O/P D:\JDBC\> javac JDBCAPP3.java

Note:- The above approach of implementing connection pooling concept is suggestable up to standalone applications, it is not suggestable in the enterprise applications.

— If we want to implement connection pooling in enterprise applications, we have to use the underlying application server provided JDBC middleware service.



BLOB & CLOB objects :- upto now in JDBC applications, we are able to interact with the database in order to insert a record, retrieve a record and so on -- with the varchar data or number data and so on --

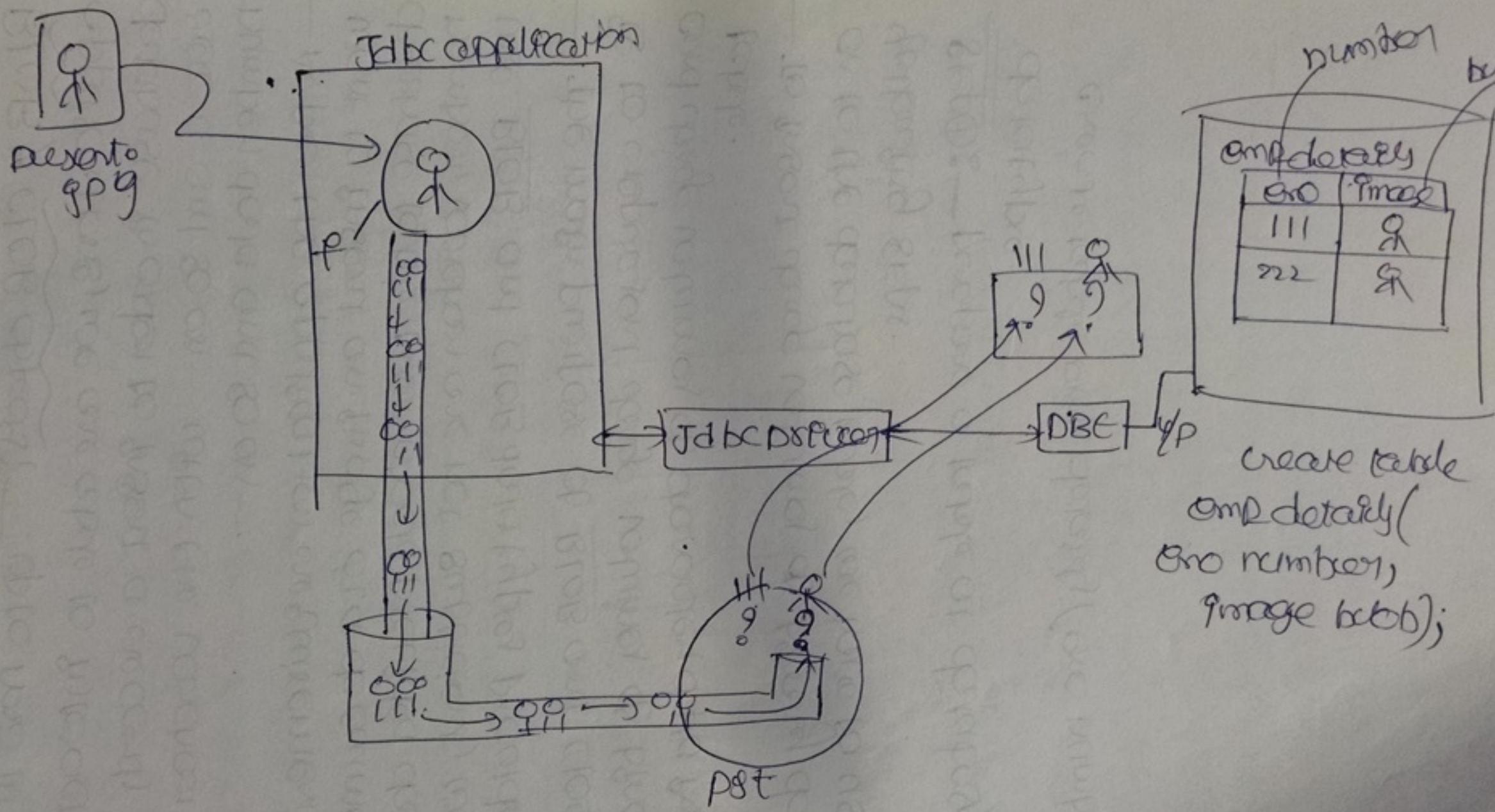
- As per the application requirements if we want to insert an image or a document in the database table then oracle provided datatypes number, varchar are not sufficient, we have to use BLOB and CLOB datatypes provided by oracle.

- The main purpose of BLOB and CLOB datatypes is to represent large volumes of binary data and large volumes of character data in database table.

- To insert large volumes of binary data (an image) on to the database table, we have to use the following steps.

Step①:- prepare a table at database with BLOB datatype.

create table emp_details (eno number, image blob);



create table
emp details(
ero number,
Image blob);

Step(2): Represent an Image file in the form of File class object.

```
File f=new File("Desert.jpg");
```

Step(3): get File class object content in the form of FileInputStream

```
FileInputStream fis=new FileInputStream(f);
```

Step(4): create preparedStatement object with insert sql query format.

```
preparedStatement pst=con.prepareStatement
("insert into emp details values(?,?)");
```

Step(5) Set Binary stream to the Blob type positional parameter in preparedStatement

To set a binary stream to the blob type positional parameter we have to use the following method from preparedStatement.

```
public void setBinaryStream(int paramIndex,
InputStream is, int length)
```

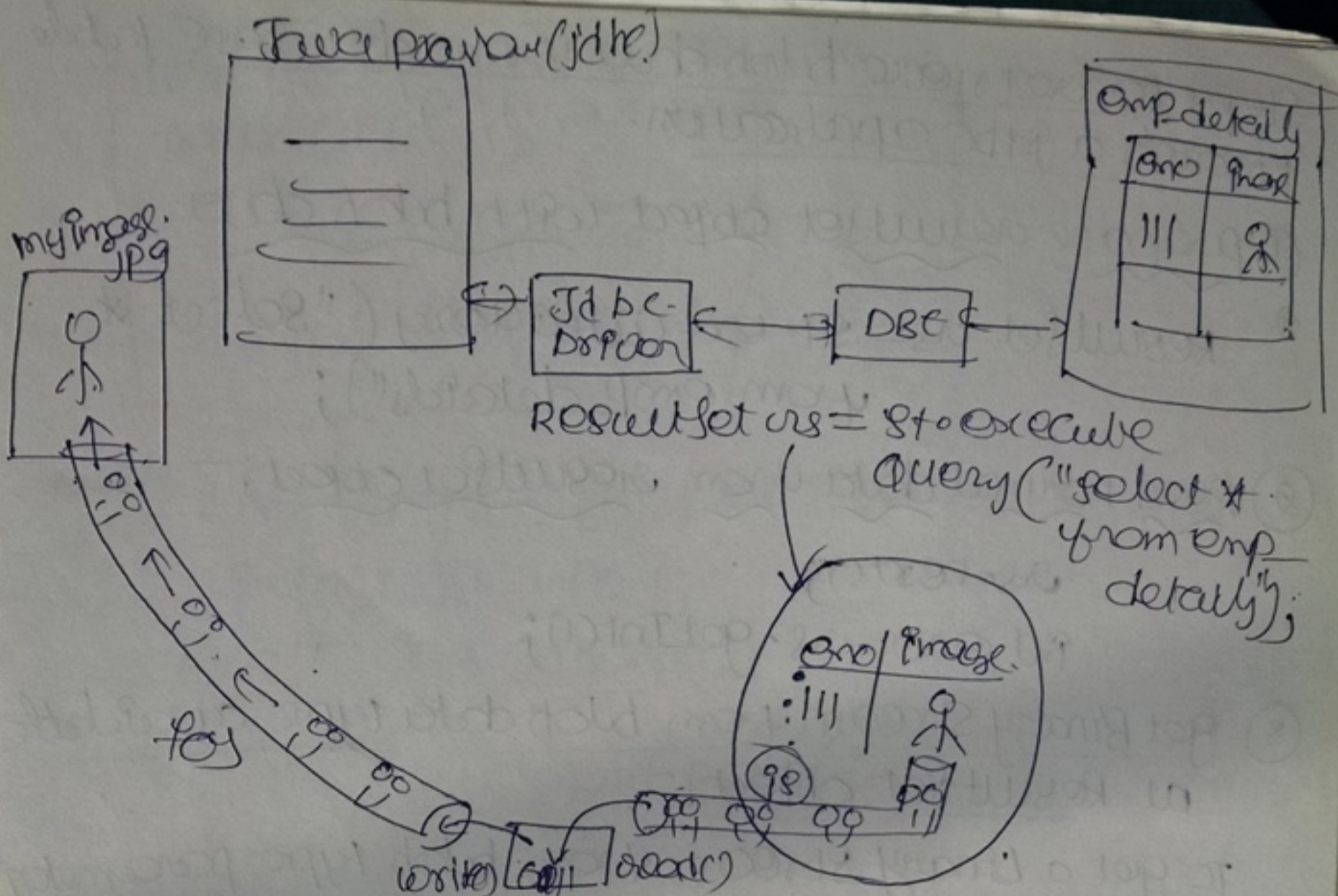
or pst.setBinaryStream(2, fis, (int)f.length());

Step(6):- execute preparedStatement

pst.executeUpdate()

jdbcbasic

```
import java.io.*;
import java.sql.*;
public class JdbcApp32
{
    public static void main(String[] args)
                                    throws Exception
    {
Class.forName("oracle.jdbc.driver.OracleDriver");
Connection con=DriverManager.getConnection
("jdbc:oracle:thin:@localhost:1521:xe","system",
File f=new File("Desert.jpg");    "durga");
FileInputStream fis=new FileInputStream(f);
PreparedStatement pst=new PreparedStatement
("insert into empdetails values(?,?)");
        pst.setInt(1,111);
        pst.BinaryStream(2, fis(int)f.length());
        pst.executeUpdate();
        sop("Image successfully inserted");
        con.close();
```



Before Pri database

Date: 18th May, 2012 Fri

Steps to retrieve blob data from database table through a jdbc application.

① Prepare resultset object with blob data.

```
ResultSet rs = st.executeQuery("select * from emp_details");
```

② Read normal data from resultset object:-

```
rs.next();
int eno = rs.getInt(1);
```

③ Get Binary stream from blob data type available at ResultSet object:-

To get a binary stream from blob type parameter available at resultset object, we have to use the following method.

```
public InputStream getBinaryStream(int parameterIndex)
```

(or)

```
InputStream is = rs.getBinaryStream(2);
```

④ Prepare the target resource to hold up the retrieved blob data by using FileOutputStream

```
FileOutputStream fos = new FileOutputStream("myimage.jpeg");
```

⑤ Read bit by bit from InputStream and write the same bit by bit on FileOutputStream to store the retrieved data on target file.

```
int i = is.read();
while (i!=-1)
    {
      fos.write(i);
      i=is.read();
    }
```

example pgm :

```
import java.sql.*;
import java.io.*;
public class JdbcApp33
  {
public static void main(String args[]) throws
                                        Exception
  {
Connection con = DriverManager.getConnection
("jdbc:oracle:thin:@localhost:1521:xe", "System",
                                        "durga");
Statement st = con.createStatement();
ResultSet rs = st.executeQuery("select * from
    rs.next();                          emp_details");
sop("Employee number-"+rs.getInt(1));
InputStream is = rs.getBinaryStream(2);
FileOutputStream fos = new FileOutputStream
                                ("myimage.jpeg");
int i = is.read();
  while (i!=-1)
    { fos.write(i);
      i=is.read();
    }
```

```
sop("Image created successfully");
                with myimage.jpg
        fos.close();
        bos.close();
    }
}
```

CLOB Datatype:

— If we want to perform operations with CLOB Data type then we have to use the same steps what we have used with BLOB Datatype. But we need to provide the following replacements

① blob ⟶ clob.
② InStream ⟶ Reader.
③ FileOutputStream ⟶ FileWriter.
④ setBinaryStream() ⟶ setCharacterStream()
⑤ getBinaryStream ⟶ getCharacterStream()
⑥ FileInputStream ⟶ FileReader.

related to blob

related to clob

Before fn database table

SQL> create table webInfo(appname varchar2(15), depdesc clob);
Table created✓

SQL> commit;
Commit completed✓

2→ The following example demonstrates how to insert a clob data(a document) on to the database table from jdbc application.

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

public class JdbcApp34
{
  p s v m(String a) throws Exception
  {
```

```
Class.forName("oracle.jdbc.driver.OracleDriver");
Connection con = DriverManager.getConnection
("jdbc:oracle:thin:@localhost:1521:xe", "system",
                                        "durga");
PreparedStatement pst = con.prepareStatement
("insert into web_info values(?,?)");
    pst.setString(1, "app1");
File f = new File("web.xml");
FileReader fr = new FileReader(f);
pst.setCharacterStream(2, fr, (int) f.length());
        pst.executeUpdate();
System.out.println("web application stored in DB successfully");
        fr.close();
        con.close();
    }
}
```

- To execute the above application, we must provide a table with clob data type in database.

SQL> create table webinfo(appname varchar(11),
appdesc clob);

— The following example demonstrates how to retrieve the clob data from database table through a JDBC application.

```
import java.sql.*;
import java.io.*;
public class JdbcApp35
  {
public static void main(String a[]) throws Exception
    {
Connection con = DriverManager.getConnection
("jdbc:oracle:thin:@localhost:1521:xe","system",
                                    "durga");
Statement st = con.createStatement();
ResultSet rs = st.executeQuery("select * from
    rs.next();                      web_info");
SOP("Application name—"+rs.getString(1));
FileWriter fw=new FileWriter("myweb.xml");
Reader r=rs.getCharacterStream(2);
int i=r.read();
while(i!=-1)
  { fw.write(i);
    i=r.read();
  }
SOP("web.xml retrieved successfully with
                                myweb.xml");
    fw.close();
    r.close();
  }
}
```