

J2EE

(Naresh Technology)

ameerpetmaterials.blogspot.in

other without establishing the connection just by sending & receiving the msg packets which are called as datagram packets

(ii) In connection-oriented C^n , the source process should specify destination address only once ie at the time of establishing the connection, whereas

In connection-less C^n , the source process should specify the same destination address again & again on every msg packet.

(iii) In connection-oriented C^n the sequence of data arrives at the destination is guaranteed, but this is not guaranteed in connection-less C^n .

- * The sun micro systems provide a package called `java.net` where there are some predefined classes are available to support these 2 types of C^n s in the net based java appn.
- * The connection-oriented C^n is also called as client-server C^n , where the client appn will make a connection request to a server appn so that a server will establish connection to the requested client address so that both client & server communicate each other indefinitely through the established connection.
- * The connection-less is also called as datagram C^n where both the appn's communicate each other by exchanging msg packets ie datagram packets
- * The following practical ex: demonstrate how the client & server will chat ie. communicate each other indefinitely through the established connection by creating `socket` class objects & this programming is also called

-5-09

Client side App'n

chatclient.java

```
import java.net.*;  
    "    .. io.*;
```

```
class ChatClient  
{
```

```
    p.s.v.m (String args[]) throws Exception
```

```
{
```

```
    Socket s = new Socket ("local host", 9001);
```

// This will create an object of socket class available in java.net package along with a respective server hostname, server port no; i.e. localhost, 9001.

At the same time, the JVM automatically prepares one connection request packet along with the server address details specified by the user & client address details automatically captured by the JVM.

This connection request packet is sent to a server, we assume that the server app'n will receive this request & establish the connection to the requested client.

```
DataInputStream stdin = new DataInputStream  
(System.in);
```

```
InputStream m = s.getInputStream();
```

// Where getInputStream() of socket class that will return the address of the inner Input Stream object available in the socket

```
DataInputStream din = new DataInputStream (in);
```

// This will create an object of DataInputStream class which is available in java.io package by connecting to the inner input stream object

```
OutputStream k = s.getOutputStream();
```

// Where getOutputStream is the method of socket class that will return the address of inner op stream object

```
PrintStream ps = new PrintStream (k);
```

// This will create an object of PrintStream class by connecting to the InnerOutputStream object
while(true)

```
{
```

// This is infinite while loop

```
s.o.p("Enter the msg for the server");
```

```
String msg = stdin.readLine();
```

```
ps.println(msg);
```

// Where println is the method of PrintStream class that will send the respective string msg to a server through the inner o/p stream object

```
if (msg.equals("bye"))
```

```
break;
```

```
String receivedmsg = din.readLine();
```

// Where readLine is the method of DataInputStream class that will read one line of reply string msg received from the server through the inner InputStream object.

```
    S.O.P ("Received msg from the server" +  
          receivedmsg);
```

```
}
```

```
}
```

```
}
```

ServerSide APP'n

ChatServer.java

```
import java.net.*;  
        .. .ID.*;
```

```
class ChatServer
```

```
{
```

```
P.S.V.M (String args[]) throws Exception
```

```
{
```

```
ServerSocket ss = new ServerSocket(9001);
```

// This will create an object of ServerSocket
class available in java.net package by connecting
to a specified port no. of the server i.e. 9001 at
the current host name i.e. localhost

```
Socket s = ss.accept();
```

// Where accept is the method of server
Socket class that will read the connection request
packet received from the client from the server
socket & establish the connection to the requested
client address by automatically creating a socket
class object at server side.

If there is no connection request packet
available in the server socket i.e. received
from the client, then the state of a server is
in waiting state until it receive the connection

Da

Ou

Pr

Dat

Wh

{

}

15-5-

request packet from the client

```
InputStream m = s.getInputStream();
DataInputStream din = new DataInputStream(m);
OutputStream k = s.getOutputStream();
PrintStream ps = new PrintStream(k);
DataInputStream stdin = new DataInputStream(System.in);
while(true)
{
    String receivedmsg = din.readLine();
    if(receivedmsg.equals("bye"))
        break;
    System.out.println("Received msg from the client " + receivedmsg);
    System.out.println("Enter the msg for the client");
    String msg = stdin.readLine();
    ps.println(msg);
}
```

15-5-09

Java Distributed Technology

Introduction:

- * The service objects i.e. the objects implemented with the business logic func of the enterprise Java app'n are distributed at the various hosts in the computer network so that any client in it can access & get the services of

these objects, this technology of distributing the service objects in the n/w is called as java distributed technology.

* The client jvm is the jvm where the client app'n is running

* A server jvm is the " " " server app'n is running & where the service object is created

* The service object is capable of providing the services to the remote clients, so the service object can also be called as Remote object.

* proxy is an object that resides at the client jvm & help the client app'n to communicate with the server jvm

* helper is an object that resides at the server jvm associated with the service object & help the server app'n to communicate with the client jvm

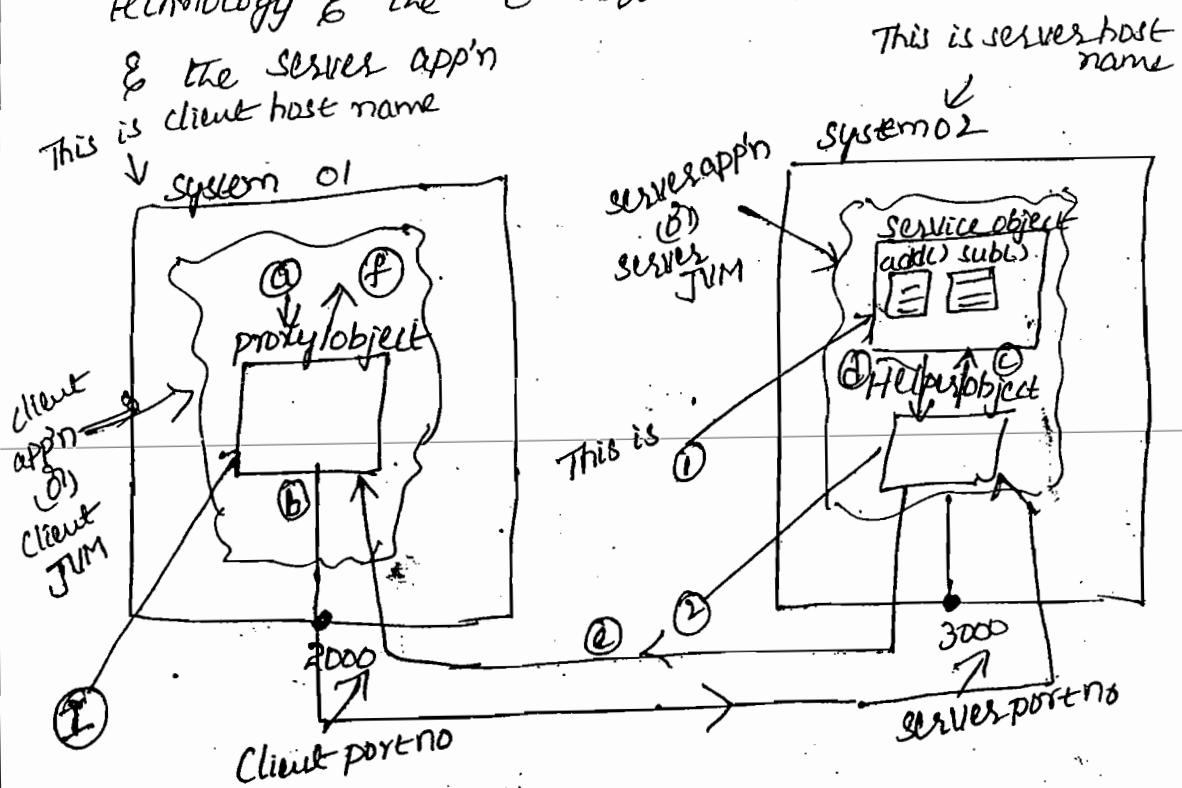
* whenever the client make a request to the proxy object to get the services from the remote service object then the proxy object will ~~marshal~~ MARSHAL the client request i.e. converting the request into a stream of bytes & send it to the helper object at the server

* The helper object at the server will UNMARSHAL the received request i.e. converting the received stream of bytes into original request format & pass it to the associated service object.

Client
app'n
(O)
Client
JVM

①

- * The helper object will receive the response from the service object & then MARSHAL the " " & send it back to proxy of the requested client.
 - * The proxy of the client will UNMARSHAL the received response & give it to the client appn
- The following ex: demonstrate this basic fundamental architecture of this java distributed technology & the C sequence b/w the client app'n & the server app'n



① This is a service object created in the server appn with some business logic func's w.r.t appn req's

② This is the helper object created in the server appn which is associated with the service object

③ This is the proxy object available in the client appn which will help in the client appn

to establish the connection to a server app'n,
sending the request to the server app'n &
receiving the response from the server app'n.

Note ① ② ③ ④ ⑤ ⑥ are the sequence
of operations

17.5
ver
of

demonstrate

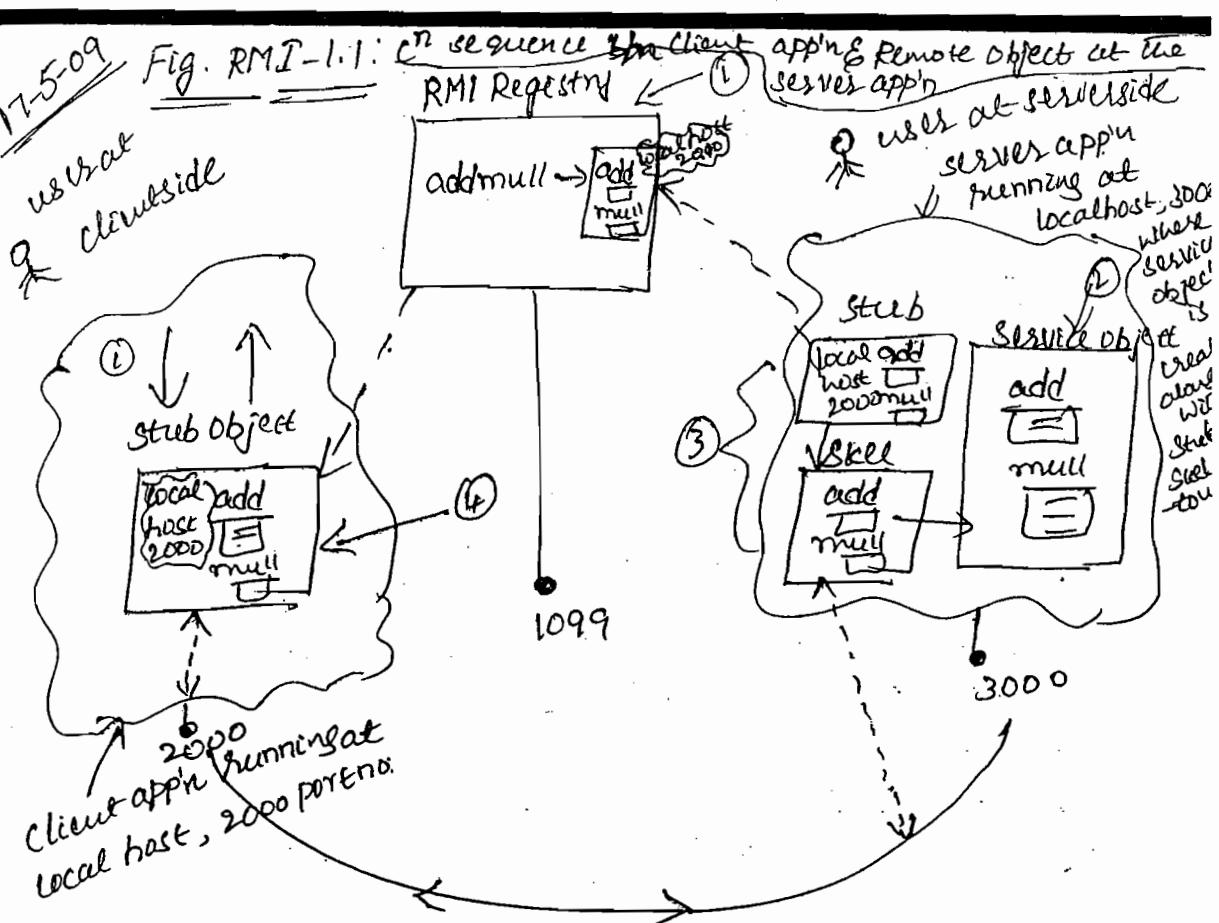
The following 6 steps in the above
sequence of op. b/w the client app'n & the
server app'n in java distributed technology

- ① The client app'n will make a request
to the proxy object to get the service
from the remote service object
- ② The proxy object at client side will
MARSHAL the received request into a
stream of bytes & send it to the helper
object at the server app'n
- ③ The helper object at the server app'n will
receive the request, unmarshal the request
& pass it to the service object
- ④ The helper object will collect the response
from the service object
- ⑤ The helper object will marshal the response
& send it to the proxy object at the requested
client app'n
- ⑥ The proxy object at the client app'n will
unmarshal the received response & give it
to the client app'n.

cli
dc

17-5-09

Fig. RMI-1.1: Cⁿ sequence b/w Client app'n & Remote Object at the server app'n



① This is RMI Registry running at localhost, 1099

port no. where this stub object of the service object is registered i.e. binded with some unique name called addnull from the server app'n

② This is service object created by the user

③ These are stub, skeleton objects automatically created along with the service object by the RMI architecture in the server app'n.

④ This is stub object of the service object received into the client app'n by looking into RMI Registry with the required name i.e. addnull

Note Now the Stub object at client side & the Skeleton object at server side will communicate each other to get the services of the service.

as peer to peer cⁿ, by marshaling &
unmarshaling "the data".

5-09

RMI (Remote Method Invocation)

sequence of concepts:

1. Introduction of RMI
2. procedure to develop RMI app'n at the server
3. " " access Remote object from the client app'n
4. RMI registry
5. RMI architecture
6. RMI exceptions
7. parameters passing in RMI.

1. Introduction of RMI

* This is the technology used to call the business logic methods of the Remote object available at one JVM from some other JVM in vendor independent approach.

* The JVM's are the different entities located on the same computers or separate computers connected with the network.

* This RMI technology will help the Java app'n in the n/w to share the resources to distribute the processing load across the diff computer systems in the n/w.

→ procedure to develop RMI app'n at the server

* We should follow the following steps to develop the RMI app'n at the server
i.e.

(1) We should define one public user defined interface by extending Remote interface available in `java.rmi` package with a set of required abstract methods i.e. Business logic methods, all these methods should be declared as throws RemoteException such as

```
public interface MyRemoteIface extends Remote  
{  
    some business logic abstract methods  
}
```

(2) We should define one public implementation class by extending the predefined class called `UnicastRemoteObject` which is available in `java.rmi.server` package & by implementing the above userdefined Remote interface Business logic methods acc to app'n req's. such as

```
public class MyRemoteImpl extends UnicastRemoteObject  
    implements MyRemoteIface  
{  
    implementation of all the business logic methods  
}
```

(3) We should compile the above 2 java files to generate the class files

(4) We should generate the stub, skelton class file for the implementation class by using the

> rmic MyRemoteImpl

① MyRemoteImpl - stub.class

② MyRemoteImpl - skel.class

This will generate automatically
Stub,skel class files for the implementation
class of the Remote object such as

MyRemoteImpl - stub.class,

MyRemoteImpl - skel.class.

(5) We should start RMI registry at the
current host name i.e. local host with the
following command i.e.

> start rmiregistry

* This will start the RMI registry at
the local host at the default port no. i.e. 1099

(6) *Where RMI rmiregistry is the common point

③ the common interface provided by Sun Micro
Systems b/w the client app'n & the server app'n
where the Remote objects are bound i.e. stored
with some unique names from the server app'n

(6) We should write the server app'n which is
also called as registry app'n to create the
service object i.e. Remote object & to bind
this Remote object into the RMI registry
such as

* We should use the following code to
create the implementation class object
i.e. Remote object in the server app'n. i.e.

```
MyRemoteImpl mri = new MyRemoteImpl();
```

- * At the time of creating this Remote object, the stub, skel objects are automatically created by the RMI architecture where skel object is associated with the Remote object, the stub object is " " skel " at server app'n.
 - * The address of the server app'n i.e. server host name & the server port no: is automatically stored into the stub object.
 - * The Remote object will have the business logic methods implemented by the user acc to app'n req's, whereas the stub, skel objects will have the same set of methods automatically implemented with the networking protocols, marshaling & unmarshaling code by the RMI architecture.
 - * We should use the following code in the server app'n to bind the Remote object into the rmi registry with some name such as
- ```
Naming.bind ("rmiaddmnl", mri);
```

\* Where bind is the static method of Naming class available in java.rmi package. This method will automatically bind i.e. store the stub object of the corresponding Remote object which is already created at the server with the name such as rmiaddmnl into the rmi registry.

Q) We should compile & execute this server app'n so that the stub object of the Remote service object is binded into a RMI registry & ready to provide the services to the remote clients.

9-5-09  
Q3. Procedure to access Remote object from the Client app'n

\* We should follow the following steps to access the Remote object from the Client app'n

(1) We should get the copy of the stub object of the Remote service object from the RMI registry such as

Remote r = Naming.lookup("rmiregistry");  
where lookup is the static method of Naming class that will look into the rmiregistry which is already running & get the copy of the stub object of the Remote object available against to the specified name such as rmiregistry into the client app'n, the returning type of this method is Remote interface.

(2) Now the stub object which is available at client side will automatically establish the connection b/w Client app'n & the server app'n.

Where r is the reference variable of Remote interface & currently referencing to the stub object of the Remote object at client side.

(3) We should convert i.e. typecast this remote reference into the required userdefined remote interface types such as

```
MyRemoteInterface mri = (MyRemoteInterface)r;
```

\* Now mri is referencing to the same stub object of the Remote object at Client side, so we can call the required business logic methods through this reference variable such as

```
int addresult = mri.add(25,35);
```

\* This will call the respective add business logic methods available in the Remote service object through the stub object available at Client side & the Skelton object available at server side.

### RMI Registry:

\* This RMI Registry is the common interface provided by the sun Micro systems b/w the Client app'n & the Server app'n

\* All the Remote objects that are to be accessed from the Client JIM must be binded i.e. registered into the RMI registry with some unique name.

\* This Registry should be done from the Server app'n which is also called as registry app'n

\* Once the stub object of the Remote object is binded into the RMI registry then any client can look into this RMI registry with the required name & get the copy of the stub object of Remote object

\* Once the client get the stub object then the Client can call the required business logic methods on this stub object so that the stub object will marshal the Client request & send it to the Skelton object

- \* The stub object at the server will receive the request, unmarshal the request & pass it to the associated remote object so that the respective business logic method is get executed at the Remote object.
- \* The result of the method is transported back through the n/w to the requested client with the help of stub & stub objects.
- \* The above figure of RMI II will demonstrate how the client app'n will communicate with the Remote service object available at the server app'n with the help of RMI registry running at localhost, 1099 port no:

(2)

5-09  
→ The following practical app'n demonstrate how the client app'n will get the services from the remote service object available at the server app'n with the help of RMI Registry.

Name of the app'n folder is → rmiaddress



- (1) MyRemoteInterface.java
- (2) MyRemoteImpl.java
- (3) MyRemoteRegistry.java
- (4) Testclient.java

### (1) MyRemoteInterface.java

Note This is the user defined RemoteInterface at server side with a set of abstract business logic methods acc to app'n req's

```
import java.rmi.*;
public interface MyRemoteIface extends Remote
{
 public String message(String s) throws RemoteException;
 public int add(int i, int j) throws RemoteException;
 public int mul(int i, int j) " "
}
}
```

## (2) MyRemoteImpl.java

This is the implementation class for the above userdefined interface & the server ask to appn req's.

```
import java.rmi.*;
" " " " .Server.*;
public class MyRemoteImpl extends UnicastRemoteObject
 implements MyRemoteIface
```

```
{
 public MyRemoteImpl() throws RemoteException
{
}
```

This is the public default constructor for the implementation class, this constructor is called at the time of creating an object.

```
 public String message(String s)
{
```

```
 System.out.println("This is message method");
 return ("Ur given msg is:" + s);
 }
```

public int add (int i, int j) {

    s.o.p ("This is add method");  
    return (i + j);

}

public int mul (int i, int j)

{

    s.o.p ("This is mul method");  
    return (i \* j);

}

(3) MyRemoteRegistry.java

This is the server app'n which is also called as Registry app'n to create the implementation class object i.e. RemoteService Object & to bind the stub object of the RemoteService object into the RMI Registry.

import java.rmi.\*;  
public class MyRemoteRegistry

{

p.s.v.m (String args[]) throws Exception

{

    MyRemoteImpl mri = new MyRemoteImpl();  
    Naming.bind ("rmi:addrmul", mri);  
    s.o.p ("Remote object is binded into rmi registry")

}

}

#### (4) TestClient.java

This is the Client side app'n to get the services of the RemoteService object available at the server by getting the copy of the stub object of the RemoteService object from the RMI registry.

```
import java.rmi.*;
public class TestClient
```

```
{
```

P.S.V.M (String args[]) throws Exception  
This connection is established  
btw the client & rmiregistry.

```
 Remote r = Naming.lookup("rmiregistry");
 MyRemoteIface mri = (MyRemoteIface)r;
 DataInputStream stdin = new DataInputStream
 (System.in);
```

// This will create an object of DataInputStream  
class which is available in java.io package by connecting  
to the keyboard of a computer system so that we can  
read the data from the keyboard line by line

```
S.O.P("Enter the msg to server");
String msg = stdin.readLine();
S.O.P("Enter first no:");
int fno = Integer.parseInt(stdin.readLine());
S.O.P("Enter second no:");
int sno = Integer.parseInt(stdin.readLine());
String receivedmsg = mri.message(msg);
int addresult = mri.add(fno,sno);
int mulresult = mri.mul(fno,sno);
```

S.O.P (received msg);  
" ("Result of addition"+addresult);  
" (" " multiplication"+mulresult);  
" ("Bye Bye");

}

}

procedure to compile & execute this app'n

(1) We should compile the above 4 java files to generate the class files with the command as

>javac \*.java

madhuzee\madhumal\miaddmul>javac \*.java

(2) We should generate the stub, skel classes for the implementation class with the following command

>rmi c MyRemoteImpl

(3) We should start RMI Registry with the following command i.e.

>start rmiregistry

This will start rmiregistry i.e. empty registry at the current host name i.e. local host at the default port no. of 1099.

(4) Now we should start or execute the server app'n i.e. registry app'n with the following command i.e.

>java MyRemoteRegistry

(5)

10

Sc

21-5

(5) Now we should execute the client app'n by opening one more new command window <sup>with</sup> ~~on~~ the localhost with the following command

> Java TestClient

O/P <sup>client-side</sup> Your given msg is: Hello Server  
Result of addition --- 85  
" " multiplication -- 1500

Bye Bye.

Server Side

Remote object is binded into rmiregistry.

This is msg method

" " add "  
" " mul "

~~21-5-09~~

### RMI Architecture

The following 3 layers of sw in the RMI architecture will support the java app'n to call the business logic methods of the remote service object ie

#### i) The Stubs / skeletons layer

This layer will interpret the method calls made by the client & redirect those calls to the remote service object, where the stub objects are available at client side whereas the skeel objects are available at server side.

\* The stub object at the client-side is the representation of Remote Service object, this stub object will act as proxy at client side.

\* The skeel objects at the server side is associated with

The Remote service object , this skel object will act as helper at server side

- \* Whenever the client call the business logic method on the stub object then the stub object will forward the request through the RMI architecture through the skel object available at the server.
- \* The skel object at the server will receive this request & pass the request to the service object so that the respective business logic method is get executed.
- \* The skel object will collect the response i.e. returning value of the method from the service object & send it back to the requested client.
- \* This stub & skel objects will do the job of marshalling & unmarshalling the data.

## 2. The Remote Reference layer

This layer will interpret the references made by the client & it will connect to the corresponding remote service objects running at the server appn.

The protocol used in this layer is called as JRMP i.e. Java Remote Method protocol.

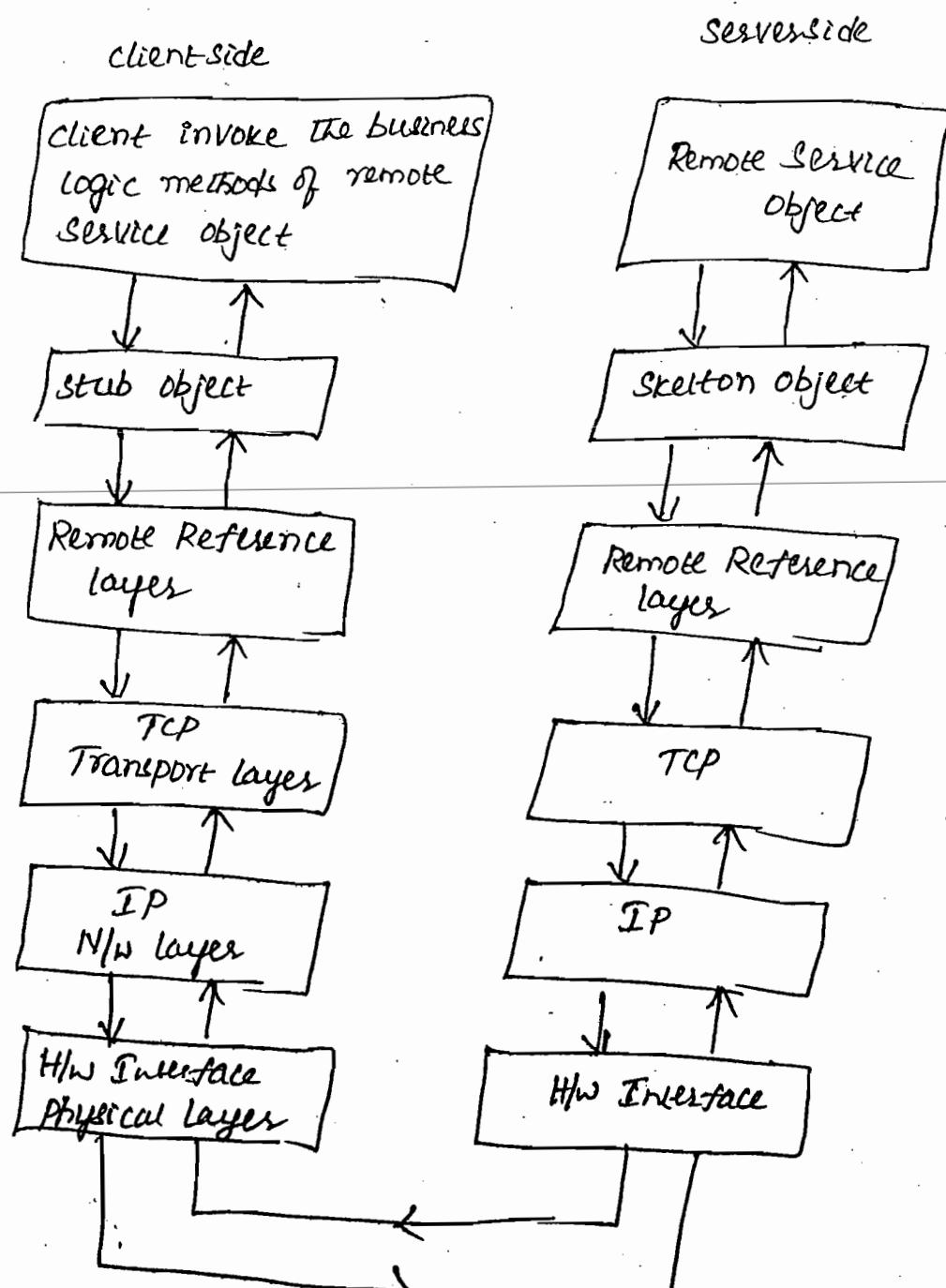
## 3. The Transport layer

This layer is based on TCP/IP protocol & it will get the request from the Remote Reference layer & then establish the connection b/w client JVM & server JVM in the new based on the references i.e. addresses requested from the

the Remote Reference layer.

- \* This transport layer will make stream based NW connection b/w the Client & server.
- The following ex: demonstrate above 3 layers of SW in RMI Architecture.

fig RMI 1.2



## RMI Exceptions

→ The following types of exception may be raised in the RMI app'n.

(1) AlreadyBoundException: This exception may be

raised if an attempt is made to bind the remote service object into the rmi registry with the name which is already existing in the RMI registry.

(2) NotBoundException This exception may be

raised if an attempt is made to look in the rmi registry with the name i.e. not existing in RMI Registry

(3) RemoteException: This communication related

exception may be raised at the time of calling the remote methods due to internal n/w working problems.

(4) StubNotFoundException

This exception may be raised if the valid stub class is not available or not generated at the time of creating the remote service object at the server app'n.

NOTE Whenever the RemoteServiceObject is created at the server app'n then the stub, skel objects are automatically created at the server along with the server address details i.e. server host name, port no.

### (5) NoSuchObjectException

This exception may be raised if an attempt is made to invoke the method onto the RemoteService object which is already Garbage collected by the Garbage Collector at the server

### (6) MarshallException & UnMarshallException

This Exception is raised at the time of sending the method call from the client side to the server side, unmarshall exception is raised at the time of sending the returning value from the server side to client side

No Incomplete

No Incomplete

Set

24

\*

22-5

92-5-09

## class path settings for RMI app's

We should set the following system environment variables to compile & execute the RMI app's.

Set path = D:\beal\jdk141-03\bin;.;

Set classpath = D:\beal\jdk141-03\lib\tools.jar;

D:\oracle\product\10.2.0\db\_1\jdbc\lib\classes12.jar;.;

Set JAVA\_HOME = D:\beal\jdk141-03

Note '.' is for current directory class path setting

24-5-09

## parameter passing in RMI

- \* The following types of parameters can be passed in RMI app's.

### (1) primitive values

If the parameter type is primitive like int, float the RMI architecture will pass the parameter by value.

- \* The copy of primitive data type is passed from client JVM to server JVM or from server JVM to client JVM

- \* These values are passed in machine independent format so that the JVM running at diff plat forms can communicate each other

## 2. Serializable Objects

→

If the parameter type is an object implemented with Serializable interface then the object is passed by the value.

- \* A copy of the object is passed b/n the client JVM & the server JVM.
- \* The RMI architecture uses the technique called object serialization at sending side to transform the object into a stream of bytes that can be sent through the NW.
- \* Similarly the RMI architecture uses the technique called object Deserialization at receiving side to retransform the stream of bytes into original object & store it in the receiver JVM.

- \* If any changes are made in the received object at the destination JVM then these changes are not effected to the object of the source JVM.

## 3) Remote objects :

If the parameter type is an object implemented with Remote interface then the reference of the object is passed b/n the client & server in the form of stub object.

- \* If any changes are made at the received object destination through the reference of the Remote object then these changes are automatically effected to the Remote object available at the source JVM.

1.  
2.  
3.  
4.  
5.  
6.  
(1)

(2)

→ The following practical ex: demonstrate how the serialized object can be passed from server to the client app'n when the business logic method of the Remote service object is called.

rmiSerial object → Name of the app'n folder  
↓

- (1) UserDetailsSerialIface.java
- (2) UserDetailsSerialImpl.java
- (3) MyRemoteIface.java
- (4) MyRemoteImpl.java
- (5) MyRemoteRegistry.java
- (6) TestClient.java

### (1) UserDetailsSerialIface.java

This is the userdefined serializable interface with a set of abstract funs

```
import java.io.*;
public interface UserDetailsSerialIface extends Serializable
{
 public void setName(String name);
 public String getName();
}
```

### (2) UserDetailsSerialImpl.java

This is the implementation class for the userdefined serializable interface.

```
public class UserDetailsSerialImpl implements (4)
 UserDetailsSerialItface
{
 String name;
 public void setName(String s)
 {
 name = s;
 }
 public String getName()
 {
 return name;
 }
}
```

3) MyRemoteItface.java  
This is the userdefined Remote interface  
with a set of abstract business logic methods

```
import java.rmi.*;
public interface MyRemoteItface extends Remote
{
 public UserDetailsSerialItface get UserDetailsSerialObj
 -ect() throws RemoteException;
 public String getUserName() throws RemoteException;
}
```

#### (4) MyRemoteImpl.java

This is the implementation class for the userdefined Remote interface.

```
import java.rmi.*;
 " " " .Server.*;
public class MyRemoteImpl extends UnicastRemoteObject
 implements MyRemoteInterface
```

{

```
public UserDetailsSerialInterface udsi;
```

This is the attribute of this class.

```
public MyRemoteImpl() throws RemoteException
```

{

```
 udsi = new UserDetailsSerialImpl();
```

```
 udsi.setName("RAMESH");
```

{

This is the constructor of this implementation class where an object of UserDetailsSerialImpl class is created.

```
public UserDetailsSerialInterface getUserDetailsSerialObject()
```

{

```
 return udsi;
```

{

```
public String getUsername()
```

{

```
 return udsi.getName();
```

{

## 5) MyRemoteRegistry.java

This is the server app'n ie. Registry app'n where RemoteService object is created & it is registered into RMI Registry.

```
import java.rmi.*;
public class MyRemoteRegistry
{
 p.s.v.m (String args[]) throws Exception
 {
 MyRemoteImpl mri = new MyRemoteImpl();
 Naming.bind ("rmi://localhost:SerialObject", mri);
 S.O.P ("Remote object binded into RMI Registry");
 }
}
```

## 6) TestClient.java

This is the Client side app'n to invoke Business logic methods available in the RemoteService object with the help of rmi registry.

```
import java.rmi.*;
public class TestClient
{
 p.s.v.m (String args[]) throws Exception
 {
 MyRemoteInterface mri = (MyRemoteInterface) Naming.lookup
 ("rmi://localhost:SerialObject");
 }
}
```

```
UserDetailsSerialIface client-udsI = mri.getUserData
 .getSerialObject();

S.O.P("Client side object username:" +
 client-udsI.getName());

S.O.P("Server side object user name:" +
 mri.getUserName());

client-udsI.setName("Suresh");
S.O.P("After changing new name");
S.O.P("Client side object user name" +
 client-udsI.getName());

S.O.P("Server side object user name" +
 mri.getUserName());
```

{

}

compile:>javac \*.java  
>rmic MyRemoteImpl  
>start rmiregistry (minimize)  
>java MyRemoteRegistry  
>java TestClient

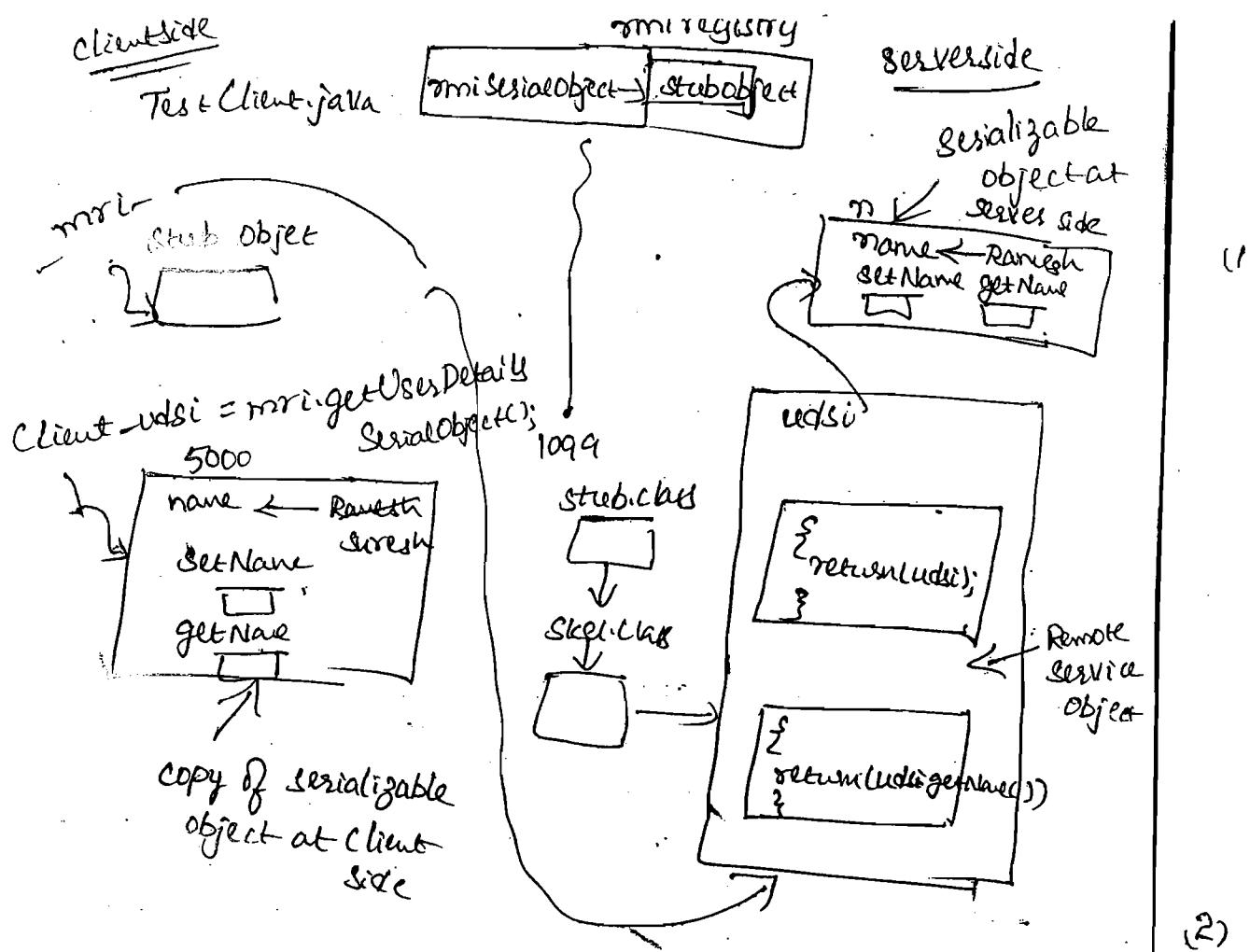
Client side object Username: Ramesh

Server " " " "

After changing the new name

Client side object username : Suresh

Server " " " " : Ramesh



25-5-09

→ The following practical ex: demonstrate how to pass the remote interface objects reference from the server side to the client side when the Business logic method of the remote service object is called.

rmiremoteobject



- (1) UserDetailsRemoteIface.java
- (2) UserDetailsRemoteImpl.java
- (3) MyRemoteIface.java
- (4) MyRemoteImpl.java

5) MyRemoteRegistry.java

6) TestClient.java

(1) UserDetailsRemoteIface.java

This is the userdefined Remote interface  
with a set of abstract funs

```
import java.rmi.*;
public interface UserDetailsRemoteIface extends Remote
{
 public void seeName(String name) throws RemoteException;
 public String getName() throws RemoteException;
}
```

(2) UserDetailsRemoteImpl.java

This is the implementation class for the  
above userdefined Remote Interface.

```
import java.rmi.*;
public class UserDetailsRemoteImpl extends UnicastRemoteObject implements UserDetailsRemoteIface
{
}
```

String name;

UserDetailsRemoteImpl() throws RemoteException

{

```
public void setName (String s)
{
 name = s;
}
public String getName ()
{
 return (name);
}
```

### ③) MyRemoteIface.java

This is the user defined Remote Interface  
for the Remote service object with a set of abstract  
business logic methods

```
import java.rmi.*;
public interface MyRemoteIface extends Remote
{
 public UserDetailRemoteIface getUserDetailsRemoteObj
 throws RemoteException;
 public String getUserName () throws RemoteException;
}
```

### ④) MyRemoteImpl.java

This is the implementation class  
for the above userdefined Remote Interface for the  
remote service object

```
import java.rmi.*;
" " " " .server.*;

public class MyRemoteImpl extends UnicastRemoteObject
 implements MyRemoteIface

{
 public UserDetailsRemoteIface udri;
 // This is the attribute of the class

 public MyRemoteImpl() throws RemoteException
 {
 udri = new UserDetailsRemoteImpl();
 udri.setName("Rameet");
 }

 // This is the constructor of this implementation
 // class where there is another remote interface object
 // is created
 public UserDetailsRemoteIface getUserDetailsRemoteObject()
 {
 return udri;
 }

 public String getName() throws RemoteException
 {
 return udri.getName();
 }
}
```

### (5) MyRemoteRegistry.java

This is the server app'n ie. registry app'n where we create the Remote Service object & registers this object into RMI Registry.

```
import java.rmi.*;
public class MyRemoteRegistry
{
 p.s.v.m (String args[]) throws Exception
 {
 MyRemoteImpl mri = new MyRemoteImpl();
 Naming.bind ("rmi remoteobject", mri);
 S.o.P ("Remote object binded into RMI Registry");
 }
}
```

### (6) TestClient.java

This is the client side app'n where we invoke the business logic methods of the RemoteService object

```
import java.rmi.*;
public class TestClient
{
 p.s.v.m (String args[]) throws Exception
 {
 MyRemoteIface mri = (MyRemoteIface) Naming.
 lookup ("rmi remoteobject");
 }
}
```

```
UserDetailsRemoteInterface client_udri = rmii.getUserDetails
 RemoteObject();
```

```
S.O.P(" user name at server side object in through
client-side reference" + client_udri.getName());
```

```
S.O.P(" Username at server side object in through
client side remote service object reference" +
rmii.getUserName());
```

```
Client_udri.setName("Sureet");
```

```
S.O.P(" After changing the new name");
```

```
S.O.P(" user name at server side object in through
client-side reference" + Client_udri.getName());
```

```
S.O.P(" username at serverside object in through
client side remote service object reference" +
rmii.getUserName());
```

```
S.O.P(" bye bye");
```

```
}
```

```
}
```

```
> javac *.java
```

```
> rmic UserDetailsRemoteImpl (Stub & skeleton classes generated)
```

```
> rmic MyRemoteImpl (" " "
```

```
> start rmiregistry (minimize) (Next execute the server appn)
```

```
> java MyRemoteRegistry (Op: Remote object binded into
RMI Registry)
```

> jalla TestClient

OPP uses name at server side object through client  
side reference Ramesh

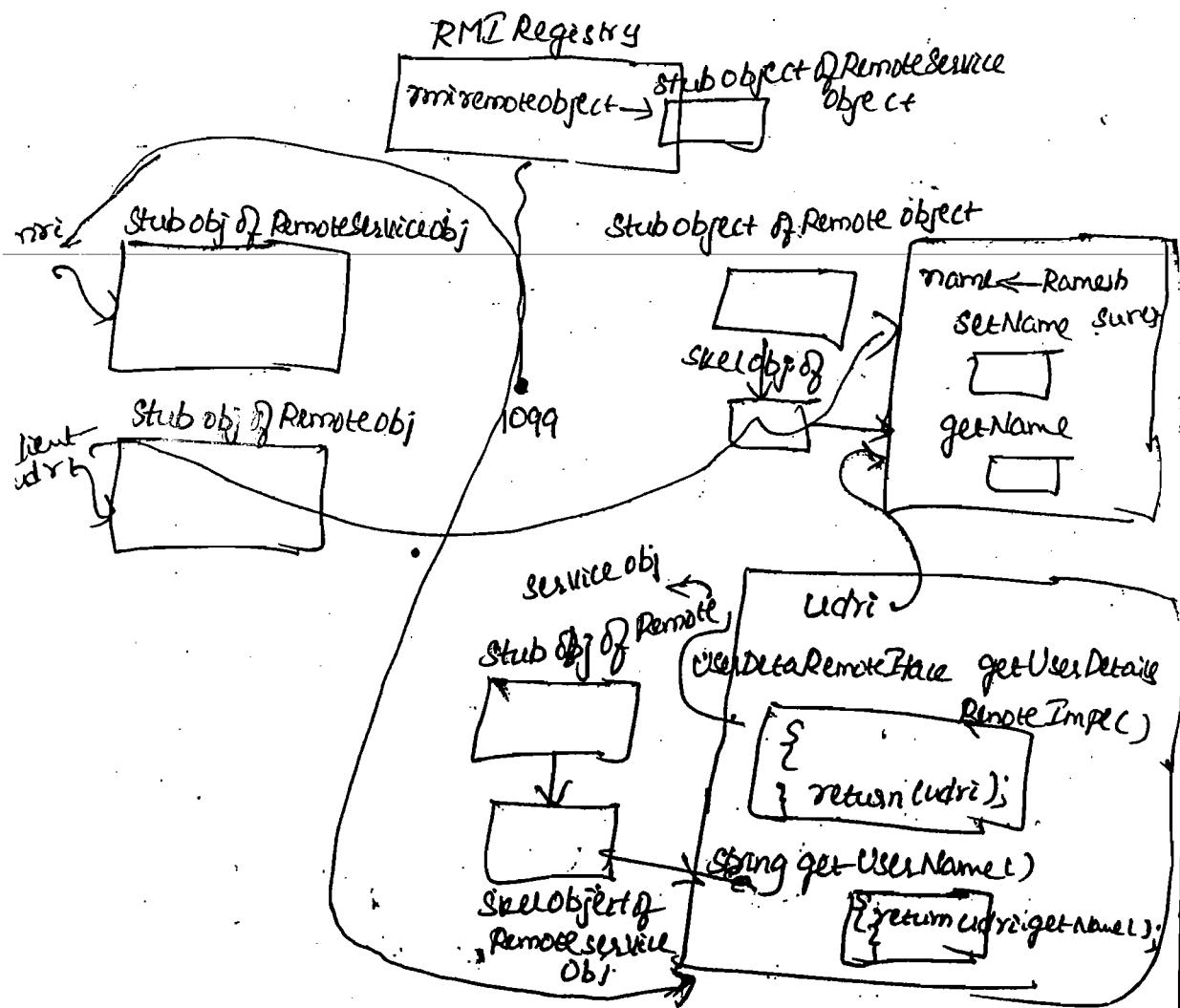
26-

side remote service object reference is Rameeh

After changing <sup>new</sup> name

Username at serverside object  
through client side reference → suresh

Username at server side object  
through client side reference is Suresh



26-5-09

→ The following practical ex: demonstrate how to pass the RemoteInterface object from client side to server side.

rmicallback



- (1) MyRemoteIface.java
  - (2) MyRemoteImpl.java
  - (3) MyRemoteRegistry.java
  - (4) ClientSideRemoteIface.java
  - (5) ClientSideRemoteImpl.java
  - (6) TestClient.java
- } serverside java files      } clientside java files

### (1) MyRemoteIface.java

This is the user defined remote interface at serverside with a set of abstract business logic methods.

```
import java.rmi.*;
public interface MyRemoteIface extends Remote
{
 public void callBack(ClientSideRemoteIface csri)
 throws RemoteException;
}
```

### (2) MyRemoteImpl.java

This is the implementation class at server side for the above userdefined Remote Interface

```

import java.rmi.*;
" " " .Server.*;

public class MyRemoteImpl extends UnicastRemoteObject
 implements MyRemoteIface

{
 public MyRemoteImpl() throws RemoteException
 {
 }

 public void callBack(ClientSideRemoteIface csri)
 throws RemoteException
 {
 System.out.println("This is call back method");
 csri.setMessage("The client .. This is server message");
 }
}

```

### (3) MyRemoteRegistry.java

This is the server app'n i.e. Registry app'n to create the RemoteService object & to bind into RMI Registry.

```

import java.rmi.*;
public class MyRemoteRegistry
{
 public MyRemoteRegistry(String args[]) throws RemoteException
 {
 }
}

```

5.

```
MyRemoteImpl mri = new MyRemoteImpl();
Naming.bind ("rmicallback", mri);
S.O.P("RemoteObject binded into RMI Registry");
```

(4) ClientSideRemoteIface.java  
This is the user defined RemoteInterface  
at client side with a set of abstract methods

```
import java.rmi.*;
public interface ClientSideRemoteIface extends Remote
{
 public void setMessage (String s) throws RemoteException;
 public String getMessage () throws
}
```

(5) ClientSideRemoteImpl.java  
This is the implementation class  
at client side for the above user defined Remote  
Interface.

```
import java.rmi.*;
" " " server.*;
public class ClientSideRemoteImpl extends
UnicastRemoteObject implements
ClientSideRemoteIface
```

```
String message;
ClientSideRemoteImpl(String s) throws RemoteException
{
 message = s;
}
public void setMessage(String s)
{
 message = s;
}
public String getMessage()
{
 return message;
}
```

### (5) Test Client.java

This is the client side app'n where we create one Remote Interface object at Client Side & we pass the reference of this object along with the Business logic method that we call of the Remote Service object available at Server side.

```
import java.rmi.*;
public class TestClient
{
 p.s.v.m(String arg[]) throws Exception
{}
```

```
MyRemoteIface mri = (MyRemoteIface) Naming.lookup
("mirecallBack");
```

```
ClientSideRemoteImpl client-csr = new
```

```
ClientSideRemoteImpl ("Hello Server...
this is client msg");
```

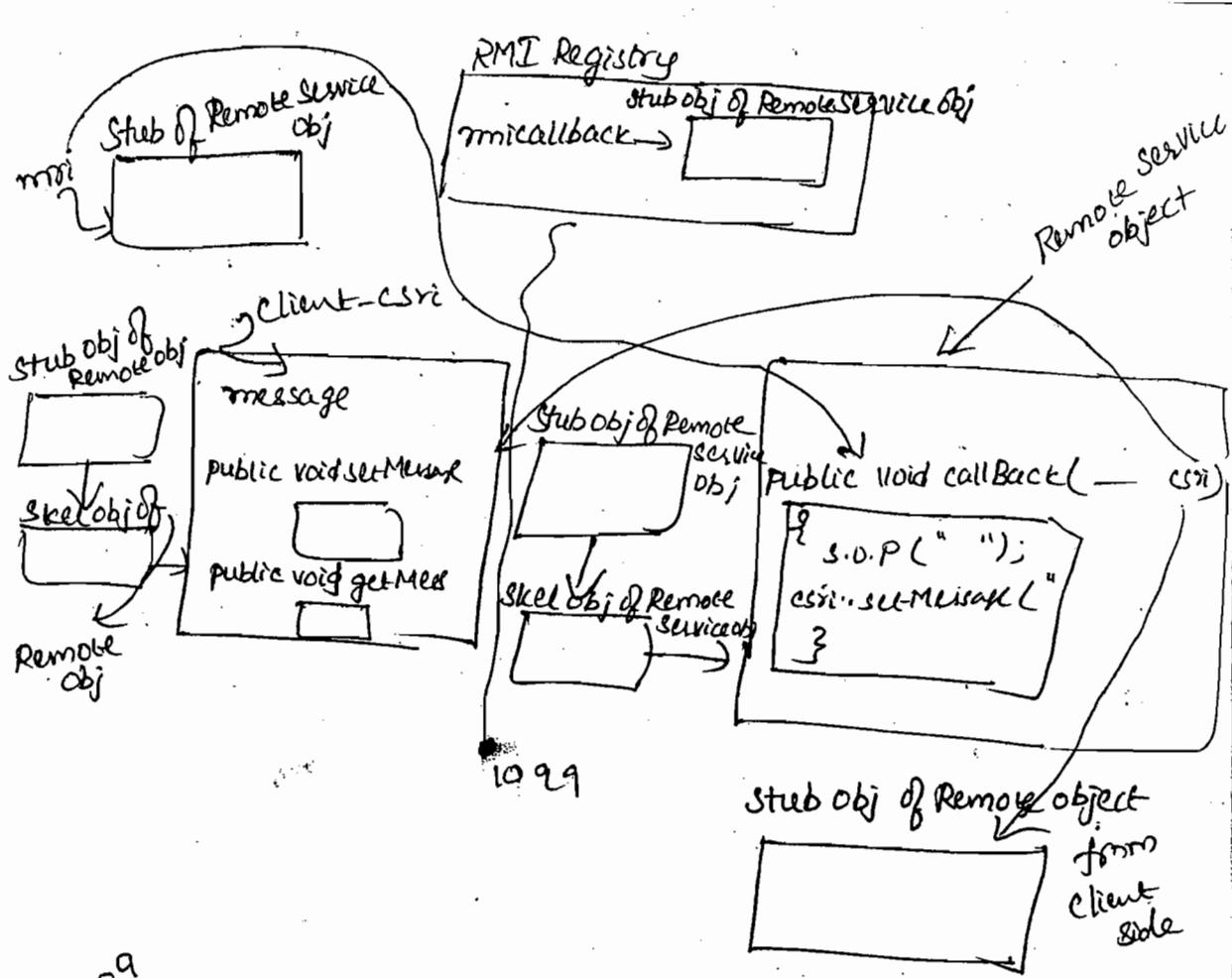
```
s.o.p ("msg in the client side Remote object in Before
call back is ... " + client-csr.getMessage());
mri.callBack (client-csr);
```

```
s.o.p ("msg in the client side Remote object in After
call back is ... " + client-csr.getMessage());
```

```
}
```

```
}
```

```
> javac *.java
> rmic ClientSideRemoteImpl (Stub & Skele classes are
generated)
> rmic MyRemoteImpl (" ")
> start RMIRegistry (minimize)
> java MyRemoteRegistry (obj: Remote service object
binded into RMI Registry)
> java TestClient
```



27-5-09  
 \* The following practical ex: demonstrate how to perform database operations like insert, delete, update into a database table through the Business logic methods of Remote service obj's

rmidbase



- (1) MyRemoteDatabaseInterface.java
- (2) MyRemoteDatabaseImpl.java
- (3) MyRemoteDatabaseRegistry.java
- (4) TestDatabaseClient.java

## 1) MyRemoteDatabaseInterface.java

This is the userdefined Remote interface with a set of abstract business logic methods.

```
import java.rmi.*;
public interface MyRemoteDatabaseInterface extends Remote
{
 public String insert(int eno, String ename,
 float esal) throws RemoteException;
 public String delete(int eno) throws RemoteException;
 public String update(int eno, float amt) throws
 RemoteException;
}
```

## 2) MyRemoteDatabaseImpl.java

This is the implementation class for the above userdefined Remote interface.

```
import java.rmi.server.*;
import java.sql.*;
public class MyRemoteDatabaseImpl extends
 UnicastRemoteObject implements
 MyRemoteDatabaseInterface, Unreferenced
{
```

Where Unreferenced is the predefined interface available in java.rmi package.

```
Connection con;
public final String driverClassName = "oracle.jdbc.driver.OracleDriver";
This is type-4 driver which is also called as
thin driver.

public final String url = "jdbc:oracle:thin:
@localhost:1521:services";
public final String user = "scott";
public final String password = "tiger";
public MyRemoteDatabaseImpl() throws RemoteException
{
 try
 {
 Class.forName(driverClassName);
 con = DriverManager.getConnection(url, user, password);
 System.out.println("connection established");
 }
 catch(Exception e)
 {
 }
}

public String insert(int eno, String ename, float esal)
{
 System.out.println("This is insert business logic method");
}
```

try

{

PreparedStatement pstmt = con.prepareStatement("insert into employee values (?, ?, ?)");

pstmt.setInt(1, eno);

pstmt.setString(2, ename);

pstmt.setFloat(3, esal);

if (pstmt.executeUpdate() == 1)

return "success";

}

catch (Exception e)

{

}

return "fail";

}

public String delete(int eno)

{

s.o.p ("This is delete business logic method");

try

{

PreparedStatement pstmt = con.prepareStatement

("delete from employee where eno=?");

```
stmt.setInt(1, eno);
if(pstmt.executeUpdate() == 1)
 return "success";
}
catch(Exception e)
{
}
return "fail";
```

```
public String update(int eno, float amt)
{
 System.out.println("This is update business logic method");
 try
 {
```

```
 PreparedStatement pstmt = con.prepareStatement(
 "update employee set esal=esal+?
 where eno=?");
```

```
pstmt.setInt(2, eno);
pstmt.setFloat(1, amt);
if(pstmt.executeUpdate() == 1)
 return "success";
```

```

}
catch(Exception e)
{
}
```

```
 return "fail";
```

```
}
```

```
public void unreferenced()
```

```
{
```

This is the implemented method of unreferenced interface, this method is automatically executed by the JVM before destroying this Remote Service object by the garbage collector.

```
try
```

```
{
```

```
 if (con!=null)
```

```
 con.close();
```

```
}
```

```
catch(Exception e)
```

```
{
```

```
}
```

```
}
```

```
}
```

### 3) MyRemoteDatabaseRegistry.java

This is the server app'n ie. Registry app'n where we create Remote Service obj & bind this object into RMI Registry.

```
import java.rmi.*;
public class MyRemoteDatabaseRegistry
{
 public void (String args[]) throws Exception
 {
 MyRemoteDatabaseImpl mrdi = new
 MyRemoteDatabaseImpl();
 Naming.bind ("rmidbase", mrdi);
 System.out.println ("Remote object binded into RMI Registry");
 }
}
```

### 4) TestDatabaseClient.java

This is the client side app'n where we invoke the business logic methods of Remote service object

```
import java.rmi.*;
import " " . io.*;
public class TestDatabaseClient
{
 public void (String args[]) throws Exception
 {
 Remote r = Naming.lookup ("rmidbase");
 MyRemoteDatabaseInterface *mrdi = (MyRemoteDatabase
 Interface) r;
 }
}
```

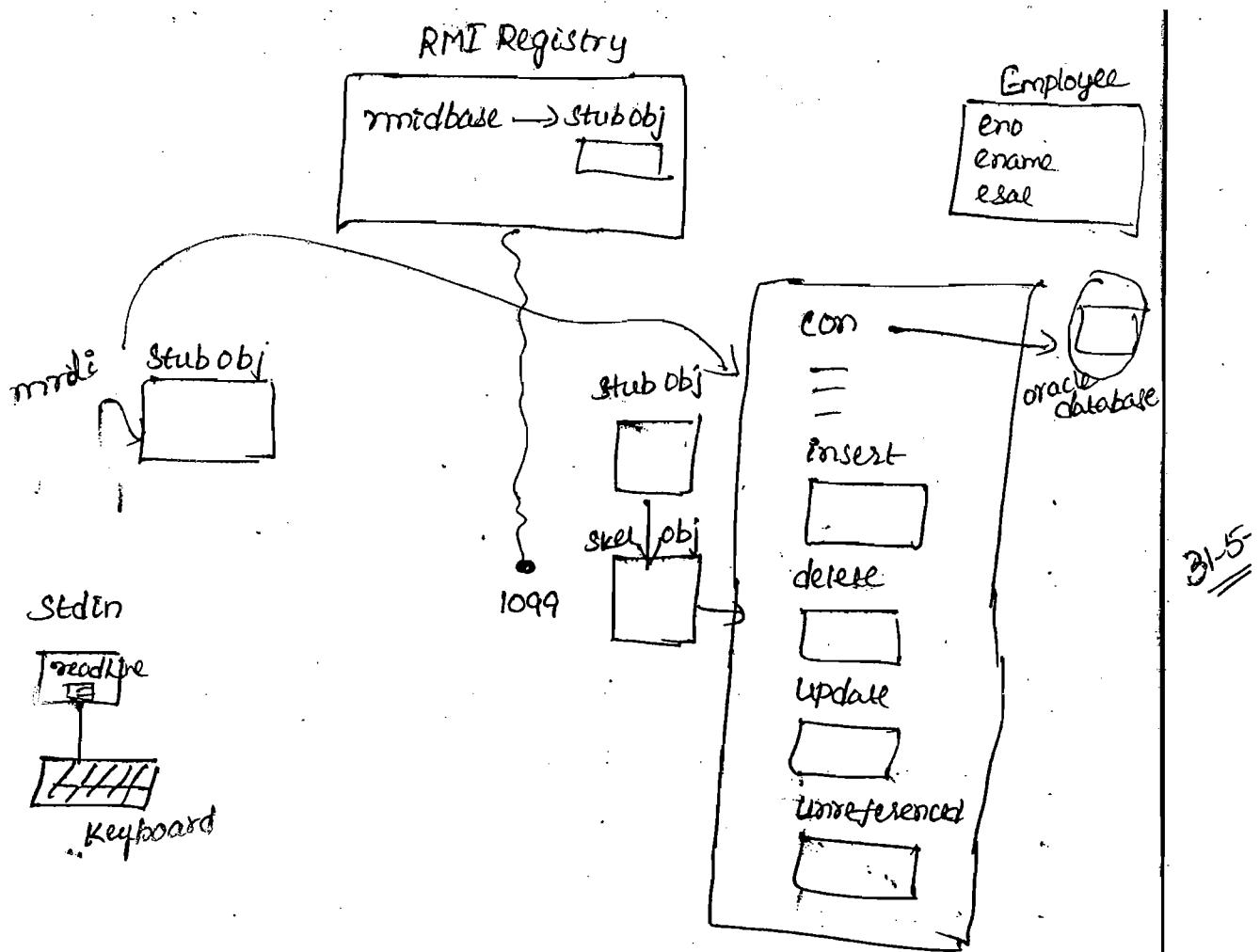
```

DataInputStream stdin = new DataInputStream(System.in);
S.O.P("Enter employee no:");
int eno = Integer.parseInt(stdin.readLine());
S.O.P("Enter employee Name:");
String ename = stdin.readLine();
S.O.P("Enter employee salary");
float esal = Float.parseFloat(stdin.readLine());
String status = mrdi.insert(eno, ename, esal);
S.O.P(status);
S.O.P("Enter employee no. to be deleted");
eno = Integer.parseInt(stdin.readLine());
status = mrdi.delete(eno);
S.O.P(status);
S.O.P("Enter employee no. to be updated");
eno = Integer.parseInt(stdin.readLine());
S.O.P("Enter bonus amount");
float amt = Float.parseFloat(stdin.readLine());
status = mri.update(eno, amt);
S.O.P(status);
S.O.P("Bye Bye");
}
}

```

A diagram illustrating the connection between a Remote Object and the RMI Registry. An arrow points from the text "Remote Obj" to the text "RMI Registry". Above this arrow, the text "OPP connection established" is written. Below the arrow, the text "Remote Obj bound into RMI Registry" is written. To the right of the arrow, there is a callout pointing to the "start(monitor)" text, containing the subtext "rmiregistry". At the bottom right, another callout points to the "stub & skeleton classes are generated" text, containing the subtext "Dynamic".

> javac \*.java  
 > rmic MyRemoteDatabaseImpl



after that client side app'n running

> java TestDatabaseClient

Enter Employee No: 101

" " Name: Ramasao  
" " sal : 12000

success

~~see~~  
Enter employee no to be deleted.

102

fail

Enter " " updated:

600

success

Bye, Bye

~~31-5-09~~  
EJB 2.0

Enterprise Java Beans

sequence of concepts

- (1) Introduction of EJB
- (2) EJB container
- (3) Types of EJB components
- (4) procedure to develop the session beans at the server
- (5) " " access " bean components from the client app'n
- (6) Java Naming Directory Interface ie JNDI
- (7) life cycle sequence of session beans at the server
- (8) connection pooling
- (9) Bean to bean c<sup>n</sup>.

1) Introduction of EJB

- \* some of the java technologies that are used to design the server side components are servlets & JSP.

JNDI Naming Registry

Server-side

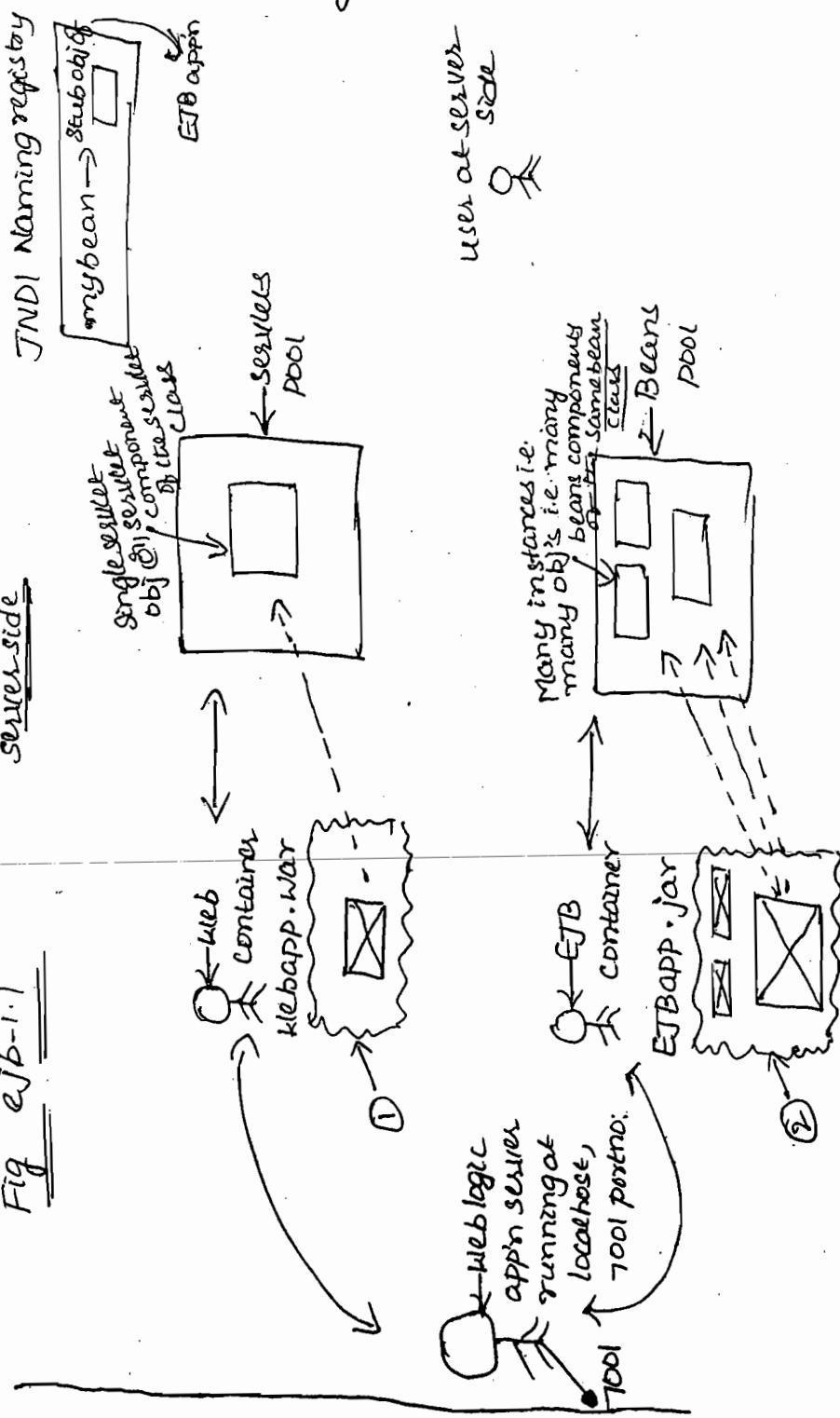
Fig ejb-1.1

Client-side

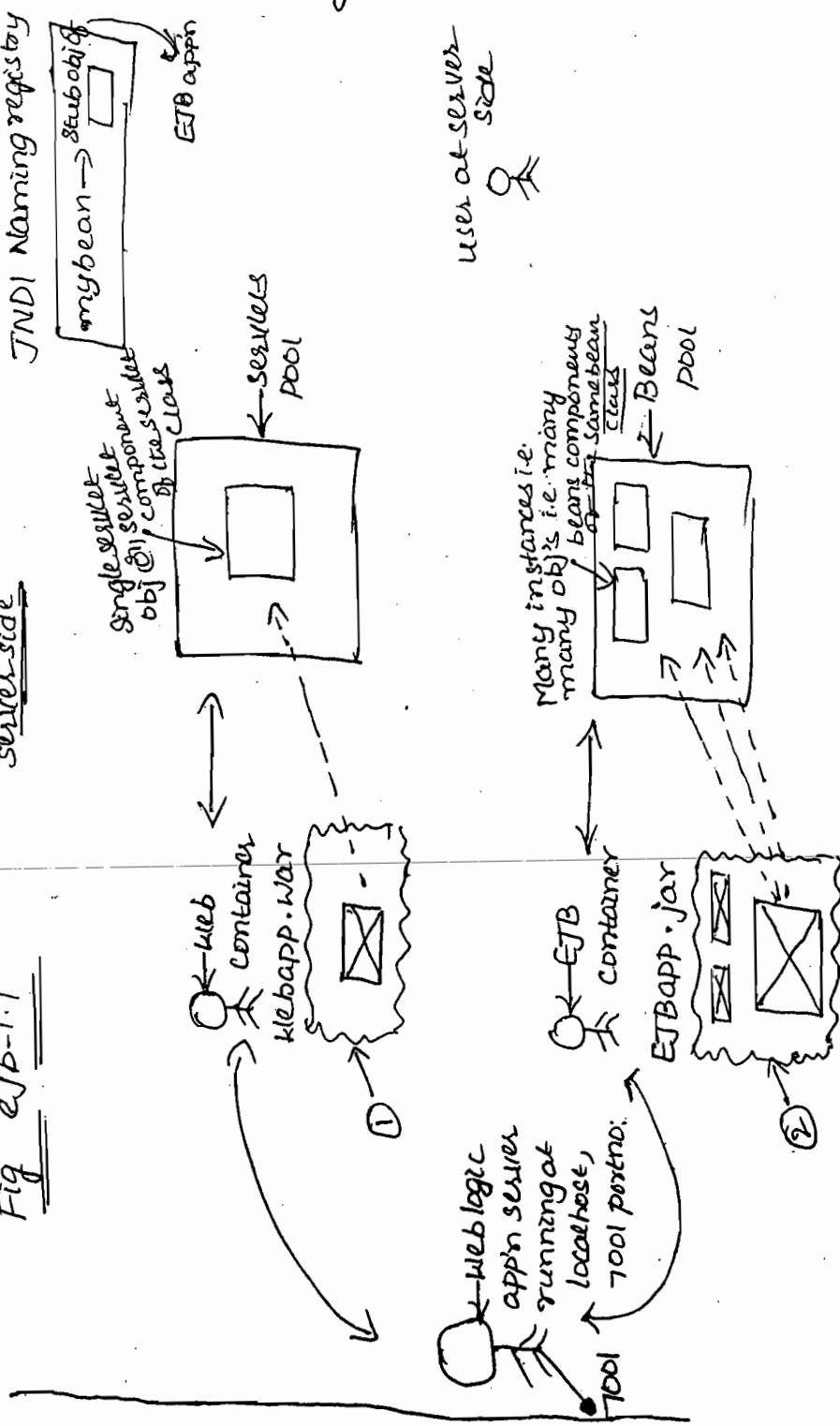
- \* These technologies are used to create & execute the web components such as Servlet in structured execution environment called the Web container at the Webserver such as tomcat server that will provide some high level services to web components.
- \* Enterprise Java Beans is another technology used to design the server side components for the implementation of Business logic methods.
- \* This EJB Technology will allow the component designer at the app'n server such as Weblogic without worrying about the system level services like transactions, security and so on.
- \* All the system level services are ~~per~~ automatically provided by the app'n server. so that the beans designer can directly use them acc to app'n req's.
- \* The beans designer has to concentrate more on the business logic func rather than the system level programming which is the powerful feature of EJB Technology.
- \* EJB app'n is the just collection of java classes defined by the user i.e. designer at serverside by following some predefined rules & the XML files bundled into a single unit & deployed at the app'n server.
- \* The diff b/w the Web server & app'n server is the Webserver will have only Webcontainer, whereas the app'n server " " " Webcontainer, EJB container, JNDI naming registry along with some high level stuff

Fig ejb-1.1

client-side



→ The following diagram demo the scenario of app'n server



① This is the Web app'n designed by the user with the Servlet classes with the lifecycle methods deployed at the Web container of the Web logic app'n server.

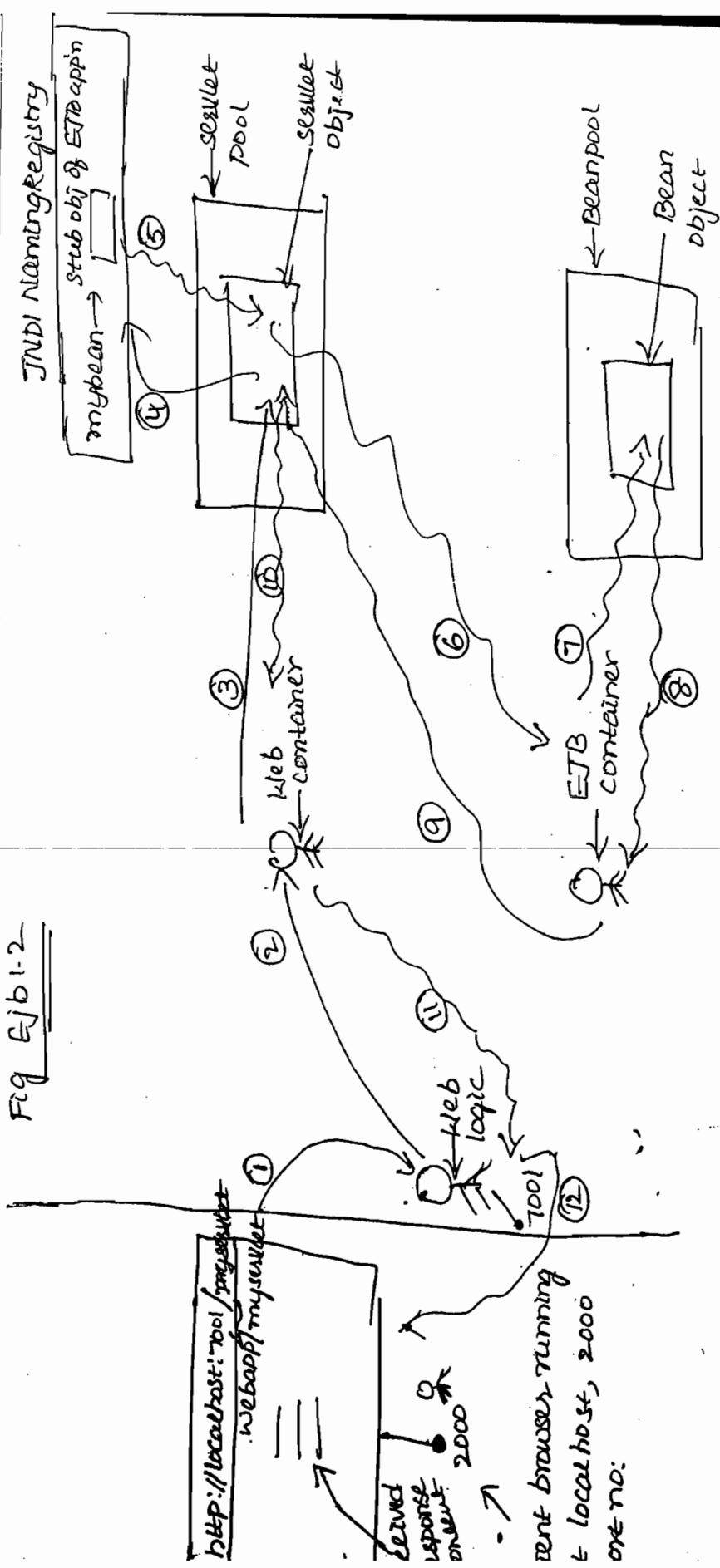
② This is the EJB app'n designed by the user at server side with a set of interfaces, a single Bean class by implementing some business logic methods & to app'n req's deployed at the EJB container with some JNDI name such as mybean at the weblogic app'n server.

- \* An EJB component will run under the EJB container, the EJB container will run under the app'n server by using the services of the app'n server.
- \* EJB components are developed by using the j2ee technology so they can run on any operating system.
- \* EJB components are the specifications for the serverside, " within J2EE environment, so any company or any organization can implement by following the same specifications & use them as to organizational req's
- \* EJB components can work with any type of client which means they can be used in services or JSP pages to provide the accessibility to the WebClients or they can be used directly in the Client-side Java apps.
- \* The following diagram demonstrate how the Webclients can access the EJB components through the web components such as services

JNDI Naming Registry

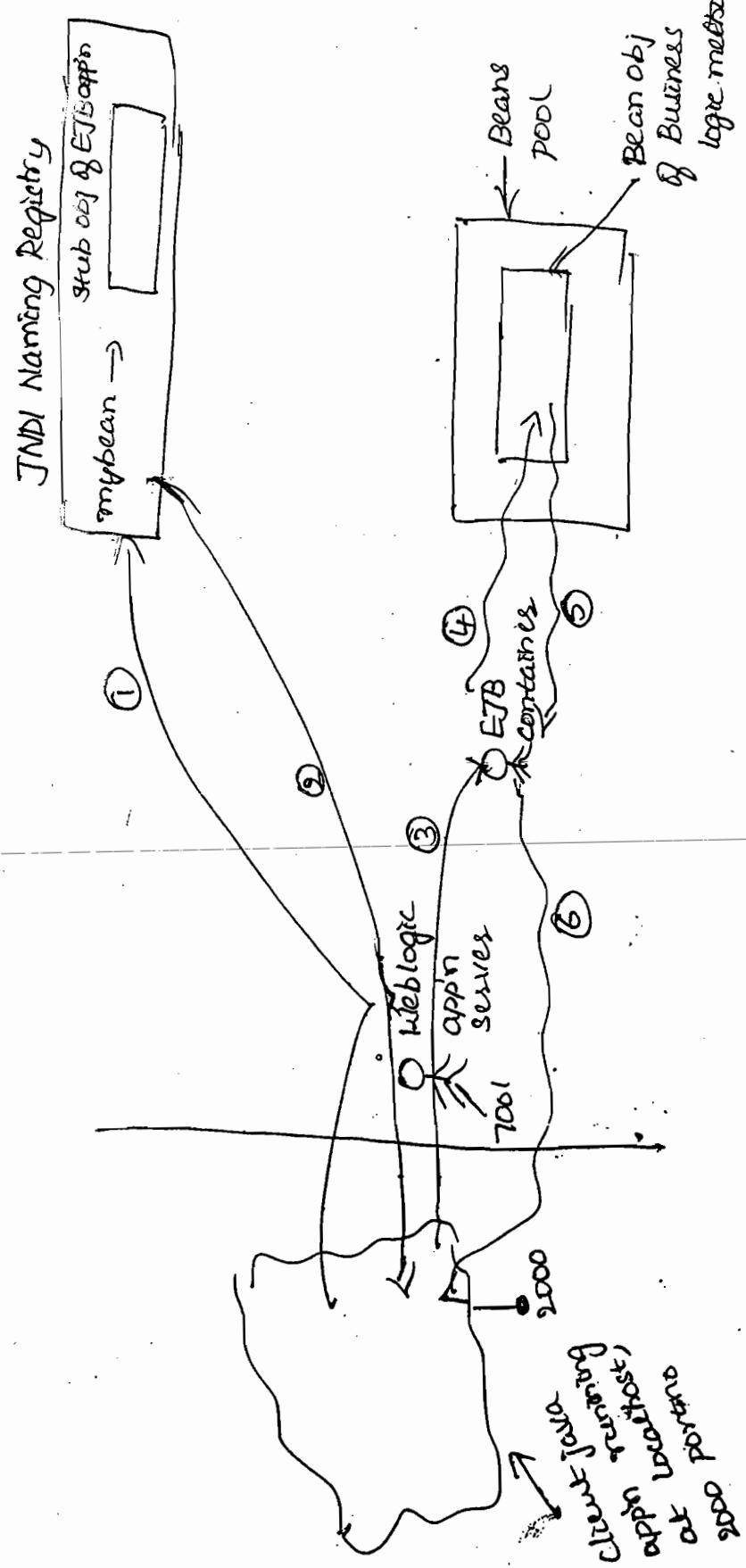
Fig Ejb 1-2

Fig Ejb 1-2



JNDI Naming Registry

→ The foll diagram demo how the java client can directly access the EJB components



\* The diff b/w java beans & enterprise java beans is the java beans are used for general purpose components whereas EJB components are the specialized components deployed in J2EE environment which are used for business logic funs.

## (2) EJB CONTAINER

- \* The EJB container is the execution environment for the EJB components which means the EJB container will provide all the required services to the EJB components.
- \* The EJB container will run in the app'n server, which means the app'n services will provide all the required services to the EJB container & other containers.
- \* Some of the services provided by the EJB container to the EJB components are Connection pooling, Data persistence, Data transaction and so on.
- \* EJB container will provide the service called Connection pooling to manage the established connection to the DB.
- \* The EJB container will provide the service called Data persistence in the DB, so that the beans designer need not write any jdbc code for the data persistence i.e. data storage in the database.
- \* The Data transactions are automatically managed by the EJB container without any coding written by the beans designer.

- \* Whenever the client make a request to the EJB component at the server, then the EJB container automatically provide required services to the EJB components.
- \* In other words, the EJB container will interact on every call made by the client to the EJB component so that it can provide all the required services to the EJB components.

~~1-8 or~~

### (3) Types of EJB components (or) Bean components

- \* The goal of EJB technology is to provide the standard component architecture for the creation & the usage of distributed object oriented business logic system.
- \* This goal can't be met with only one type of EJB components, so all the EJB components defined by the user at the server are categorized into 3 diff types depending on how the business logic func are implemented on the bean components i.e.

(i) Session Beans

(ii) Entity "

(iii) Message Driven Beans

- \* The session Beans are designed to provide the business logic func for the client

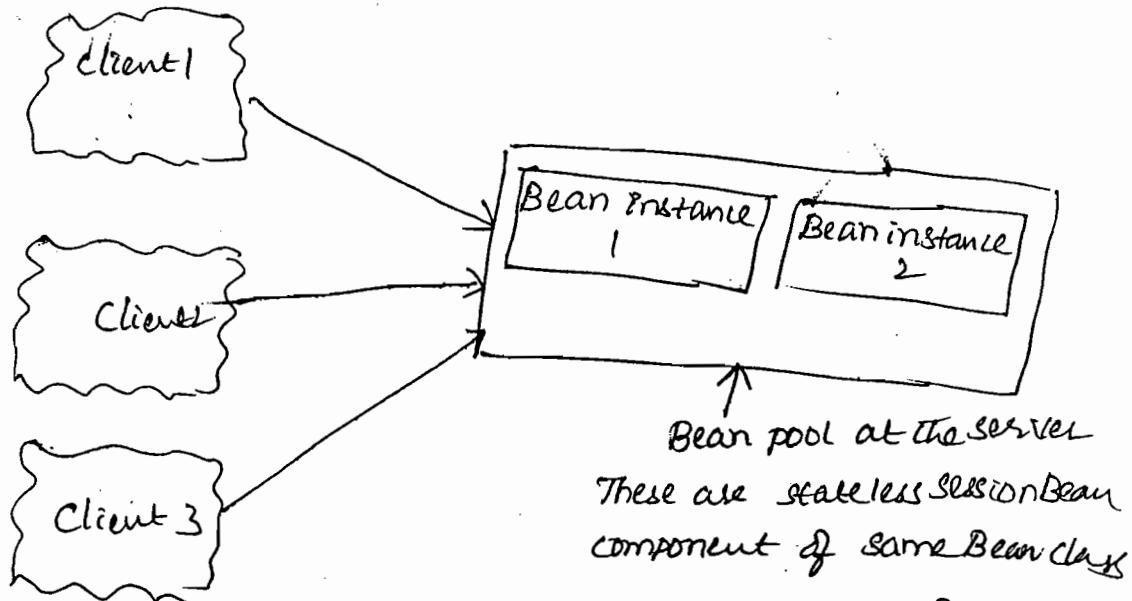
- \* The name session implies that the bean component is available at the server is some duration of time until the server is shut it down i.e. until the session is closed.

2

- \* There are two types of session beans can be defined by the user i.e. stateless session beans (or) stateful session beans acc to app'n req's.
- \* If the bean component is designed as stateless session bean then the EJB container can assign any bean component of the bean class from the bean pool to the requested client based on the following factors i.e.
  - (i) Work load at the server
  - (ii) Bean pool capacity
- \* If the bean component is designed as stateful session bean then the EJB container will assign one specific bean component to every requested client.
- \* In this stateful session Bean, the EJB container automatically maintain one unique ID for every requested client so that the clients bean component state is maintained at the server.
- \* In stateless bean, it is possible that the same bean component is assigned to the diff clients, but this is not possible in stateful bean.

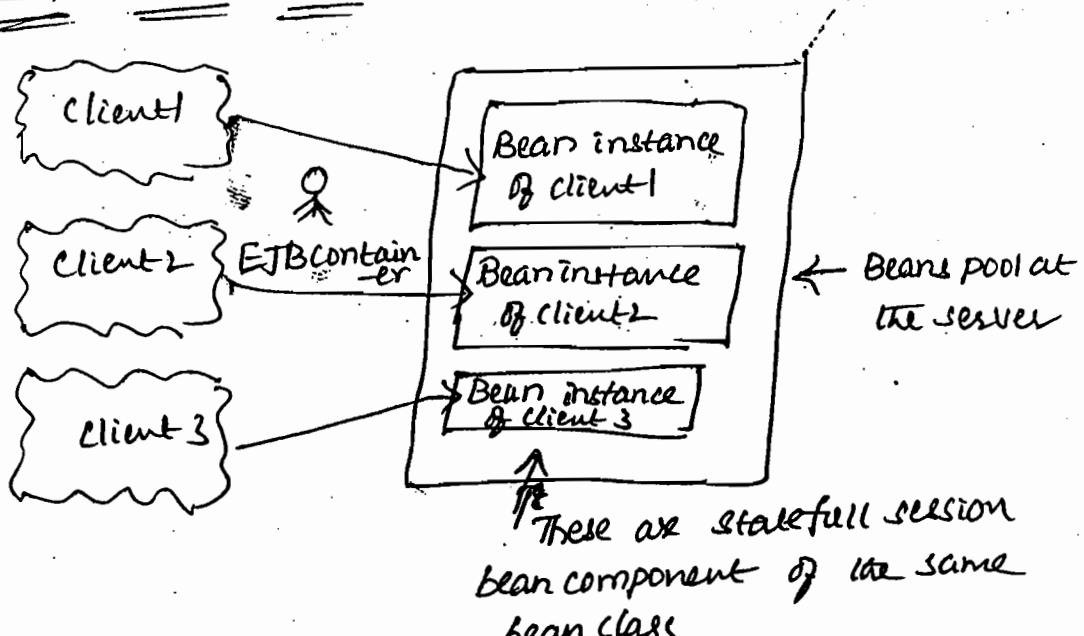
~~26-09~~  
→ The following diagram demonstrate diff b/w stateless & statefull session bean components

## Stateless Session Bean



Note The EJB container will assign any one of the bean component to the required client based on the workload at the server

## Statefull Session Bean



Note The EJB container will assign one specific bean component to every requested client.

- \* If the Bean component is designed to maintain the data persistence in the database ie storing the records & retrieving the records from the DB tables by using Java Persistence API ie JPA service provided by the server then that bean components are called as Entity Beans
- \* persistence is the ability of storing the data available in the java obj's automatically into the DB such as Oracle Database.
- \* If the Bean component is designed to send & receive the msg asynchronously from the app'n server with the help of java message service ie JMS service provided by the server then that bean components are called as MessageDrivenBeans.
- \* All the Session Bean components which are either stateless or statefull are called as entity beans & messageDriven beans depending on the funs performed by the respective bean component.

→ Procedure to Develop the SessionBeans at the server

- \* We should follow the following steps to develop the SessionBeans at the server ie.

- (1) We should define one public Remote interface for the bean class

→ This userdefined RemoteInterface should be extended with the predefined interface called EJBObject which is available in javax.ejb package.

→ We should include all the required abstract business logic methods in this RemoteInterface, all these methods should be declared as throws RemoteException.

(2) We should define one public Home interface for the EJB app'n.

→ This userdefined Home interface should be extended with the predefined interface called EJBHome which is available in javax.ejb package.

→ We should include one create method without any arguments if it is stateless bean or with arguments if it is statefull bean acc to app'n req's, which means this create method in the HomeInterface will decide the bean component is either stateless or statefull bean component.

→ The returning type of this create method must be declared as the above userdefined Remote Interface type.

→ This create method must be declared as throws CreateException, RemoteException.

(3) We should define one public bean implementation class

→ This Bean implementation class should be implemented with the predefined interface called SessionBean which is available in javax.EJB package.

→ We should define one public default constructor in this implementation class.

→ We should implement the following EJBContainer life cycle methods of SessionBean interface in this implementation class i.e.

- (i) public void setSessionContext(SessionContext sc);
- (ii) public void ejbCreate();
- (iii) public void ejbRemove();
- (iv) public void ejbActivate();
- (v) public void ejbPassivate();

→ All the above 5 abstract methods of SessionBean interface must be implemented in the bean class otherwise the bean class will become an abstract class.

→ We should implement all the business logic methods of the above userdefined Remote interface in this bean class acc to appn req's

(4) We should provide the description about the bean class to the EJB container which is called as Deployment description.

→ This description about the bean class should be provided to the EJBcontainer regarding the name of RemoteInterface, the name of HomeInterface, the name of the Bean class, the name of JNDI Registry and so on so that EJB container automatically provide stub & skele classes for the respective Remote & Home interfaces.

→ All this description should be provided by writing some XML tags in some XML files. Normally these XML files are automatically generated by the appn server itself with one folder called as META-INF, which means the user need not write this XML code.

5) We should prepare one jar file for the EJB app'n along with the above class files & the META-INF folder

6) We should deploy the jar file of the EJB app'n at the app'n server such as Weblogic server along with some JNDI name such as hellobean

→ Whenever we deploy the EJB app'n, the EJB container automatically generate stub,skel classes for the respective Remote,Home interfaces.

→ The EJB container automatically create the Stub,skel obj's of the Home interface for the deployed app'n, the stub obj of the Home interface is binded into the JNDI Registry with <sup>the</sup> specified JNDI name i.e. hellobean.

→ so far the stub,skel obj's of RemoteInterface is not yet created by the EJB container

→ Now any client from the client app'n can look into the JNDI registry of the Weblogic server & get the copy of the stub obj of the Home interface of the EJB app'n.

→ procedure to access the Session bean component from the Client app'n

\* We should follow the following steps to access the SessionBean component available at Weblogic app'n server i.e.

(1) We should store the Weblogic JNDI properties into one file such as jndiproperties.properties

with the following Weblogic vendor dependent java code

```
java.naming.factory.initial = weblogic.jndi.WLInitialCon
-textFactory
```

```
java.naming.provider.url = t3://localhost:7001
```

- 2) We should load this weblogic JNDI properties into an obj of Properties class such as

```
Properties p = new Properties();
```

```
p.load(new FileInputStream("jndi.properties.properties"));
```

where properties is the predefined class available in `java.util` package.

- 3) We should create one InitialContent obj along with this properties obj to create the stub obj of JNDI Registry such as

```
InitialContext ic = new InitialContext(p);
```

→ This will create an obj of `InitialContext` class which is available in `javax.naming` package along with the respective `Properties` class object.

→ Now `ic` is referencing to the Stub obj of JNDI Registry where these are properties of `Weblogic JNDI Registry` are available.

- 4) We should get the copy of the stub obj of the `HomeInterface` of the EJB app in from the weblogic JNDI Registry into the client appn such as

```
MyBeanHomeIface mbhi = (MyBeanHomeIface) ic.lookup
("hellobean"),
```

→ Where MyBeanHomeInterface is the userdefined Home interface of the EJB app'n.

→ Where lookup method of InitialContext class will establish the connection to the Weblogic app'n server & get the copy of stub obj of HomeInterface of the EJB app'n available in JNDI registry of Weblogic server against to the JNDI name such as Hellobean into the client app'n.

→ Where Hellobean is the JNDI name given at the time of deploying the EJB app'n at the server

→ Now mbhi is referencing to the stub obj of the Home interface of the EJB app'n.

5) We should get the Stub obj of the RemoteInterface of the bean class from the EJB container into client app'n such as

MyBeanRemoteInterface mbri = mbhi.create();

Where MyBeanRemoteInterface is the userdefined RemoteInterface, where create is the method of the userdefined Home interface

→ When this create method is received from the clientside then the EJB container at the serverside automatically create the stub, stub obj's of the Remote interface & its copy of the stub obj of the Remote interface is sent to the client app'n.

\* Now mbri is referencing to the stub obj

of the RemoteInterface of the Bean class at clientside.

(b) We should call the required business logic method, through this stub obj of the RemoteInterface such as

String Receivedmsg = mbr.getMsg();

→ Here getMsg is the Business logic method of the Remote interface

→ When this method call is received by the EJB container at serverside then it will automatically create the bean obj of the bean class (or) Select existing bean obj from the bean pool, pass the request to the bean component, collect the response & send the response back to the requested client.

Note The skel obj's of home interface, Remote interfaces which are available at the serverside automatically communicate with the EJB container every time when the request is received from the client so that the EJB container will take the decision of creating the new bean obj of the bean class or selecting the existing bean obj available in the bean pool based on the workload at the server.

No Incomplete

3.609 The following practical ex: demonstrate the above procedure to create an EJB app'n at serverside & to access the bean component from the client app'n to invoke the business logic methods available in the Bean Component.

2.

Name of the app'n folder is → stsmainbean

↙ (stateless)



- (1) MyBeanRemoteInterface.java
- (2) MyBeanHomeInterface.java
- (3) MyBeanImpl.java
- (4) TestClient.java
- (5) JndiProperties.properties

Where 1,2,3 java files are serverside EJB app'n java files.

Where 1,2,4,5 are required java files at client side

### (1) MyBeanRemoteInterface.java

(3)

This is userdefined Remote Interface at server side with a set of abstract Business logic methods

import javax.ejb.\*;

|| java.rmi.\*;

public interface MyBeanRemoteInterface extends EJBObject

{

    public String getMessage() throws RemoteException;

    public int add(int p1, int p2) throws RemoteException;

    public int mul(int p1, int p2)     "        "

}

### (2) MyBeanHomeInterface.java

This is userdefined Home Interface with one abstract create() without parameters if we want stateless session bean or with parameters if we want to fulfill session bean.

```
import javax.ejb.*;
import java.rmi.*;
```

public interface MyBeanHome extends EJBHome

public void create() throws CreateException,  
RemoteException;

### (3) BeanImpl.java

This is the implementation bean class by the user or server side has one default constructor, EJB specific methods, it implements Business logic method of the above userdefined Remote interface.

```
import javax.ejb.*;
public class MyBeanImpl implements SessionBean
```

String name = "abc";

public This is an attribute of session bean  
whose type would be in ejb-jar.xml declared  
in the implementation class that is defined  
method of this implementation class.

alc to app'n req's.

```
public MyBeanImpl()
```

```
{
```

```
S.O.P ("This is bean constructor");
```

```
}
```

```
public void seeSessionContext (SessionContext sc)
```

```
{
```

This method is automatically executed by the EJB container along with an object of SessionContext interface at the time of creating the bean object of this bean class.

```
this.sc = sc; attribute name
```

```
S.O.P ("This is session context method");
```

```
}
```

```
public void ejbCreate()
```

```
{
```

```
S.O.P ("This is ejb create method");
```

```
}
```

```
public void ejbRemove()
```

```
{
```

```
S.O.P ("This is ejb remove method");
```

```
}
```

```
public void ejbActivate()
```

```
{
```

```
S.O.P ("This is ejb activate method");
```

```
}
```

```
public void ejbPassivate()
```

```
{
```

```
s.o.p("This is ejbPassivate method");
```

```
}
```

All the above 5 methods are the implemented abstract life cycle methods available in SessionBean interface, if we do not implement these 5 methods then this Bean class will become as abstract class, all these 5 life cycle methods are automatically executed by the EJB container.

```
public String getMessage()
```

```
{
```

```
s.o.p("This is get message business logic method");
```

```
return ("Hello client, welcome to EJB world");
```

```
public
```

```
{
```

```
public int add(int P1, int P2)
```

```
{
```

```
s.o.p("This is add business logic method");
```

```
return (P1 + P2);
```

```
}
```

```
public int mul(int P1, int P2)
```

```
{
```

```
s.o.p("This is mul business logic method");
```

```
return (P1 * P2);
```

```
}
```

These are 3 business logic methods of the above user-defined RemoteInterface implemented in this bean class acc to appn req's; these methods are executed only when the client call these methods from the client appn otherwise these methods are not automatically executed by the EJB container.

}

#### (4) TestClient.java

This is the client side java appn to access the SessionBean component available at the server & to invoke the business logic methods available in the respective bean component.

```
import javax.naming.*;
 import java.util.*;
 import java.io.*;
```

```
public class TestClient {
```

{

```
 p.s.v.m (String args[]) throws Exception
```

{

```
 properties p = new properties();
```

This will create an obj of properties class available in java.util package.

```
p.load (new FileInputStream ("jndiproperties.properties"));
```

Where load is the method of properties class ie. going to load the weblogic jndiproperties i.e. jndiname, server address of the weblogic appn server which are available in the file called

Load jndiProperties properties into properties class object  
With the help of FileInputStream class.

Now the properties class obj is binding  
Weblogic jndi registry name & the Weblogic server address.

InitialContext ic = new InitialContext();

This will create an obj. of InitialContext  
class which is available in javax.naming package  
along with the respective properties class object.

Now ic is referencing to the stub obj jndi using  
registry, but this stub obj of jndinaming registry is not  
yet connected to the Weblogic appn server jndi registry.

MyBeanHomeIFace mthi = (MyBeanHomeIFace) ic.lookup(  
"s1smain");

Where s1smain is the jndi name of the EJB  
appn specified at the time of deployment of the app  
at serverside.

Where lookup is the method of InitialContext  
class, this method will automatically establish the  
connection to the Weblogic appn server jndi registry  
& get the copy of the stub obj of the HomeInterface  
(the EJB appn available against to the jndi name i.e.  
binded name as 's1smain' into the client app).

Now mthi is referencing to the stub obj  
of the HomeInterface of the EJB appn where there  
one create() is available with networking, marshalling  
& unmarshalling implemented code.

~~4.6.09~~  
MyBeanRemoteIface m bri = mbhi.create();

Where `create()` of userdefined home interface,  
this method is called onto the stub object of home  
Interface available at client side.

When this method call is received at serverside  
then the EJB container automatically create the stub,  
stub obj's of the RemoteInterface & then the copy of  
this stub obj of RemoteInterface is sent back to the  
client, but the bean object i.e. Bean instance of the  
Bean class not yet created so far at serverside.

Now `m bri` is referencing to the stub obj  
of the RemoteInterface where there are business logic  
methods are implemented with networking, marshalling,  
unmarshalling code.

S.O.P

String receivedmsg = m bri.getMessage();

Where `getMessage` is the method of userdefined  
remote interface, this method is called onto the stub  
obj of RemoteInterface available at clientside.

When this method call is received by the  
EJB container at serverside then it will either  
create the bean object of the Bean class or, select  
the existing bean obj available in the Bean pool,  
pass the request to the Bean object so that the  
respective user implemented business logic method

is get executed on the Bean object & then it  
will collect the response & send it to the requested  
client.

```
S.O.P ("received msg");
```

```
DataInputStream stdin = new DataInputStream(System.in);
```

```
S.O.P ("Enter first no:");
```

```
int fno = Integer.parseInt(stdin.readLine());
```

```
S.O.P ("Enter second no:");
```

```
int sno = Integer.parseInt(stdin.readLine());
```

```
int addresult = mбри.add (fno,sno);
```

```
S.O.P ("The result of addition is" + addresult);
```

```
int mulresult = mбри.mul (fno,sno);
```

```
S.O.P ("The result of multiplication is" + mulresult);
```

```
S.O.P ("Bye Bye");
```

```
}
```

```
}
```

## (5) JndiProperties.properties

This is the weblogic properties file

created at clientside along with weblogic jndiregistry name  
& the weblogic server address details.

java.naming.factory.initial = weblogic.jndi.WLInitialContextFactory

java.naming.provider.url = t3://localhost:7001

property name

property value

5-6-09

The following diagram demo the sequence b/w the client & the Weblogic app'n server to invoke the business logic methods available in the stateless session Bean component at server side EJB app'n.

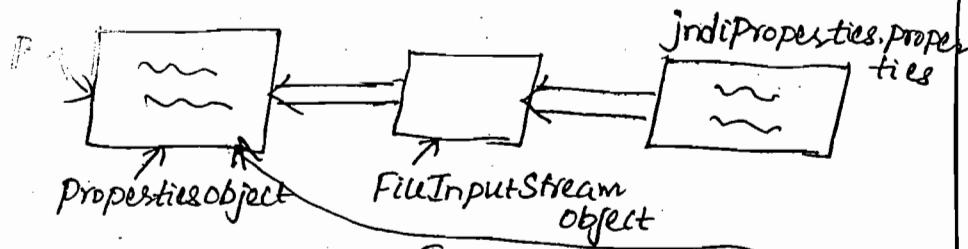
### Client side

#### Test Client.java

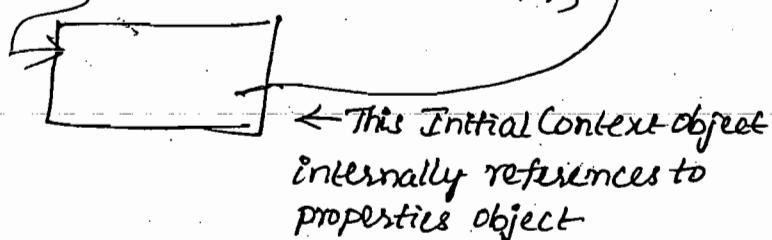
p.o

```
[Properties p = new Properties(new FileInputStream("jndi
Properties.properties"));
```

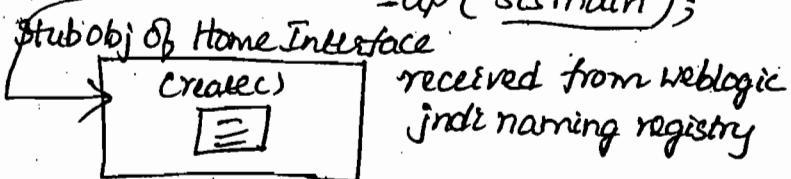
```
Properties P = new Properties();
P.load(new FileInputStream("jndiProperties.properties"));
```



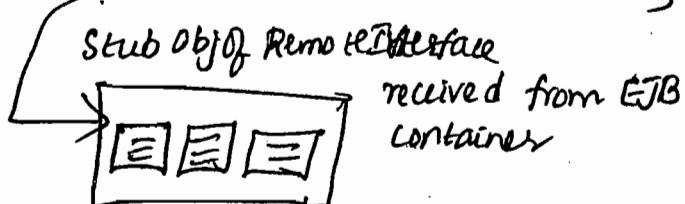
```
Initial Context ic = new InitialContext(p);
```



```
MyBeanHomeInterface mbhi = (MyBeanHomeInterface) ic.lookup
- up("SLSmain");
```



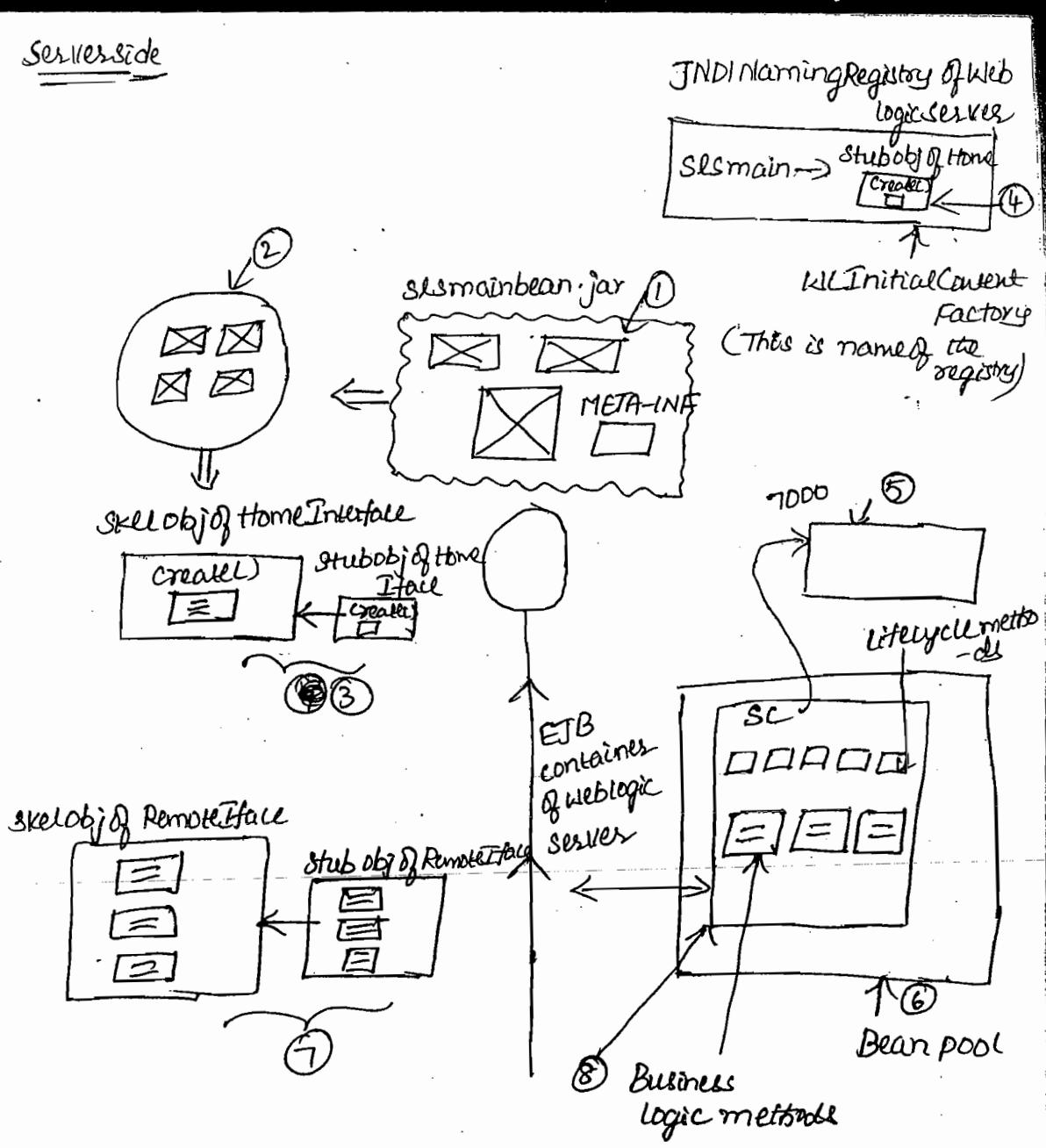
```
MyBeanRemoteInterface mbri = (MyBeanRemoteInterface)
mbhi.create();
```



```
String receivedmsg = mbri.getMessage();
```

← This will call the resepective business logic method

Web logic  
App'n  
Tool server  
running at localhost,  
7001 port no.



- ① This is the deployed jar file of the EJB app'n at the EJB container of weblogic server along with the class files of HomeInterface, RemoteInterface, Bean class, META-INF folder
- ② These are the skel, stub classes for Home, Remote Interfaces are automatically generated by the EJB container at the time of deploying the jar file of the EJB app'n at the server
- ③ These are skel, stub obj's of the HomeInterface automatically created by the EJB container at the time of deploying the jar file of the EJB app'n where there is one create() is available with networking, marshalling; unmarshalling implemented code.
- ④ This is the stub obj of the HomeInterface binded into JNDI Naming Registry of Weblogic server along with the name as slsmain which is specified JNDI name at the time of deploying the jar file of the EJB app'n.
- ⑤ This is SessionContextInterface obj automatically created by the EJB container at some address when the server is started, if we want we can use this object in the EJB app'n alc to app'n req's
- ⑥ This is Empty Bean Pool created by the EJB container when the server is started where there are no bean obj's of the Bean class are available so far
- ⑦ These are skel, stub obj's of the RemoteInterface are automatically created by the EJB container where there are some business logic methods are available with the implemented networking code when the create method call is received from the Client,

copy of this Stub Obj is sent back to the Client JVM.

- ⑧ This is the bean obj ie. bean instance i.e. bean component of the bean class automatically created by the EJB container where there are 5 lifecycle methods & 3 business logic methods which are implemented by the user at server side are available. when it is received the call to the business logic method from the client, the sc attribute of this bean obj is referencing to the session context obj at the server, the request of the client is passed to the bean object, the response is collected from the bean obj & then the response is given back to the requested client.

### → Practical procedure

The following practical procedure demonstrate how to compile the above EJB appn called sesmainbean, how to deploy the jar file of this appn at weblogic appn server after creating the userdomain at the server & how to execute the client appn i.e. TestClient.java

### Path settings (a) Environmental classpath settings

set path = D:\bea\jdk141-03\bin;;  
i.e.) set path↑

set classpath = D:\bea\jdk141-03\lib\tools.jar; D:\bea\webligic81\server\lib\Weblogic.jar;;

JAVA\_HOME = D:\bea\jdk141-03

## Weblogic Builder properties Path

```
D:\bea\jdk141-03\bin\java.exe -client -cp "D:\
bea\weblogic81\server\lib\weblogic.jar;
D:\bea\weblogic81\common\eval\pointbase\lib\pbclient
44.jar;D:\bea\jdk141-03\lib\tools.jar" weblogic.
marathon.Main
```

In to the Target Text Field of the Weblogic  
Builder properties you can copy the above path.

## Creation of User Domain at Weblogic Server

Start



Programs



BEA Weblogic 8.1



Configuration Wizard (Click on this Wizard)



u will get one window , click on next



again click on next



again click on next, A new window is opened



U type the required username as : Anjali

Userpassword as : weblogic

Confirm userpassword : weblogic

After that click on next



Again click on next

at right side change Domain (Default: MyDomain)

With username u can change

it as Anjali



You click on create



Click on done



Finished

Starting the userDomain Weblogic app'n server

Start



programs



BEA Weblogic 8.1



User projects



Anjali (ur username)



Click on start server (starting the Weblogic server)

Compiling & Deploying the EJB APP'n

(1) compile the java files available in the app'n folder  
to generate the class files

(2) start Weblogic Builder, to generate XML file, to  
create jar file, to deploy the jar file of EJB app'n.  
such as

## Generating XML file with META-INF folder:

→ file



open



open the required app'n folder



click on yes

Now XML files are successfully generated with  
META-INF folder.

## Saving XML files:

Click on the modulename i.e. EJBname (the defa  
-ult name is  
MyBeanImpl2)



Give the required EJBname & JNDI name at right  
side such as SLSmain



click on floppy symbol (i.e. save symbol)

Now XML files are saved with META-INF folder  
in the app'n folder.

## Validation of XML files:

Goto Tools



validate descriptors

8-

Now the validation is successful

## Creation of jar file

→ file



archive  
↓  
open the required app'n folder  
↓  
click on save

Now the jar file such as `SLSessionbean.jar` is successfully created & saved in the app'n folder.

Connecting to the server & deploying jarfile of the EJB app'n :

Tools  
↓  
Deploy module  
↓  
Give the Username & Password & click on connect  
↓  
click on deploy module

Now the jarfile of the EJB app'n is successfully deployed.

(3) Execute the Client java app'n such as  
> `java TestClient`

Now enjoy the output.

8-6-09

The following is another practical ex: to demonstrate for stateless SessionBean

`SLSessionbean`  
↓

(1) `MyBeanRemoteFace.java`

- 
- (2) MyBeanHomeIface.java
  - (3) MyBeanImpl.java
  - (4) TestClient.java
  - (5) JndiProperties.properties

### MyBeanRemoteIface.java

```
import javax.ejb.*;
// java.rmi.*;
public interface MyBeanRemoteIface extends EJBObject
{
 public String incrementedBeanValueMessage() throws
 RemoteException;
}
```

### MyBeanHomeIface.java

```
import javax.ejb.*;
// java.rmi.*;
public interface MyBeanHomeIface extends EJBHome
{
 public MyBeanRemoteIface create() throws CreateException,
 RemoteException;
}
```

### MyBeanImpl.java

```
import javax.ejb.*;
public class MyBeanImpl implements SessionBean
{
```

```
int value;
SessionContext sc;
public MyBeanImpel()
{
 S.O.P ("This is Bean constructor");
}
public void setSessionContext(SessionContext sc)
{
 this.sc = sc;
 S.O.P ("This is setSessionContext method");
}
public void ejbCreate()
{
 S.O.P ("This is ejbcreate method, value" + value);
}
public void ejbRemove()
{
 S.O.P ("This is ejbremove method");
}
public void ejbActivate()
{
 S.O.P ("This is ejbactivate method");
}
```

hient

s

Excep

```
public void ejbPassivate()
{
 S.O.P ("This is ejbPassivate method");
}

public String incrementedBeanValueMessage()
{
 value++;
 return "The current bean value is "+value;
}
```

### TestClient.java

```
import javax.naming.*;
// java.util.*;
// java.io.*;

public class TestClient
{
 p.s.v.m (String args[]) throws Exception
 {
 Properties p = new Properties();
 p.load(new FileInputStream("jndi.properties.properties"));
 InitialContext ic = new InitialContext(p);
 MyBeanHomeInterface mbhi = (MyBeanHomeInterface)
 ic.lookup("sleloop");
```

```
MyBeanRemoteIface mbri = mbhi.create();
for(int i=1; i<=10; i++)
{
 String receivedmsg = mbri.incrementedBeanValueMessage();
 System.out.println(receivedmsg);
 Thread.sleep(2000);
}
```

}

### JndiProperties Properties

> javac \*.java  
> StartServer  
> Start (3) Click on weblogicBuilder ( Deployment can be done)  
> java TestClient

## → Java Naming & Directory Interface i.e. JNDI

- \* Naming service is the service that will allow the server app's to store the service obj's with some unique names, these stored obj's are called as named obj's.  
9:
- \* Directory service is added service & top of the naming service to assign some additional attributes such as directory names to the named obj's
- \* JNDI is the specifications i.e. standards provided by the Sun Microsystems with the common API for all the vendors of the app'n servers so that they can develop their own JNDI registers by following these specifications.
- \* The JNDI register of any vendor app'n server can be accessed from the client app'n in Vendor independent approach.
- \* So, the JNDI register is the common point or common interface b/w the client app'n & <sup>the</sup> server app'n
- \* Once the bean app'n is designed & deployed at the app'n server such as Weblogic server with some jndi registered name then the app'n server automatically provide the stub, stub classes for the RemoteInterface & for the HomeInterface.
- \* At the time of deploying the ejb app'n, the server automatically create stub, stub obj's for the HomeInterface the stub obj of the HomeInterface is automatically binded into the jndi registry along with the specified jndi name.

- \* Now any client from the client app'n can look into the JNDI registry of the respective app'n server & get the copy of the stub obj of the HomeInterface of the EJB app'n into the client app'n.

9-6-09

### Life cycle sequence of session Beans at the server

#### (1) Instantiation & Initialization phase

- \* The EJB container will call the default constructor of the bean class at the time of creating bean obj.
- \* After creating the bean obj, the container will automatically call & execute the following 2 lifecycle methods of the bean obj at the Initialization phase i.e.

(1) public void setSessionContext(SessionContext sc);  
(2) public void ejbCreate();

- \* After execution of these two methods, the bean obj is placed into the bean pool so that it is ready to provide the services to the clients.

#### (2) Servicing phase

- \* When the Client call the business logic methods through the stub obj of the RemoteInterface available at client side then the container will select one of the bean obj from the bean pool, pass the request to the bean obj, collects the response from the bean obj, place the bean obj again into the bean pool & then send the response to the requested client.

### (3) Destruction Phase

- \* When the container want to destroy the bean obj then it will pickup the bean obj from the bean pool & automatically execute the following method of the bean obj ie.

public void ejbRemove();

- \* After execution of this method, the state of the bean obj is moved from pool state to does not exist state so that the garbage collector will destroy the bean obj.

- \* The following 2 lifecycle methods are not executed if the bean component is stateless bean component, which means these 2 methods are executed only if the bean component is stateful bean component ie.

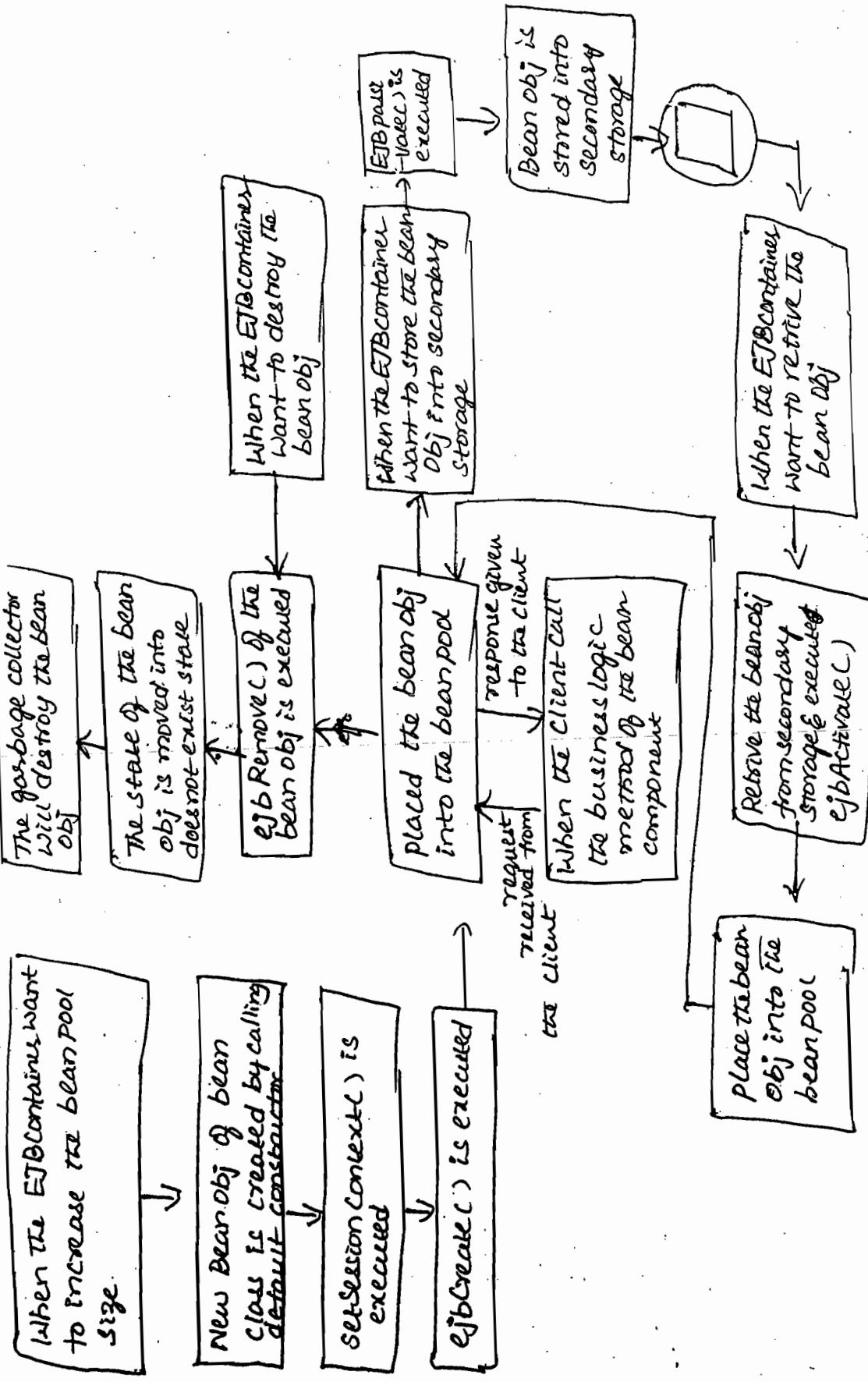
(1) public void ejbActivate();

(2) public void ejbPassivate();

- \* If the bean component is stateful session bean component then the EJB container automatically store the bean obj from the bean pool into the secondary storage & retrieve the bean component from the secondary storage into a bean pool depending on the bean pool capacity & the overload at the server.

- \* In such case, the ejbPassivate() is executed before storing the bean obj into the secondary storage, the ejbActivate() is executed after retrieving the bean obj from the secondary storage.

→ The following diagram demonstrate this lifecycle sequence of the Session Beans at the server



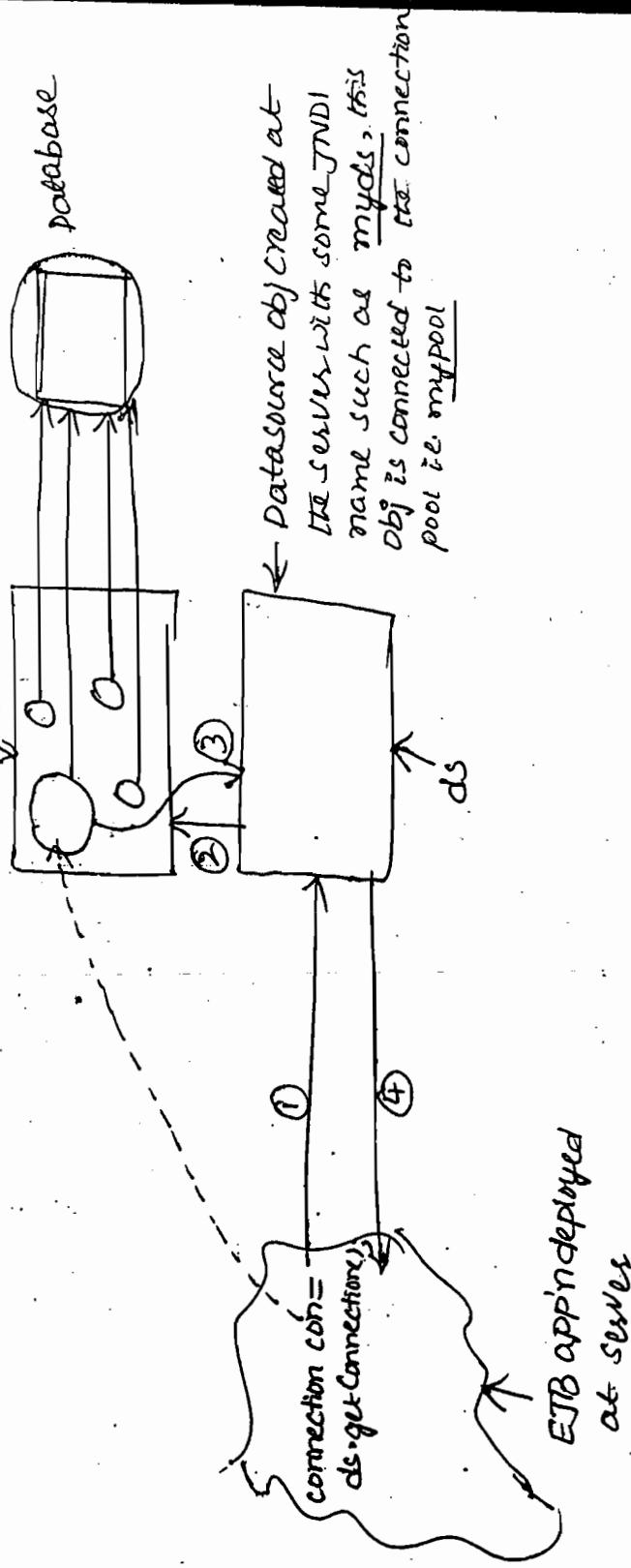
10/5/09

## Connection pooling mechanism at the server

- \* If we develop the Enterprise java app'n with the EJB components to communicate with the database by establishing the connection every time to the DB to process the various clients requests then the following problems will be encountered i.e.
  - (i) This is very time consuming process bcoz the time taken to establish the connection, to close the connection to the DB is also included in the request processing time.
  - (ii) This will effect the system performance bcoz the overhead at the server is increased.
- \* So, we should use the mechanism called Connection pooling at the server to overcome these problems.
- \* In this connection pooling mechanism, the connection pool manager at the appn server will establish a pool of connections to the DB & maintain the pooled connections in the connection pool available at the appn server.
- \* If we want to use these connections in the EJB app'n then we should communicate with the connection pool manager through the obj of datasource interface which is available in javax.sql package.
- \* The foll diag demon how to get the database connection into the EJB app'n from the connection pool which is already created at the appn server through the Datasource object.

## Serverside

connection pool created at server  
with some name such as mypool



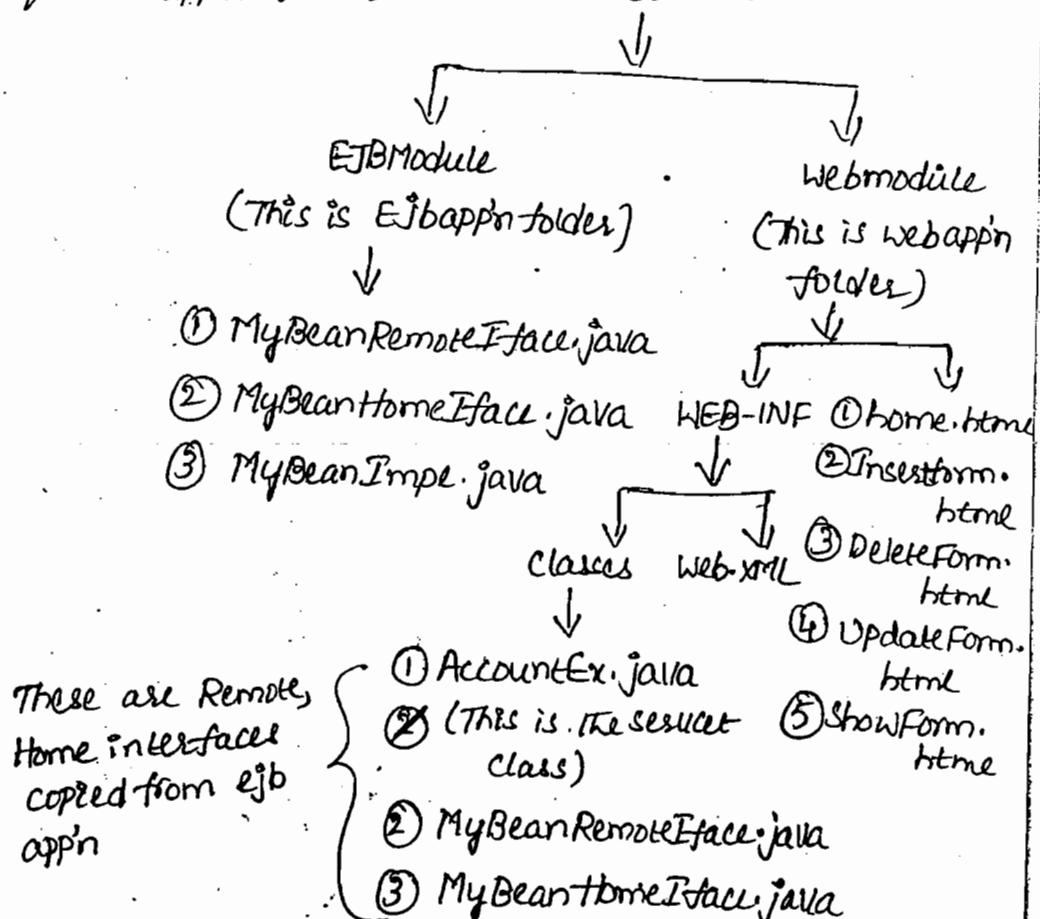
- Note \* Here `getconnection` is the method of `Data source interface obj`, this method will select one of the connection obj available in the connection pool & return the reference of that obj to the EJB appn.
- \* Now `con` which is the reference variable of connection interface in the EJB appn is referring to the selected connection obj in the connection pool. so that we can directly use this connection obj in the EJB appn to perform

the required funs in the DB

→ The following practical ex: demonstrate how to use the bean components of the EJB app'n in the service of the Web app'n. to provide the accessability of the bean components to the web clients by using Connection pooling mechanism to perform the required funs in the DB.

### Hierarchy of the app'n

Name of the app'n folder is → s1s1dbwebbean



## Ejb module

## MyBeanRemoteIface.java

This is user-defined Remote interface in the ejb app'n.

```
import javax.ejb.*;
import java.rmi.*;
public interface MyBeanRemoteInterface extends EJBObject
{
 public boolean insert(int accno, String accname,
 float accbal) throws RemoteException;
 public boolean delete(int accno) throws RemoteException;
 public boolean update(int accno, float amt) throws
 RemoteException;
 public float getBalance(int accno) throws RemoteException;
```

## MyBeanHomeIface.java

This is user-defined Home interface in the esp appn

```
import javax.ejb.*;
import java.rmi.*;
public interface MyBeanHomeInterface extends EJBHome
{
 public MyBeanRemoteInterface create() throws
 CreateException, RemoteException;
```

## MyBeanImpl.java

This is the implementation class for the bean class in the ejb app'n.

```
import javax.ejb.*;
import java.sql.*;
import javax.sql.*;
import javax.naming.*;

public class MyBeanImpl implements SessionBean
{
 SessionContext sc;
 DataSource ds;

 public MyBeanImpl()
 {
 System.out.println("This is the bean constructor");
 }

 public void setSessionContext(SessionContext sc)
 {
 this.sc = sc;
 System.out.println("This is setSessionContext method");
 }

 public void ejbCreate()
 {
 System.out.println("This is ejbCreate method");
 }

 public void try
 {
 }
```

```
InitialContext ic = new InitialContext();
```

properties obj is not required for this InitialContext obj becoz this InitialContext obj is created at server side, which means the properties obj is required when we create the initial context obj at client side.

```
ds = (DataSource) ic.lookup("myds");
```

attribute<sup>class</sup> where myds is the jndi name of the datasource obj which is already created at the weblogic app in server by connecting to the connection pool at the server.

```
}
```

```
catch (Exception e)
```

```
{
```

```
err
```

```
}
```

```
}
```

```
public void ejbRemove()
```

```
{
```

```
s.o.p ("This is ejbRemove method");
```

```
}
```

```
public void ejbActivate()
```

```
{
```

```
s.o.p ("This is ejbActivate method");
```

```
}
```

```
public void ejbPassivate()
```

```
{
```

```
s.o.p ("This is ejbPassivate method");
```

```
public boolean insert(int accno, String accname,
 float accbal)
{
 Connection con = ds.getConnection();
 PreparedStatement pstmt = con.prepareStatement("insert
 into account values(???)");
 pstmt.setInt(1, accno);
 pstmt.setString(2, accname);
 pstmt.setFloat(3, accbal);
 int count = pstmt.executeUpdate();
 pstmt.close();
 con.close();
 if(count > 0)
 {
 return true;
 }
 else
 {
 return false;
 }
}
catch(Exception e)
{
}
```

```
 return false;
 }
 public boolean delete(int accno)
 {
 try
 {
 Connection con = ds.getConnection();
 PreparedStatement pstmt = con.prepareStatement(""
 delete from alc where accno=?");
 pstmt.setInt(1, accno);
 int count = pstmt.executeUpdate();
 pstmt.close();
 con.close();
 if (count > 0)
 {
 return true;
 }
 else
 {
 return false;
 }
 }
 catch (Exception e)
 {
 return false;
 }
 }
```

```
public boolean update(int accno, float amt)
{
 try
 {
 Connection con = ds.getConnection();
 PreparedStatement pstmt = con.prepareStatement("
 update a/c set accbal=accbal+?
 where accno=?");
 pstmt.setFloat(1, amt);
 pstmt.setInt(2, accno);
 int count = pstmt.executeUpdate();
 pstmt.close();
 con.close();
 if(count>0)
 {
 return true;
 }
 else
 {
 return false;
 }
 }
 catch(Exception e)
 {
 return false;
 }
}
```

```
public float getBalance(int accno)
{
 float bal = 0;
 try
 {
 Connection con = ds.getConnection();
 PreparedStatement pstmt = con.prepareStatement("select
accbal from alc where accno=?");
 pstmt.setInt(1, accno);
 ResultSet rs = pstmt.executeQuery();
 if(rs.next())
 {
 bal = rs.getFloat(1);
 }
 else
 {
 System.out.println("acc no does not exist");
 }
 rs.close();
 pstmt.close();
 con.close();
 }
 catch(Exception e)
 {
 return false;
 }
}
```

~~11-6-v~~ Web module  
Home.html

```
<html>
<body bgcolor = wheat>
<center>
ONLINE BANKING

 INSERT ACCOUNT

 DELETE ACCOUNT

 SHOW BAL

 UPDATE ACCOUNT
</center>
</body>
</html>
```

Insertform.html

```
===== =====
<html>
<body bgcolor = lightgreen>
<center>
 INSERT ACCOUNT
<form method = get action = /Webmodule/myservlet>
 Enter acc no: ↴ ↴
 name of the appn url pattern
 <input type = text name = accno>

 Enter name:
```

<input type="text" name="accname">  
<br> <br>  
Enter balance:  
<input type="text" name="accbal">  
<br> <br>  
<input type="submit" name="send" value="INSERT">  
</form>  
</center>  
</body>  
</html>

### Deleteform.html

<html>  
<body bgcolor="lightgreen">  
<center>  
Account closer (8) closure  
<form method="get" action="/Webmodule/myservlet">  
Enter acc no:  
<input type="text" name="accno">  
<br> <br>  
<input type="submit" name="send" value="DELETE">  
</form>  
</center>  
</body>  
</html>

### updateform.html

<html>  
<body bgcolor="pink">

## Account update

```
<form method="get" action="/Webmodule/myservlet">
Enter a/c no:
<input type="text" name="accno">

Enter amount:
<input type="text" name="amt">

<input type="submit" name="send" value="UPDATE">
</form>
</center>
</body>
</html>
```

## ShowForm.html

```
<html>
<body bgcolor="cyan">
<center>
```

## Balance Enquiry

```
<form method="get" action="/Webmodule/myservlet">
Enter a/c no:
<input type="text" name="accno">

<input type="submit" name="send" value="SHOW">
</form>
</center>
</body>
</html>
```

## AccountEx.java

The url pattern mapping of this HttpServlet class must be /myservlet, this will receive either insert form request or delete form request or update form request or show form request along with the respect -ive parameters names & values as get request, to support this get request we should override the doGet() in this HttpServlet class.

```
import javax.servlet.*;
" " " http.*;
```

```
" java.io.*;
```

```
" javax.naming.*;
```

```
public class AccountEx extends HttpServlet
```

```
{
```

```
MyBeanRemoteInterface mbri;
```

```
public void init()
```

```
{
```

```
try
```

```
{
```

```
InitialContext ic = new InitialContext();
```

The properties obj is not required bcoz this  
InitialContext obj is created at serverside

```
MyBeanHomeInterface mbhi = (MyBeanHomeInterface)
ic.lookup("selsdb");
```

Where selsdb is the jndi name of the deployed  
ejb appn.

```
mbri = mbhi.create();
}

catch(Exception e)
{
}

}

public void doGet(HttpServletRequest req, HttpServletResponse
 res) throws ServletException, IOException;
{
 res.setContentType("text/html");
 PrintWriter out = res.getWriter();
 String choice = req.getParameter("send");

 The value of this choice is either insert or delete
 or update or show depending on the form request received
 by this servlet class.

 if(choice.equals("INSERT"))
 {
 int accno = Integer.parseInt(req.getParameter(
 "accno"));
 String accname = req.getParameter("accname");
 float accbal = Float.parseFloat(req.getParameter(
 "accbal"));

 if(mbri.insert(accno, accname, accbal) == true)
 }
}
```

This will call insert business logic method available in the Session Bean component of the EJB app'n.

```
out.println ("<body bgcolor = wheat >");
 " (" Account inserted successfully ");
 " (" <center> HOME
 </center> ");
 " (" </body > ");
```

}

else

{

```
 out.println (" Account insertion failed ");
```

}

}

```
if (choice.equals ("DELETE"));
```

{

```
 int accno = Integer.parseInt (req.getParameter ("accno"));
```

```
 if (mbsi.getDelete (accno) == true)
```

{

```
 out.println ("<body bgcolor = yellow >");
```

```
 " (" acc deleted successfully ");
```

```
 " (" <center> HOME
 </center> ");
```

```
 " (" </body > ");
```

}

else

{

```
 out.println (" Account deletion failed ");
```

```
if(choice.equals("SHOW"))
{
 int accno = Integer.parseInt(req.getParameter("accno"));
 float bal = mbri.getBalance(accno);
 out.println("<body bgcolor=pink>");
 " "<div balance Rs:"+bal);
 " ("<center>HOME
 </center>");
 " ("</body>");
}

if(choice.equals("UPDATE"))
{
 int accno = Integer.parseInt(req.getParameter("accno"));
 float amt = float.parseFloat(req.getParameter("amt"));
 if(mbri.update(accno,amt) == true)
 {
 out.println("<body bgcolor=wheat>");
 " ("<div updated successfully");
 " ("<center>HOME
 </center>");
 " ("</body>");

 }
 else
 {

```

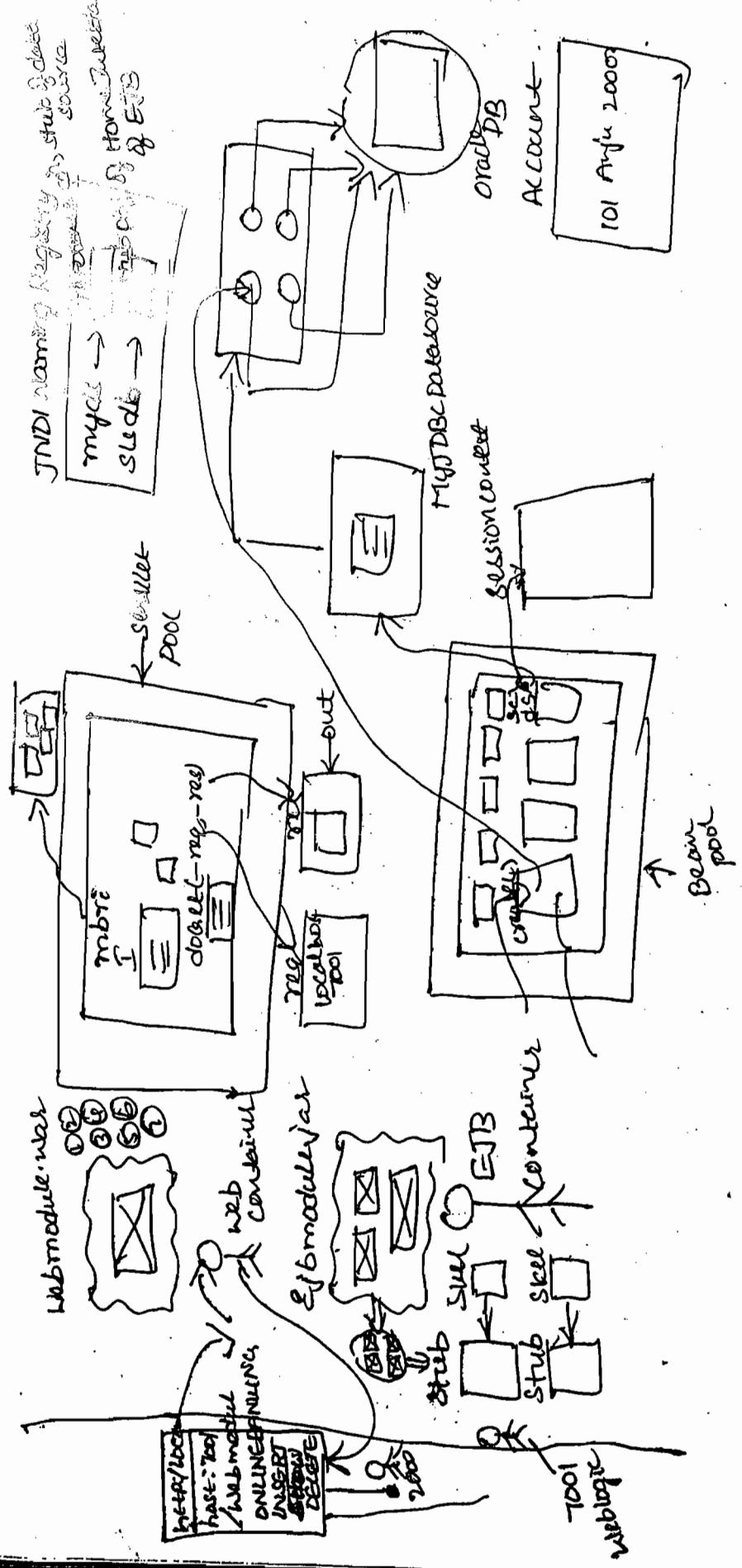
```
out.println (" a/c updation failed");
```

```
}
no));
}
}
}
E
```

### Web.XML

```
<web-app>
<service>
<service-name> first </service-name>
<service-class> AccountEx </service-class>
</service>
<service-mapping>
<service-name> first </service-name>
<url-pattern> /myservlet </url-pattern>
</service-mapping>
<welcome-file-list>
<welcome-file> Home.html </welcome-file>
</welcome-file-list>
</web-app>
```

ME



6-09

## practical procedure

### Creation of connection pooling in Weblogic 8.1

- (1) start weblogic server
- (2) open console of the server ie. open the browser & give the url as

`http://localhost:7001/console`

\* Give the username & password

\* click on signin

\* Now the console of the 8.1 weblogic server is opened

- (3) Creation of connection pooling ie.

\* click on + of services

\* click on + of jdbc

\* " " connectionpools

\* " " new jdbc connection pool

\* give the database type as oracle, the database drivers as thin drivers & then click on continue.

\* Give the database name as server, username as scott, password as tiger, confirm the password as tiger, click on continue.

\* click on test-drivers configuration.

\* Now the connection is successful.

\* Click on continue & deploy

Note Make sure that deployed is true. otherwise

we should repeat the above procedure once again.

## Creation Of Datasource

- Click on + of services
- " " " " jdbc
- " " " datasources
- <sup>click on</sup> Configure a new jdbc datasource
- Give the required jndi name such as myds.
- Click on continue & again continue
- " " Create

Note: Make sure that deployed is true.

\* After creation of the connection pool & the datasource, at the server, we should deploy the web app'n & the ejb app'n with weblogic builder such as

- (1) Create war file & jar files
- (2) Deploy war & jar files with the known procedure
- (3) Create account database table at the database with the column names as accno, accname, accbal.
- (4) Request the <sup>web</sup> app'n through the browser with the URL as

`http://localhost:7001/Webmodule`

Now enjoy the response

## Main Bean to Local Bean Communication

- \* We should follow the following steps to communicate from the mainbean component to the local bean component of the same ejb app'n i.e.

- (1) We should define the mainbean Remote interface
- (2) We should define mainbean Home interface
- (3) " " " " " implementation class
- (4) " " " " localbean remote interface by extending Ejb LocalObject interface
- (5) We should define localbean Home Interface by extending EJBLocalHome interface
- (6) We should define localbean implementation class
- (7) " " " create " object in the ejbCreate() of the mainbean class
- (8) We should call the localbean object business logic methods from the main bean obj business logic methods acc to appn req's.

### practical APPN

Slsbean to bean



- (1) MyBeanLocalRemoteInterface.java
- (2) MyBeanLocalHomeInterface.java
- (3) MyBeanLocalImpl.java
- (4) MyBeanRemoteInterface.java
- (5) MyBeanHomeInterface.java
- (6) MyBeanImpl.java      (7) TestClient.java
- (8) JndiProperties.properties.

## MyBeanLocalRemoteInterface.java

This is the userdefined RemoteInterface for the localbean component with a set of abstract business logic methods.

```
import javax.ejb.*;
public interface MyBeanLocalRemoteInterface extends
 EJBLocalObject
{
 public String getReversedName(String name);
}
```

## MyBeanLocalHomeInterface.java

This is the userdefine HomeInterface for the localbean component with one create method.

```
import javax.ejb.*;
public interface MyBeanLocalHomeInterface extends
 EJBLocalHome
{
 public MyBeanLocalRemoteInterface create() throws
 CreateException;
}
```

## MyBeanLocalImpl.java

This is the implementation class for the localbean component

```
import javax.ejb.*;
public class MyBeanLocalImpl implements SessionBean
{
```

```
 SessionContext sc;
```

```
 public void setSessionContext(SessionContext sc)
{
```

```
 this.sc = sc;
```

```
 S.O.P("This is setSessionContext method of
local bean component");
```

```
}
```

```
 public void ejbCreate()
```

```
{
```

```
 S.O.P("This is ejbCreate method of local bean
component");
```

```
}
```

```
 public void ejbRemove()
```

```
{
```

```
 S.O.P("This is ejbRemove method of local bean
component");
```

```
}
```

```
 public void ejbActivate()
```

```
{
```

```
 S.O.P("This is ejbActivate method of
local bean component");
```

```
}
```

```
public void ejbPassivate()
{
 S.O.P("This is ejbPassivate method of local
 bean component");
}

public String getReversedName(String name)
{
 S.O.P("This is getReversedName method of
 local bean component");
 String reversedname = " ";
 for(int i=name.length()-1; i>=0; i--)
 {
 reversedname = reversedname + name.charAt(i);
 }
 return reversedname;
}
```

M  
=

### MyBeanRemoteInterface.java

undefined

This is the <sup>↑</sup> Remote Interface for the  
mainbean component with a set of Business  
logic methods.

```
import javax.ejb.*;
" java.rmi.*;
public interface MyBeanRemoteInterface extends EJBObject
{
```

```
public String getMessage (String name) throws
 RemoteException;
```

### MyBeanHomeIface.java

This is userdefined Home interface for the main bean component with one create method.

```
import javax.ejb.*;
import java.rmi.*;
public interface MyBeanHomeIface extends EJBHome
{
 public MyBeanRemoteIface create() throws CreateException,
 RemoteException;
```

### MyBeanImpl.java

This is the implementation class for the main bean component.

```
import javax.ejb.*;
import javax.naming.*;
public class MyBeanImpl implements SessionBean
{
 SessionContext sc;
 MyBeanLocalHomeIface mbLhi;
 MyBeanLocalRemoteIface mbLri;
 public void setSessionContext (SessionContext sc)
 {
```

```
this.sc = sc;
```

```
s.o.p("This is setSessionContext method of
mainbean component");
```

```
}
```

```
public void ejbCreate()
```

```
{
```

```
s.o.p("This is ejb Create() of main bean
component");
```

```
}
```

```
try
```

```
{
```

```
InitialContext ic = new InitialContext();
```

```
mblhi = (MyBeanLocalHomeInterface)ic.lookup
("first local");
```

Where firstlocal is the jndi name specified  
for the localbean component at the time of deploying  
this ejb app'n at the server.

```
mberi = mblhi.create();
```

```
}
```

```
catch(Exception e)
```

```
{
```

```
}
```

```
}
```

```
public ejbRemove()
```

```
{
```

→

```
S.O.P ("This is ejbRemove method of mainbean component");
```

```
}
```

```
public void ejbActivate();
```

```
{
```

```
S.O.P (" This is ejbActivate() of mainbean component");
```

```
}
```

```
public void ejbPassivate();
```

```
{
```

```
S.O.P (" This is ejbPassivate() of mainbean component");
```

```
}
```

```
public String getMessage(String name);
```

```
{
```

```
S.O.P (" This is getMessage() of mainbean component");
```

```
String reversedName = mblri.getReversedName(name);
```

This will call the business logic method of the local bean component.

```
String message = "your name is:" + name + "In your
Reversed name is" + reversedName;
```

```
return (message);
```

```
}
```

```
}
```

→ Indeproperties properties Same as previous one

## TestClient.java

This is the client side java app'n to invoke the business logic methods of the mainbean component of the EJB app'n deployed at the server.

```
import javax.naming.*;
```

```
 " java.util.*;
```

```
 " " .io.*;
```

```
public class TestClient
```

```
{
```

```
 p.s.v.M(String args[]) throws Exception
```

```
{
```

```
 Properties p = new Properties();
```

```
 p.load(new FileInputStream("jndiproperties.properties"));
```

```
 InitialContext ic = new InitialContext(p);
```

```
 MyBeanHomeIface mbhi = (MyBeanHomeIface) ic.lookup(
```

```
 (" seebtob" >"));
```

Where seebtob is the jndi name specified for the mainbean component at the time of deploying the EJB app'n at the server.

```
 MyBeanRemoteIface mbri = mbhi.create();
```

```
 DataInputStream stdin = new DataInputStream(
```

```
 (System.in));
```

```
 S.O.P(" Enter ur name");
```

```
 String name = stdin.readLine();
```

```
 String receivedmsg = mbri.getMessage(name);
```

This will call the business logic method of the main bean component available at the server.

S.O.P (received msg);

S.O.P ("Bye Bye");

}

3.

5-6-09  
The following practical example shows how to define the stateful session bean components in the EJB appn

Note \* The create() declared in the Home Interface will specify the bean component as either stateless session bean component or stateful sessionbean component to the EJB container.

\* If the create() is declared in the Home interface without parameters then the bean component will be treated as sessionbean component, similarly if this create() is declared with parameters then the bean component will be treated as stateful session bean component.

\* The EJB create() implemented in the bean class must exactly match with the parameter list of the create() otherwise the EJB compiler which is the tool available at the WeblogicBuilder will raise an error at the time of deploying the EJB appn.

\* This EJB create() which is implemented in the bean class refers to the create() in the Home

interface will specify the bean component as stateless session bean component or stateful session bean component so this EJB create life cycle method is called as specification method

- \* If we do not implement this EJBCreate() in the bean class then the java compiler will not raise any error at the time of compiling the java files, but the EJB compiler will raise an error at the time of deploying the EJB appn.
- \* We can't deploy the EJB appn without implementing this EJB create specification method in the bean class.

statefulbean  
↓  
(stateful) ↓

- (1) MyBeanRemoteInterface.java
- (2) MyBeanHomeInterface.java
- (3) MyBeanImpl.java
- (4) TestClient.java
- (5) JndiProperties.properties

MyBeanRemoteInterface.java

```
import javax.ejb.*;
import java.rmi.*;
public interface MyBeanRemoteInterface extends EJBObject
{
 public String incrementedBeanValue(Message message) throws
 RemoteException;
}
```

MyBeanHomeInterface.java

```
import javax.ejb.*;
" java.rmi.*;
public interface MyBeanHomeInterface extends EJBHome
{
 public MyBeanRemoteInterface create(int p) throws
 CreateException, RemoteException;
}
```

MyBeanImpl.java

```
import javax.ejb.*;
public class MyBeanImpl implements SessionBean
{
 int value;
 SessionContext sc;
 public MyBeanImpl()
 {
 S.O.P("This is bean constructor");
 }
 public void setSessionContext(SessionContext sc)
 {
 this.sc = sc;
 S.O.P (" This is setSessionContext method");
 }
 public void ejbCreate(int p)
 {
```

```
S.O.P ("This is ejbCreate method");
```

```
S.O.P ("This bean component is created for the client"
+P);
}
```

```
public void ejbRemove()
```

```
{
```

```
S.O.P ("This is ejbRemove method");
```

```
}
```

```
public void ejbActivate()
```

```
{
```

```
S.O.P ("This is ejbActivate method");
```

```
}
```

```
public void ejbPassivate()
```

```
{
```

```
S.O.P ("This is ejbPassivate method");
```

```
}
```

```
public String incrementedBeanValueMessage()
```

```
{
```

```
Value++;
```

```
return "The current bean value is" + Value;
```

```
}
```

```
}
```

## TestClient.java

```
import javax.naming.*;
import java.util.*;
```

```

import java.io.*;
public class TestClient
{
 p.s.v.m(String args[]) throws Exception
 {
 properties p = new properties();
 p.load(new FileInputStream("jndiproperties.prop-
 -erties"));
 InitialContext ic = new InitialContext(p);
 MyBeanHomeIface mbhi = (MyBeanHomeIface)
 ic.lookup("sfsloop");
 MyBeanRemoteIface mbri = mbhi.create(Integer-
 parseInt(args[0]));
 for(int i=1; i<=10; i++)
 {
 String receivedmsg = mbri.incrementedBeanValue
 Message();
 System.out.println(receivedmsg);
 Thread.sleep(2000);
 }
 }
}

```

→ Jndiproperties.properties Same as above

→ The following practical app'n demonstrate how to define stateful Session Bean component in the EJB app'n for the product building of every individual requested client.

SfsBillingbean ← Name of the app'n folder

- (1) MyBeanRemoteInterface.java
- (2) MyBeanHomeInterface.java
- (3) MyBeanImpl.java
- (4) Product.java
- (5) TestClient.java
- (6) JndiProperties.properties

MyBean Remote Interface

```
import javax.ejb.*;
// java.rmi.*;
// java.util.*;

public interface MyBeanRemoteInterface extends EJBObject
{
 public void addProduct(Product p) throws RemoteException;
 public Map getAllProducts() throws RemoteException;
}
```

Where Product is the user-defined Serializable class

in the same ejb app'n.

Where Map is the predefined class available in java.util package.

Define  
or

### MyBeanHomeIface.java

```
import javax.ejb.*;
import java.rmi.*;
public interface MyBeanHomeIface extends EJBHome
{
 public MyBeanRemoteIface create(String uid) throws
 CreateException, RemoteException;
}
```

### MyBeanImpl.java

```
import javax.ejb.*;
import java.util.*;
public class MyBeanImpl implements SessionBean
```

```
{
 SessionContext sc;
 Map m;
```

```
 public void setSessionContext(SessionContext sc)
```

```
{
 this.sc = sc;
```

```
 S.O.P ("This is setSessionContext method");
```

```
?
 public void ejbCreate(String uid)
```

```
{
 S.O.P ("This is ejbCreate method");
```

```
S.O.P ("This bean component is created for the
client" + uid);
```

lect

Excep

???

class

```
 p.o (m = new HashMap()); } p.o
```

This will create an obj of HashMap class available in java.util package

```
}
```

```
public void ejbRemove()
```

```
{
```

```
 S.O.P ("This is ejbRemove()");
```

```
}
```

```
public void ejbActivate()
```

```
{
```

```
 S.O.P ("This is ejbActivate method");
```

```
}
```

```
public void ejbPassivate()
```

```
{
```

```
 S.O.P ("This is ejbPassivate method");
```

```
}
```

```
public void addProduct(Product p)
```

```
{
```

```
 productid
```

```
 String pid = p.getPid();
```

```
 boolean exist = m.containsKey(pid);
```

Where getPid() is the userdefined method

in the product class that will return the Id name of the product as a string.

Where containsKey() is the method of HashMap class that will check whether the respective product id is available in the HashMap obj, if so

it will return true otherwise return false.

if(exist)

{

Product existpro = (Product)m.get(pid);

Where get() is the method of HashMap class  
that will return the reference of the product obj  
available in the HashMap with respective the specified  
product id.

float totalprice = existpro.getPrice() + p.getPrice();

Where getPrice() is the userdefined method  
in the product class that will return the price of the  
product as float value.

existpro.setPrice(totalprice);

Where setPrice() is the userdefined method  
in the product class that will store the respective  
price value into the respective product obj.

}

else

{

m.put(pid,p);

Where put() is the method of HashMap class  
that will store the respective product obj along  
with the respective product id into the HashMap obj

}

}

```
public Map getAllProducts()
```

```
{
```

```
 return m;
```

```
}
```

```
}
```

~~16-6-09~~

### product.java

This is userdefined class to store the product id & the product price by implementing Serializable interface so that the copy of this obj can be passed b/w the client JVM & server JVM

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

```
public class Product implements Serializable
```

```
{
```

```
 String pid; → productprice
```

```
 float pprice;
```

```
 public Product(String pd, float pr)
```

```
{
```

```
 pid = pd;
```

```
 pprice = pr;
```

```
}
```

```
 public void setPid(String pd)
```

```
{
```

```
 pid = pd;
```

```
}
```

```
public void setPrice(float pr)
{
```

```
 pprice = pr;
```

```
}
```

```
public String getPidc()
```

```
{
```

```
 return pid;
```

```
}
```

```
public float getPriceC()
```

```
{
```

```
 return pprice;
```

```
}
```

```
}
```

### TestClient.java

```
import java.util.*;
```

```
 "i. lo.*;
```

```
 "javax.naming.*;
```

```
public class TestClient
```

```
{
```

```
 p.s.v.m(String args[]) throws Exception
```

```
{
```

```
 properties p = new Properties();
```

```
 p.load(new FileInputStream("jndiproperties.properties"));
```

```
 InitialContext ic = new InitialContext(p);
```

```
MyBeanHomeIface mbhi = (MyBeanHomeIface) ic.lookup
("stsbilling");
```

```
MyBeanRemoteIface m bri = mbhi.create(args[0]);
```

```
DataInputStream stdin = new DataInputStream(System
• in);
```

```
while(true)
{
```

This is infinite while loop bcoz of the condition  
of the while is always true.

```
S.O.P ("Enter product id:");
```

```
String pid = stdin.readLine();
```

```
S.O.P ("Enter product price");
```

```
float pprice = Float.parseFloat(stdin.readLine());
```

```
Product pro = new Product(pid, pprice);
```

```
m bri.addProduct(pro);
```

[If we are giving Qty  
(pid, pprice, i)]

```
S.O.P ("Do u want to add another product (y/n):");
```

```
String choice = stdin.readLine();
```

```
If (!choice.equalsIgnoreCase ("yes"))
```

```
break;
```

```
}
```

```
S.O.P (args[0] + " Billing Details are ... \n\n");
```

```
Map m = m bri.getAllProducts();
```

```
Set s = m.keySet();
```

Where keySet is the method of map interface  
that will return an obj of set interface available

el

se

the  
pas  
ele

}

in `java.util` package along with a set of Hashkey elements ie. a set of product id names available in the respective HashMap object.

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

Where `Iterator` is the method of set interface that will return the reference of the initial cursor position available in the set interface obj before the first element.

while (*i*.hasNextC())

۸

2  
Where `hasNext` is the method of iterator interface that will check whether the next element available in the respective set interface obj, if so it will return true otherwise return false.

String pid = (String)i.next();

`String pid = (String) iterator.next();`  
Where `next()` is the method of iterator  
interface that will return the next available element  
in the respective set interface obj.

Product pro = (product) m.get(productId);

```
s.o.p("product id".+pro.getId());
```

```
S.O.P("product price:" + pro.getPrice());
```

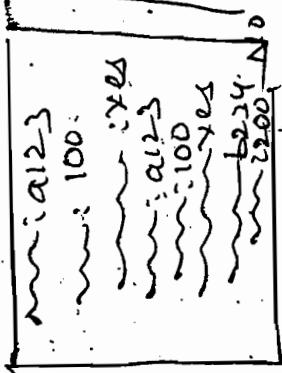
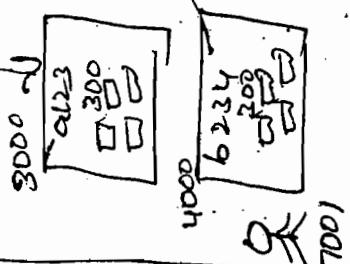
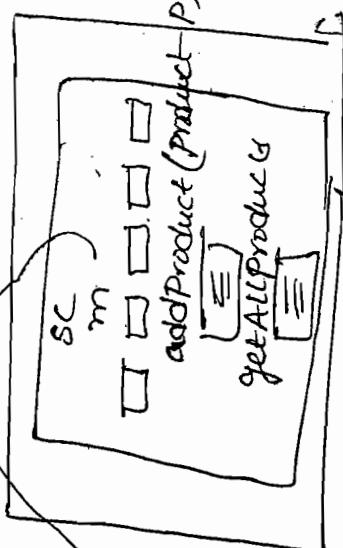
S.O.P ("In");

3

g.o.p ("Bye Bye");

2

| Hash Key | obj Reference |
|----------|---------------|
| a123     | 3000          |
| b234     | 4000          |



If we are including Qty

product.java

Implementation  
existpro.increment();

public class  
{

String pid;

float pprice;

int pQty;

public product (String pd, float pr, int pq)

{

pid = pd;

pprice = pr;

pQty = pq;

}

public void increment() → public int getQty()

{

pQty++;

}

{ return pQty;  
}

TestClient.java

Product pro = new product(pid, pprice, 1);

while(i.hasNext())

{

s.o.p ("product Qty" + pro.getQty());

→ The following practical app'n demonstrate how to define stateful Session Bean component in the EJB app'n for the product billing & quantity of every individual registered client.

MyBeanRemoteIface.java → same as above

MyBeanHomeIface.java → " " "

MyBeanImpl.java

Same as above except if block, In the if block we can write the following code:

if (exist)

```
 product existpro = (product) m.get(Pid);
 float totalprice = existpro.getPrice() + p.getPrice();
 existpro.setPrice(totalprice);
 existpro.increment();
```

}

else

{

```
 m.put(Pid, p);
```

}

remaining as same

product.java

```
public class product implements Serializable
```

```
{ String Pid; }
```

```
float Pprice;
```

```
int Pqty;
```

fine

value

```
public product (String pd, float pr, int pqty)
```

{

```
 pid = pd;
 pprice = pr;
 pqty = pq;
```

}

remaining methods are same include increment  
( ), getQty()

```
public void increment()
```

{

```
 pqty++;
```

}

```
public int getQty()
```

{

```
 return pqty;
```

}

### TestClient.java

Same as above in while condition

you can include the following code.

```
while (true)
```

{

    same as above

```
product pro = new product (pid, pprice, 1);
```

remaining same as above In another  
while condition you include this code

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

~~18-6-09~~

## EJB 3.0

→ We should follow the following steps to design the EJB app'n with EJB3.0 container at the app'n server ie.

(1) We should define the Remote interface just like a plain java interface with a set of abstract business logic methods. (4)

We should declare this interface by using a Remote meta data annotation to provide the info to the EJB container that the interface is Remote interface.

(2) We should define the bean implementation class by implementing the above Remote Interface just like a plain java class. (5)

We should declare this implementation class by using either ~~@~~ stateless or ~~@~~ statefull metadata annotation to provide the info to the EJB container that the bean component is stateless (~~&~~) statefull.

We should implement all the business logic methods acc to app'n req's in this implementation class otherwise the bean class will become an abstract class.

We need not implement any lifecycle methods in the bean class.

Note NO HomeInterface is required in EJB3.0 (6)

(3) We should store the above 2 classes ie. Remote interface class file & the Bean Implementation class file into one package by using the package stmt in the respective java files.

We should use the following command to compile the above two java file & to store the class files into the respective package i.e.

```
>javac -d . *.java
```

(4) We should create jarfile with the above 2 class files available in the package by using the following command i.e.

```
>jar -cvf somefilename.jar packagename
```

Note NO XML files are required in this jarfile bcoz we are using metadata annotations to provide the descriptive info to the EJB Container.

(5) We should deploy the jarfile of the EJB app'n at some app'n server such as Sun app'n server

Whenever we deploy the jarfile of the EJB app'n at the server then the EJB container automatically generates Stub,skel classes for the Remote Interface, create Stub,skel obj's of those classes, bind the stub obj of the Remote interface into jndi registry with the name such as packagename.RemoteInterface name

Now the server is waiting to receive the requests from the clients to the deployed EJB app'n

(6) We should create clientside java app'n to look into the jndi registry of the Sun app'n server with the jndi properties file so that we will directly get the copy of the stub obj of the Remote interface into the

We can call the required business logic methods of the Bean component available at the server through this stub obj of Remote interface.

→ The following practical app'n demonstrate how to create the EJB 3.0 app'n at the app'n server.

Slemainbean

1

- 1. MyBeanInterface.java
- 2. MyBeanImpl.java
- 3. TestClient.java
- 4. JndiProperties.properties

} serverside

} clientside

## MyBeanInterface.java

```
package session;
import javax.ejb.*;
```

② Remote public interface MyBeanInterface

{

```
public String getMessage());
public int add(int p1, int p2);
public int mul(int p1, int p2);
```

۳

## MyBeanImpl.java

package Session;

```
import javax.ejb.*;
```

## 11. 11 • Annotation.\*;

① stateless public class MyBeanImpl implements  
MyBeanIface

{

② Resource SessionContext sc;

Where ② Resource annotation available in javax.annotation package is used to provide the info to the EJB container to store the reference of SessionContext obj available at the EJB container into sc attribute of this bean component, this concept is called as Dependency Injection which means we are injecting the SessionContext obj available at the EJB container into the Bean Obj.

public MyBeanImpl()

{

This is the default constructor of the Bean class which is optional in the bean class

s.o.p ("This is bean constructor");

}

public String getMessage()

{

s.o.p ("This is getMessage Business logic method");

return "Hello Client, Welcome to EJB3.0 World";

}

public int add(int p1, int p2)

{

s.o.p ("This is add business logic method");

```
 return (P1+P2);
 }

 public int mul(int P1, int P2)
 {
 S.O.P ("This is mul business logic method");
 return (P1 * P2);
 }
}
```

### TestClient.java

```
import session.MyBeanIface;
import javax.naming.*;
import java.util.*;
import java.io.*;

public class TestClient
{
 public void main (String args[]) throws Exception
 {
 Properties p = new Properties();
 p.load (new FileInputStream ("jndiproperties.properties"));

 InitialContext ic = new InitialContext (p);
 MyBeanIface mbi = (MyBeanIface) ic.lookup ("Session.
 MyBeanIface");

 String receivedmsg = mbi.getMessage ();
 S.O.P (receivedmsg);
 }
}
```

```

DataInputStream stdin = new DataInputStream(System.in);
System.out.print("Enter first no:");
int fno = Integer.parseInt(stdin.readLine());
System.out.print("Enter second no:");
int sno = Integer.parseInt(stdin.readLine());
int addresult = mbi.add(fno, sno);
System.out.println("The result of addition is:" + addresult);
int mulresult = mbi.mul(fno, sno);
System.out.println("The result of multiplication is:" + mulresult);
System.out.println("Bye Bye");
}
}

```

### Jndi Properties.properties

This is the jndi.properties file created along with the properties value of the jndi registry of the sun app'n server

java.naming.factory.initial = com.sun.enterprise.naming.

    SerialInitContextFactory

java.naming.provider.url = iiop://localhost:4848.

=

ion.

## 19-6 OA classpath settings

(4)

```
set path = D:\sun\jdk16\bin;;
set classpath = D:\sun\jdk16\lib\tools.jar; D:\sun\
AppServer\lib\jaraee.jar; D:\sun\
AppServer\lib\appserver-rt.jar;;
```

JAVA-HOME = D:\sun\jdk16

## practical procedure

- 1) compile the java files to generate the classfiles & to store the classfiles into the package with the following command.

>javac -d . \*.java

- 2) prepare the jar files <sup>along</sup> with the classes available in the package by using the following command.

>jar -cvf somefilename.jar packagename

- 3) start the sun application server

Go to Start Windows

↓  
programs  
↓  
sun microsystems  
↓  
application server

(5)

Click on Start Default Server.

Now the server is started running at localhost,

4848 portno:

(4) open the server admin console

start of windows



programs



Sun Microsystems



application server



Click on adminconsole



Give the username as admin & the  
password as 12345678

Now the server console is opened

(5) Deploy the jar file of the EJB app'n.

Click on ejb modules on server console



Click on deploy



Browse the required jar file of the EJB app'n.



Click on OK.

Now the app'n is deployed successfully.

(6) Execute the client app'n such as

> java TestClient

Now enjoy the o/p.

→ SLSloopbean  
↓

- (1) MyBeanInterface.java
- (2) MyBeanImpl.java
- (3) TestClient.java
- (4) JndiProperties.properties

### MyBeanInterface.java

```
package session;
import javax.ejb.*;
@Remote public interface MyBeanInterface
{
 public String incrementedBeanValueMessage();
}
```

### MyBeanImpl.java

```
package session;
import javax.ejb.*;
 " " . annotation.*;
@Stateless(mappedName = "SLSloop")
public class MyBeanImpl implements MyBeanInterface
{
 int value;
 @Resource SessionContext sc;
 public MyBeanImpl()
 {
 System.out.println("This is Bean constructor");
 }
}
```

```
public String incrementedBeanValueMessage ()
{
 value++;
 return "The current Bean Value is "+value;
}
```

### TestClient.java

Note Where mappedName attribute is used along with  
① stateless annotation to bind the stub obj of the  
RemoteInterface of this bean class with the required  
jndi name such as SLSB001  
If we do not use this mappedName attribute then  
the server automatically bind with the default name  
i.e Session.MyBeanIFace.

### TestClient.java

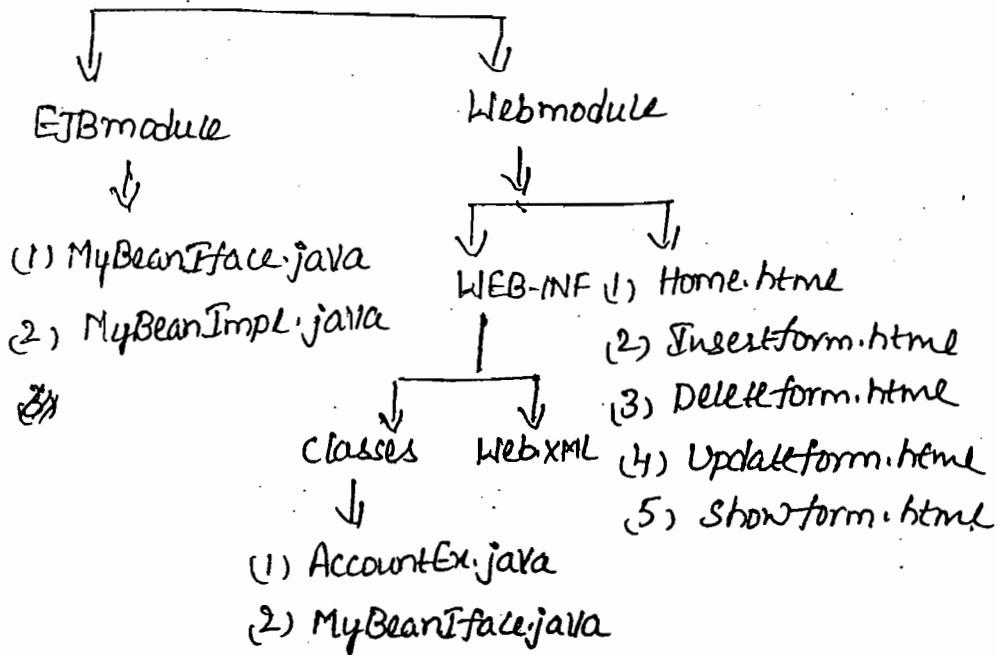
```
import session.MyBeanIFace
" javax.naming.*;
" java.util.*;
" " .io.*;
public class TestClient
{
 p.s.v.m(String args[]) throws Exception
 {
 Properties p = new Properties();
 p.load(new FileInputStream("jndi.properties.
 properties"));
```

```

InitialContext ic = new InitialContext();
MyBeanInterface mbi = (MyBeanInterface) ic.lookup("slsloop");
String recd
for (int i=1; i<=10; i++)
{
 String receivedmsg = mbi.incrementedBeanValueMess
 -age();
 System.out.println(receivedmsg);
 Thread.sleep(2000);
}

```

→ sesdbWebbean



### MyBeanInterface.java

```
package session;
import javax.ejb.*;
 " ".annotation.*;

@Remote public interface MyBeanInterface
{
 public boolean insert(int accno, String accname,
 float accbal);

 public boolean delete(int accno);
 public boolean update(int accno, float amt);
 public float getBalance(int accno);
}
```

### MyBeanImpl.java

```
package session;
import javax.ejb.*;
 " ".java.sql.*;
 " ".javax.sql.*;
 " ".annotation.*;

@Stateless public class MyBeanImpl implements
 MyBeanInterface
{
```

@Resource SessionContext sc;

@Resource(name = "myds") DataSource ds;

public MyBeanImpl()

{

s.o.p("This is Bean constructor");

}

```
public boolean insert(int accno, String accname,
 float accbal)
```

```
{
```

```
try
```

```
{
```

```
Connection con = ds.getConnection();
```

```
PreparedStatement pstmt = con.prepareStatement
 ("insert into account values
 (?, ?, ?);")
```

```
pstmt.setInt (1, accno);
```

```
pstmt.setString (2, accname);
```

```
pstmt.setFloat (3, accbal);
```

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

```
pstmt.close();
```

```
con.close();
```

```
if (count > 0)
```

```
{
```

```
 return true;
```

```
}
```

```
else
```

```
{
```

```
 return false;
```

```
}
```

```
}
```

```
catch (Exception e)
```

```
{
```

```
}
```

```
return false;
```

```
public boolean delete(int accno);
{
 try
 {
 Connection con = ds.getConnection();
 PreparedStatement pstmt = con.prepareStatement
 ("delete from ac/c where
 accno = ?");
 pstmt.setInt (1, accno);
 int count = pstmt.executeUpdate();
 pstmt.close();
 con.close();
 if(count > 0)
 {
 return true;
 }
 else
 {
 return false;
 }
 }
 catch (Exception e)
 {
 }
 return false;
}
```

```
public boolean update(int accno, float amt)
{
 try
 {
 Connection con = ds.getConnection();
 PreparedStatement pstmt = con.prepareStatement
 ("update alc set ACCBAL=ACCBAL+
 where ACCNO=?");
 pstmt.setFloat(1, amt);
 pstmt.setInt(2, accno);
 int count = pstmt.executeUpdate();
 pstmt.close();
 con.close();
 if (count > 0)
 {
 return true;
 }
 else
 {
 return false;
 }
 }
 catch (Exception e)
 {
 }
 return false;
}
```

```
public float getBalance (int accno)
{
 float bal = 0;
 try
 {
 Connection con = ds.getConnection();
 PreparedStatement pstmt = con.prepareStatement
 ("select acbal from account
 where accno = ?");
 pstmt.setInt (1, accno);
 ResultSet rs = pstmt.executeQuery();
 if (rs.next())
 {
 bal = rs.getFloat(1);
 }
 else
 {
 System.out.println ("acc no does not exist");
 }
 rs.close();
 pstmt.close();
 con.close();
 }
 catch (Exception e)
 {
 return bal;
 }
}
```

Where myds is the Jndiname specified for the datasource obj which is already created at Sun application server by connecting to the connection pool which is connected with the Oracle database.

### Home.html

```
<html>
<body bgcolor = "wheat">
<center>
ON LINE BANKING

 INSERT ACCOUNT

 DELETE ACCOUNT

 SHOW BAL

 UPDATE ACCOUNT
</center>
</body> </html>
```

### Insertform.html

```
<html>
<body bgcolor = "lightgreen">
<center>
INSERT ACCOUNT
<form method = "get" action = "/webmodule/myservlet">
Enter acc no:
```

cation  
is

```
<input type="text" name="accno">

Enter name:
<input type="text" name="accname">

Enter bal:
<input type="text" name="accbal">

<input type="submit" name="send" value="INSERT">
</form>
</center>
</body> </html>
```

Deleteform.html

---

```
<html>
<body bgcolor="lightgreen">
<center>
 ACCOUNT CLOSURE
<form method="get" action="/webmodule/myservlets">
 Enter acc no:
 <input type="text" name="accno">

 <input type="submit" name="send" value="DELETE">
</form>
</center>
</body>
</html>
```

## updateform.html

```
<html>
<body bgcolor=pink>
<center>
 ACCOUNT UPDATE
<form method=get action=/webmodule/myservlet>
 Enter acc no:
 <input type=text name=accno>

 Enter amount:
 <input type=text name=amt>

 <input type=submit name=send value=UPDATE>
</form>
</center>
</body>
</html>
```

## Showform.html

```
<html>
<body bgcolor=cyan>
<center>
 BALANCE ENQUIRY
<form method=get action=/webmodule/myservlet>
 Enter acc no:
 <input type=text name=accno>


```

```
<input type="submit" name="send" value="SHOW">
</form>
</center>
</body> </html>
```

21-6-09

```
import session.MyBeanInterface;
" javax.servlet.*;
" " . http.*;
" java.io.*;
" javax.ejb.*;
public class AccountEx extends HttpServlet
{
 @EJB MyBeanInterface mbi;
```

Where @EJB metadata annotation is used to inject the stub obj of Remote interface of the bean component into this service obj automatically by the ejb container.

Now mbi attribute of this service obj is automatically referencing its the stub obj of the Remote interface of the bean component available at the ejb container so that through this attribute we can call all the business logic methods available in the respective bean component.

```
public void doGet (HttpServletRequest req,
 HttpServletResponse res) throws ServletException
{
 res.setContentType ("text/html");
 PrintWriter out = res.getWriter();
 String choice = req.getParameter ("send");
 if (choice.equals ("Insert"))
 {
 int acno = Integer.parseInt (req.getParameter ("acno"));
 String acname = req.getParameter ("acname");
 float acbal = Float.parseFloat (req.getParameter ("acbal"));
 if (Cmbt.insert (acno, acname, acbal) == true)
 {
 out.println (" <body bgcolor = >");
 " (" "Successfully");
 " (" "<center>
 Home
 </center> ");
 " (" "</body>");
 }
 }
 else
 {
 out.println ("account insertion failed");
 }
}
```

if (choice.equals("delete"))

SE, IOE

{

int accno = Integer.parseInt(req.getParameter("accno"));

{

if (mbi.delete(accno) == true)

{

out.println("<body bgcolor=yellow>");

" alc deleted successfully");

" <center> <a href=home.html>  
Home </a>");

" </center> </body>");

}

else

{

out.println(" alc deleted successfully");

}

}

if (choice.equals("show"))

{

int accno = Integer.parseInt(req.getParameter("accno"));

float bal = mbi.getBalance(accno);

out.println("<body bgcolor=pink>");

" alc. bal is:" + bal);

" <center> <a href=home.html>  
Home </a></center>");

" </body>");

}

```

if (choice.equals ("update"))
{
 int acno = Integer.parseInt (req.getParameter
 ("acno"));
 float amt = Float.parseFloat (req.getParameter
 ("amt"));
 if (mbi.update (acno, amt) == true)
 {
 out.println ("");
 " (" "alc successfully created");
 " (" "<center>
Home </center>");
 " (" "</body >");
 }
 else
 {
 out.println (" alc updated failed");
 }
}

```

web.xml

```

<Web-app>
<service>
 <service-name> first </service-name>
 <u-class> AccountEx </service-class>
</service>
<service-mapping>

```

```

<{servlet-name}> first <{servlet-name}>
<url-pattern> /myservlet </url-pattern>
</servlet-mapping>
<welcome-file-list>
<welcome-file> home.html </welcome-file>
</welcome-file-list>
</web-app>

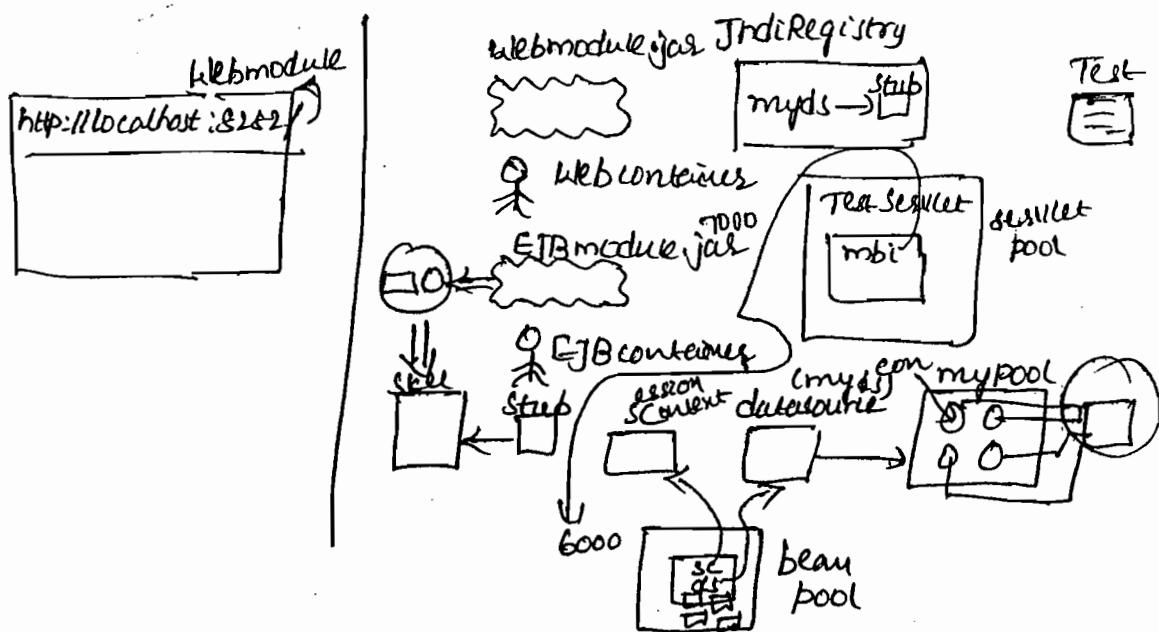
```

ejbmodule  
-> jarac -d \*.java

-> jar -cvf ejbmodule.jar session;

webmodule > jar -cvf Webmodule.war

Creating datasource Resources → <sup>jdbc</sup><sub>new</sub> → jndi name (myds),  
pool name (mypool)



When we deploy the ejbmodule.jar it automatically generate the stub & stub obj of Remote. In this first we deploy the Ejb app'n.

Ejbmodule.jar → open → OK

Web app'n deploy

Web app'n deploy → deploy → browse → webmodule →

Webmodule.war → open → OK.

For verification

click on mypool  
↓  
ping.

22-6-09

## practical procedure

(1) creation of connection pooling in sun app'n server

start the server



open the admin console



Click on Resources



Click on JDBC



click on connection pooling



click on new



give some pool name such as mypool



give resource type as javax.sql.datasource



give the database vendor as oracle



click on next



give the DB name as server



give the password as tiger



give the URL as jdbc:oracle:thin:@localhost:1521:  
server



give the username as scott



click on finish.

Now we should test driver configuration to see whether the connection pool is created successfully or not such as

→ click on mypool

→ click on ping

now displayed as ping succeeded

(2) Creation of the datasource at sun app'n server

After clicking the jdbc in the above procedure

→ click on jdbc resources

→ click on new

→ give the required JNDI name such as myds

→ select the pool name ie. mypool

→ click on OK

Now the datasource obj is created successfully.

(3) Creation of War file & Jar file

→ We should use the following command to create the war file for the web app'n ie.

> jar -cvf warfilename.war(war)

(4) We should deploy jar file of the EJB app'n with the known procedure at the server before deploying the war file of the web app'n bcoz we are using dependency injection.

(5) Deploy the war file of the web app'n at the server such as

→ click on web apps

→ click on deploy

→ browse the required war file  
→ click on OK.

(6) Now open the browser and make a request to the web app'n with the following url

http://localhost:8282/web module.

Now enjoy the response.

→ subbean to bean



- (1) MyBeanLocalIface.java
- (2) MyBeanLocalImpl.java
- (3) MyBeanIface.java
- (4) MyBeanImpl.java
- (5) TestClient.java
- (6) Jndiproperties.properties

### MyBeanLocalIface.java

This is userdefined Remote interface for the Localbean component.

```
package session;
import javax.ejb.*;
@Local public interface MyBeanLocalIface {
 public String getReversedName(String name);
}
```

Note Where ~~@~~ **@Local** metadata annotation is used to provide the info to the EJB container that this is local interface.

## MyBeanLocalImpl.java

This is the implementation class for the local bean component.

```
package Session;
import javax.annotation.*;
 // // EJB.*;
```

② stateless public class MyBeanLocalImpl implements  
MyBeanLocalIface

{

③ Resource SessionContext sc;

```
public String getReversedName (String name)
```

{

s.o.p ("This is getReversedName method of local  
bean component");

```
String reversedname = " ";
```

```
for (int i = name.length() - 1; i >= 0; i--)
```

{

```
reversedname = reversedname + name.charAt(i);
```

}

```
return (reversedname);
```

}

}

## MyBeanInterface.java

This is userdefined RemoteInterface for the mainbean component.

```
package session;
import javax.ejb.*;
④ Remote public interface MyBeanInterface
{
 public String getMessage(String name);
}
```

## MyBeanImpl.java

This is the implementation class for the mainbean component.

```
package session;
import javax.annotation.*;
" " . ejb.*;
④ stateless public class MyBeanImpl implements
 MyBeanInterface
{
 ④ EJB MyBeanLocalInterface mbl;
}
```

This metadata annotation is used to inject the local bean component into the mainbean component.

Now mbl attribute of this mainbean component is referencing to the stub obj of the RemoteInterface of the local bean component.

```

@Resource SessionContext sc;
public String getMessage(String name)
{
 S.O.P("This is getMessage() of mainbean component");
 String reversedname = mbli.getReversedName(name);
 This will call getReversedName business logic method
 available in the local bean component
 String message = "ur name is "+ name + "In ur rever-
 sed name is "+ reversed
 -name;
 return (message);
}

```

### TestClient.java

```

import session.MyBeanIface;
import javax.naming.*;
import java.util.*;
import java.io.*;
public class TestClient
{
 p.s.v.M (String args[]) throws Exception
 {
 Properties p = new Properties();
 p.load (new FileInputStream ("jndiproperties.
 properties"));
 InitialContext ic = new InitialContext(p);
 }
}

```

```

MyBeanIface mbi = (MyBeanIface) ic.lookup("session.

 MyBeanIface");
);
DataInputStream stdin = new DataInputStream(System.in);
S.O.P ("Enter ur name");
String name = stdin.readLine();
String receivedmsg = mbi.getMessage(name);
S.O.P (receivedmsg);
S.O.P ("Bye Bye");
}
}

```

→ The following practical appn demonstrate how to define  
the Stateful SessionBean component in EJB 3.0 appn.

SFSloopbean

↓

- (1) MyBeanIface.java
- (2) MyBeanImpl.java
- (3) TestClient.java
- (4) Jndiproperties.properties

MyBeanIface.java

```

package session;
import javax.ejb.*;
@Remote public interface MyBeanIface
{

```

```
public String IncrementedBeanValueMessage();
```

3

## MyBeanImpl.java

```
package Session;
import javax.ejb.*;
" " annotation.*;
```

(@) stateful public class MyBeanImpl implements MyBeanIface

3

```
int value;
```

## ② Resource Session Content Sc;

```
public MyBeanImpl()
```

{

S.O.P ("This is Bean constructor");

S.O.P ("new Bean created for new client");

3

```
 public String incrementedBeanValueMessage();
```

{

Value++;

return "The current Bean value is "+value;

3

3

TestEJB NOTE Where ~~(a)~~ stateful metadata ~~(a)~~ annotation  
is used to provide the info to the EJB container  
that this bean component is stateful bean component.

## TestClient.java

```
import session.MyBeanInterface;
// javax.naming.*;
// java.util.*;
// java.io.*;

public class TestClient
{
 public static void main(String args[]) throws Exception
 {
 Properties p = new Properties();
 p.load(new FileInputStream("JndiProperties.properties"));
 InitialContext ic = new InitialContext(p);
 MyBeanInterface mbi = (MyBeanInterface) ic.lookup("session"
 + "MyBeanInterface");
 for(int i=1; i<=10; i++)
 {
 String receivedmsg = mbi.incrementedBeanValue();
 System.out.println(receivedmsg);
 Thread.sleep(2000);
 }
 }
}
```

→ The following practical ex: demonstrate how to define  
the stateful Session Bean component in EJB 3.0 app for  
the product billing for every requested client.

Sfsbillingbean

↓

- (1) product.java
- (2) MyBeanInterface.java
- (3) MyBeanImpl.java
- (4) TestClient.java
- (5) JndiProperties.properties

product.java

```
package Session;
import java.io.*;
public class Product implements Serializable
{
 String pid;
 float pprice;
 int pquantity;
 public Product(String pd, float pr, int pq)
 {
 pid = pd;
 pprice = pr;
 pquantity = pq;
 }
 public void setPid(String pd)
 {
 pid = pd;
 }
}
```

```
public void setPrice(float pr)
{
 pprice = pr;
}

public String getPid()
{
 return pid;
}

public float getPrice()
{
 return pprice;
}

public void incrementQuantity()
{
 quantity++;
}

public int getQuantity()
{
 return quantity;
}
```

### MyBeanIface.java

```
package Session;
import javax.ejb.*;
import java.util.*;
```

① Remote public interface MyBeanInterface

```
{
 public void addProduct(Product p);
 public Map getAllProducts();
}
```

MyBeanImpl.java

```
package session;
import javax.ejb.*;
import java.util.*;
import javax.annotation.*;
```

② Stateful public class MyBeanImpl implements  
 MyBeanInterface

```
{
 @Resource SessionContext sc;
 Map m;
 public MyBeanImpl()
 {
 m = new HashMap();
 }
 public void addProduct(Product p)
 {
 String pid = p.getPid();
 boolean exist = m.containsKey(pid);
 if(exist)
 }
```

```
product existpro = (product) m.get(pid);
float totalprice = existpro.getPrice() + p.getPrice();
existpro.setPrice(totalprice);
existpro.incrementQuantity();
}
else
{
 m.put(pid, p);
}
}
}

public Map getAllProducts()
{
 return m;
}
```

### TestClient.java

~~23-b-09~~

```
import session.MyBeanInterface;
// session.product;
// java.util.*;
// .io.*;
// javax.naming.*;

public class TestClient
{
 P.S.V.M (String args[]) throws Exception
{
```

```
Properties p = new Properties();
p.load(new FileInputStream ("jndi.properties.properti
es"));
InitialContext ic = new InitialContext(p);
MyBeanInterface mbi = (MyBeanInterface) ic.lookup(".
Session. MyBeanInterface");
DataInputStream stdin = new DataInputStream(System.
in);
while(true)
{
 System.out ("Enter product id");
 String pid = stdin.readLine();
 System.out ("Enter product price");
 float pprice = Float.parseFloat(stdin.readLine());
 Product pro = new Product (pid, pprice, 1);
 mbi.addProduct (pro);
 System.out ("Do you want to add another product (yes/no)");
 String choice = stdin.readLine();
 if (!choice.equalsIgnoreCase ("yes"))
 break;
}
System.out ("\n\n Billing details are ..");
Map m = mbi.getAllProducts();
Set s = m.keySet();
Iterator i = s.iterator();
while (i.hasNext())
{
```

```
String pid = (String)i.next();
Product pro = (Product)m.get(pid);
S.O.P ("product id" + pro.getId());
S.O.P ("product quantity" + pro.getQuantity());
S.O.P ("product total price" + pro.getPrice());
S.O.P ("\\n\\n");
```

{

S.O.P ("Bye Bye");

{

{

→ The following practical app'n demonstrate how to implement the lifecycle methods inside the bean class by using some metadata annotations acc to app'n reqs.

### SFSloopbeanlifecycle

↓

- (1) MyBeanIface.java
- (2) MyBeanImpl.java
- (3) TestClient.java
- (4) JndiProperties.properties

### MyBeanIface.java

```
package session;
import javax.ejb.*;
@Remote public interface MyBeanIface
```

{

```
public String incrementedBeanValueMessage();
```

```
}
```

### MyBeanImpl.java

```
package session;
import javax.ejb.*;
 " annotation.*;
```

@stateful public class MyBeanImpl implements  
MyBeanIface

```
{
```

```
int value;
```

@Resource SessionContext sc;

```
public MyBeanImpl()
```

```
{
```

```
s.d.p("This is bean constructor");
```

```
s.d.p("New bean created for new client");
```

```
}
```

```
public String incrementedBeanValueMessage()
```

```
{
```

```
value++;
```

```
return "The current bean value is "+value;
```

```
}
```

@PostConstruct public void init()

```
{
```

Where @PostConstruct metadata annotation  
is used to provide the info to the EJB Container  
to execute this following init() after creation of

bean obj of this bean class.

s.o.p ("This is init()");

}

(a) PrePassivate public void passivate()

{

where (a) PrePassivate metadata annotation is used to provide the info to the EJBContainer to execute this following passivate() before storing the bean obj into the secondary storage.

s.o.p ("This is passivate()");

}

(a) PostActivate public void activate()

{

Where (a) PostActivate method is executed after retrieving the bean obj from the secondary storage into the bean pool.

s.o.p ("This is activate()");

}

(a) PreDestroy public void destroy()

{

This method is executed before destroying the bean obj.

s.o.p ("This is destroy()");

}

{

## TestClient.java

24

```
import Session.MyBeanIface;
" javax.naming.*;
" java.util.*;
" java.io.*;
public class TestClient
{
 p.s.v.m (String args[]) throws Exception
{
 Properties p = new Properties();
 p.load (new FileInputStream ("jndiproperties.
 properties"));
 InitialContext ic = new InitialContext (p);
 MyBeanIface mbi = (MyBeanIface) ic.lookup
 ("Session.MyBeanIface");
 for (int i=1; i<=10; i++)
 {
 String receivedmsg = mbi.incrementedBeanValue
 Message();
 S.O.P (receivedmsg);
 Thread.sleep (2000);
 }
}
```

24-6-09

In the above practical ex: we have define lifecycle methods inside the bean class by using the metadata annotations.

- \* If we want we can define a separate plain java class with the lifecycle methods by using the same metadata annotations & then we attach this plain java class <sup>as</sup> an Interceptor class to the bean class, in such case we need not implement the lifecycle methods inside the bean class.
- \* If we attach the interpreter class to the bean class then the ejbcontainer automatically create the obj of interpret class at the time of creating the bean obj of bean class, execute the life cycle methods available in the interpreter obj & then pass the request to the bean obj
- \* We prefer to use this technique of defining the lifecycle methods in the separate interpreter class only if we want to perform the same initialization same closing func are required to be performed for the various bean components of various ejb app'n deployed at the server.  
→ The following practical ex: demonstrate how to define the lifecycle methods in the plain java class & how to attach that plain java class as an interceptor class to the bean class of the ejb app'n.

So loop bean lifecycle.



- (1) LifecycleMethods.java
- (2) MyBeanInterface.java
- (3) MyBeanImpl.java
- (4) TestClient.java , 5) TestProperties.properties

## lifecyclemethods.java

This is plain java class defined by user with the lifecycle methods required for the bean component so that this class can be attach as the interceptor class to the bean class.

```
package session;
import javax.ejb.*;
import annotation.*;
import interceptor.*;
public class LifecycleMethods
{
```

① PostConstruct public void init(InvocationContext ic)

{

This init() is automatically executed by ejb container along with invocation context interface obj available at the ejb container after creating the bean obj of bean class to perform some initialization fns in the ejb app'n.

```
s.o.p ("This is init()");
```

}

② PrePassivate public void passivate(InitialContext ic)

{

```
s.o.p ("This is passivate method");
```

}

③ PostActivate public void activate(InvocationContext ic)

{

```
s.o.p ("This is activate()");
```

}

① PreDestroy public void destroy(InvocationContext ic)

{

S.O.P ("This is destroy()");

}

}

### MyBeanInterface.java

package session;

import javax.ejb.\*;

① Remote public interface MyBeanInterface

{

public String incrementedBeanValueMessage();

}

### MyBeanImpl.java

package session;

import javax.ejb.\*;

" " .annotation.\*;

" " .interceptor.\*;

① Stateful

① Interceptors(LifecycleMethods.class)

public class MyBeanImpl implements MyBeanInterface

{

Where ① Interceptors metadata annotation is used to attach the respective interceptor class ie. LifecycleMethods class as an Interceptor class to this bean class so that the ejb container automatically create the respective interceptor class obj at the time of creating the bean obj of this bean class.

```

int value;
① Resource SessionContext sc;
public MyBeanImpl()
{
 S.O.P ("This is bean constructor");
 " ("new bean created for new client");
}
public String incrementedBeanValueMessage()
{
 Value++;
 return "The current bean value is" + Value;
}

```

### TestClient.java

25  
=

```

import session.MyBeanIFace;
" javax.naming.*;
" java.util.*;
" " .io.*;
public class TestClient
{
 P.S.V.M (String args[]) throws Exception
 {
 Properties p = new Properties();
 p.load (new FileInputStream ("jndiproperties.
 properties"));
 InitialContext ic = new InitialContext (p);
 MyBeanIFace mbi = (MyBeanIFace) ic.lookup
 ("Session.MyBeanIFace");
 }
}
```

```

for (int i=0; i<=10; i++)
{
 String receivedmsg = mbi.IncrementedBeanValue
 Message();
 S.O.P (receivedmsg);
 Thread.sleep (2000);
}
}

```

25/6/09

### Entity Beans

- \* The session beans which are developed at the server for the data persistence ie storing & retrieving the data from the database are called as Entity beans
- \* The persistence is the ability of storing the data contained in the java obj's into the database such as Oracle database.
- \* Persistence in EJB 3.0 is managed by Java Persistence API (JPA), this API automatically persist the Java Obj's into the database by using the technique called Object Relational Mapping (ORM)
- \* ORM is the technique of mapping the data available in the java obj's into the DB tables by using metadata annotations.

\* The persistence java obj's which are called as entities are automatically managed through the Entity Manager Interface Obj available at the persistence context of the EJB container.

\* The foll

### Entity Mgr func

→ The following are the entity mgr func used to perform the database operations such as insert, delete, update, select persistence operations on the DB tables.

(1) em.persist(emp);

Where emp is the Employee entity class obj, this method will insert the record into the database table along with the data available in the respective entity obj.

(2) em.remove(emp);

This method will delete the respective record from the DB-table, if such record does not exist in the table then it will raise an exception.

(3) Employee emp = (Employee) em.find(Employee.class, eno);  
method

This will select the record from the DB-table against to the specified employee no & return an obj of the employee entity class along with the data of the record, if such employee number does not exist in the table then it will return null null value.

→ If we update or change the values in the entity class obj then those values are automatically updated into the corresponding DB table as shown below.

Employee emp = (Employee) emf.create(Employee.class, eno)  
emp.esal = emp.esal + 500;

These updated values in the Entity class obj are automatically reflected i.e. updated into the corresponding record in the corresponding dbtable.

#### Procedure to develop Entity Beans at the server

We should follow the following steps to develop the entity beans at the server.

(1) We should define one Entity class along with the attributes i.e. fields with respective the column names in the db-table.

(2) The metadata annotations that can be used along with this Entity class are ~~(@)~~ Entity, ~~(@)~~ Table, ~~(@)~~ Entity, ~~(@)~~ Table (name = "database tablename"), ~~(@)~~ id and so on.

(3) This Entity class must be implemented with Serializable interface available in java.io package.

(4) We can directly create this Entity class obj in the business logic form of the sessionbeans acc to appn req's.

(5) If we want to perform the db operations on this Entity obj's then we should inject the Entity Mgr

interface obj available at the persistence context of the EJB container such as

(a) PersistenceContext Ed.

```
EntityManager em;
```

(b) We can call the required funs of the EntityManager interface obj ac to app'n req's. such as

```
em.persist(emp);
```

```
em.remove(emp); and so on
```

(c) We should create one META-INF folder in the ejb app'n folder along with the XML file called as persistence.xml with the following content is

```
<persistence>
```

```
<persistence-unit>
```

```
<jta-data-source> myds </jta-data-source>
```

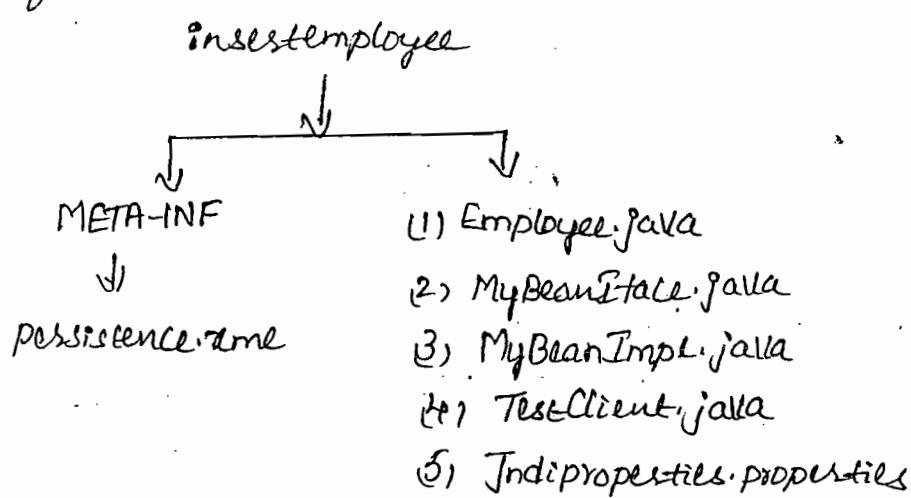
```
</persistence-unit>
```

```
</persistence>
```

Where jta stands for java transaction api.  
where myds is the jndi name of the datasource  
Obj already created at the sun app'n server by  
connecting to the connection pool called mypool.  
which is in turn connected to the oracle db.

(d) compile the java file to generate the class files  
for the EJB app'n, create the jar file along with  
the class files available in the package & the  
META-INF folder; deploy the jar file of the  
EJB app'n at the ap sun app'n server.

→ The following practical Ex: demo how to define the entity beans in the EJB app'n to insert the records into the DB table called Employee with 3 column names id, ename, esal by injecting Entity Mgr interface obj into the Session Bean component & by calling the methods of Entity Mgr interface obj available at the app'n server without writing any jdbc code in the business logic methods of the session bean component.



### Employee.java

This is the userdefined Entity class ok to the column names available in the dbtable called Employee

```

package entity;
import javax.persistence.*;
@Entity public class Employee implements Serializable
{
 @Id
 public int id;
 public String ename;
}

```

```
public float esal;
```

}

Where (a) `@Id` metadata annotation is used inside the entity class to provide the info to the EJB container that the `eno` attribute of this entity class is the primary attribute with respective corresponding db table called `employee`

### MyBeanInterface.java

```
package entity;
import javax.ejb.*;
(a) Remote public interface MyBeanInterface
```

{

```
 public void insert(int eno, String ename,
 float esal);
```

}

### MyBeanImpl.java

```
package entity;
import javax.ejb.*;
 " " . persistence.*;
(a) stateless public class MyBeanImpl implements
 MyBeanInterface
```

{

```
 (a) PersistenceContext EntityManager em;
 public void insert(int eno, String ename,
 float esal)
```

{

```
Employee emp = new Employee();
```

This will create an obj of Employee Entity class

```
emp.empno = empno;
```

```
emp.ename = ename;
```

```
emp.esal = esal;
```

```
emp.persist(emp);
```

Where persist is the method of EntityManager interface obj, this method automatically insert one new record into the db-table called Employee along with the current 3 values available in the respective Employee Entity class Obj.

Where @PersistenceContext metadata annotation is used to provide the info to the EJB container to inject the EntityManager interface obj available at the app'n server into the Session bean component.

Where em (em) is the attribute of this Session Bean component is now currently referencing to the EntityManager interface obj available at the app'n server.

### TestClient.java

```
import entity.MyBeanInterface;
import javax.naming.*;
import java.util.*;
import java.io.*;
```

```
public class TestClient
{
 public void main (String args[]) throws Exception
 {
 Properties p = new Properties();
 p.load (new FileInputStream ("judiproPERTIES-
 properties"));
 InitialContext ic = new InitialContext (p);
 MyBeanInterface mbi = (MyBeanInterface) ic.lookup
 ("entity. MyBeanInterface");
 while (true)
 {
 DataInputStream stdin = new DataInputStream
 (System.in);
 while (true)
 {
 System.out.print ("Enter Employee no:");
 int eno = Integer.parseInt (stdin.readLine ());
 System.out.print ("Enter Emp name");
 String ename = stdin.readLine ();
 System.out.print ("Enter Emp salary");
 float esal = float.parseFloat (stdin.readLine ());
 mbi.insert (eno, ename, esal);
 System.out.println ("Record inserted successfully");
 System.out.println ("Insert another record (Yes/No)");
 }
 }
 }
}
```

```
String choice = stdin.readLine();
if(!choice.equalsIgnoreCase("Yes"))
 break;
}
}
}
```

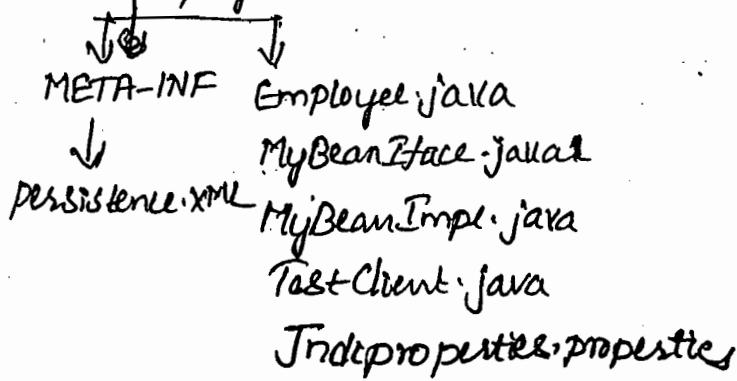
### Persistence.XML

```
<persistence>
<!-- unit -->
<jta-data-source> myds </jta-data-source>
</persistence-unit>
</persistence>
```

26-6-09

→ The following practical ex: demonstrate how to design the entity beans in the EJB app'n to select the record from the DBtable called Employee with 3 column names ie. eno, ername, eSal.

### SelectEmployee



### Employee.java

```
package entity;
import javax.persistence.*;
 " java.io.*;
```

① Entity public class Employee implements Serializable  
{

```
 @Id public int eno;
 public String ename;
 public float esal;
```

}

### MyBeanIface.java

```
package entity;
import javax.ejb.*;
```

② Remote public interface MyBeanIface

{

```
 public Employee getEmployee(int eno);
```

}

### MyBeanImpl.java

```
package entity;
import javax.ejb.*;
 " " . persistence.*;
```

③ stateless public class MyBeanImpl implements  
 MyBeanIface

{

```
(② Persistence Context EntityManager em;
public Employee getEmployee(int eno)
{
 Employee emp = (Employee) em.find(Employee.class,
 eno);
```

Where find is the method of EntityManager interface Obj that will check for the availability of the respective employee no: in the DB-table called Employee. If the employee no: is existed then it will return an Obj of Employee entity class along with the respective 3 values available in the table otherwise it will return null value.

```
return emp;
```

```
}
```

```
}
```

### TestClient.java

```
import entity.MyBeanInterface;
" " . Employee;
" " javax.naming.*;
" " java.util.*;
" " i18n.*;
```

```
class TestClient
```

```
{
```

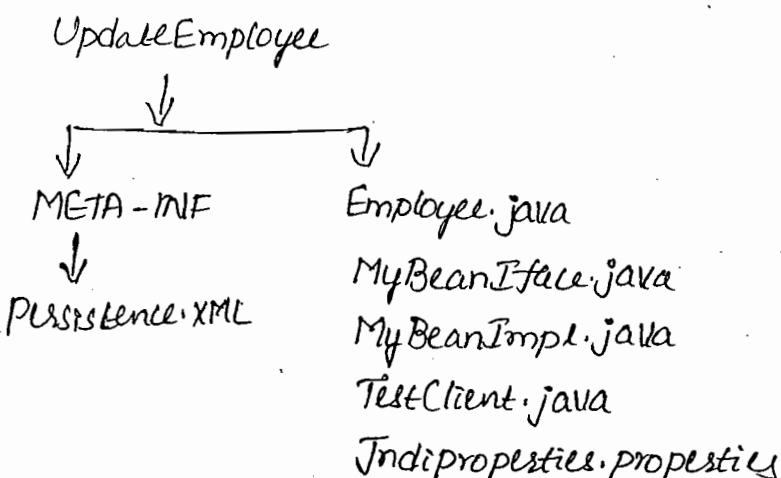
```
p.s.v.M(String args[])throws Exception
```

```
{
```

```
Properties p=new Properties();
```

```
p.load(new FileInputStream("jndiproperties.properties"));
InitialContext ic = new InitialContext(p);
MyBeanIface mbi = (MyBeanIface) ic.lookup("entity.
MyBeanIfaci");
DataInputStream stdin = new DataInputStream(Sys
em.in);
while(true)
{
 S.O.P ("Enter Employee no.");
 int eno = Integer.parseInt(stdin.readLine());
 Employee emp = mbi.getEmployee(eno);
 if(emp!=null)
 {
 S.O.P ("Employee details are - ");
 S.O.P ("Employee no: " + emp.eno);
 S.O.P ("Employee name " + emp.ename);
 " " " salary " + emp.esal);
 }
 else
 {
 S.O.P ("Employee no: does not exist");
 }
 S.O.P ("Select another record(yes/no)");
 String choice = stdin.readLine();
 if(!choice.equalsIgnoreCase("yes"))
 break;
```

→ The following practical example demonstrate how to design the Entity Beans in the EJB app'n to update the records in the db-table called Employee.



### Employee.java

```
package entity;
import javax.persistence.*;
import java.io.*;
```

① Entity public class Employee implements Serializable  
{

② Id public int eno;  
 " : String ename;  
 " float esal;

}

### MyBeanInterface.java

```
package entity;
import javax.ejb.*;
```

① Remote public interface MyBeanInterface

{}

```
public Employee updateEmployee(int eno, float amt);
}
```

### MyBeanImpl.java

```
package entity;
import javax.ejb.*;
" javax.persistence.*;
```

① Stateless public class MyBeanImpl implements  
MyBeanInterface

```
{
```

② Persistence Context EntityManager em;

```
public Employee updateEmployee (int eno, float amt)
```

```
{
```

```
Employee emp = (Employee)em.find (Employee.class, eno);
```

```
: if (emp != null)
```

```
{
```

```
emp.esal = emp.esal + amt;
```

This updated value in the Employee  
entity class obj is automatically reflected i.e.  
automatically updated in the corresponding record  
in the DBtable called Employee.

```
}
```

```
return emp;
```

```
}
```

```
}
```

## TestClient.java

```
import entity.MyBeanInterface;
 " Employee;
 " javax.naming.*;
 " java.util.*;
 " PO.*;

public class TestClient
{
 public static void main(String args[]) throws Exception
 {
 Properties p = new Properties();
 p.load(new FileInputStream("jndiproperties.properties"));

 InitialContext ic = new InitialContext(p);
 MyBeanInterface mbi = (MyBeanInterface) ic.lookup("Entity,
 MyBeanInterface");

 DataInputStream stdin = new DataInputStream
 (System.in);

 while(true)
 {
 System.out.print("Enter Employee no.");
 int eno = Integer.parseInt(stdin.readLine());
 System.out.print("Enter bonus amt");
 float amt = Float.parseFloat(stdin.readLine());
 Employee emp = mbi.updateEmployee(eno, amt);
 if(emp != null)
```

```
{
 S.O.P ("Employee details after updating are");
 S.O.P ("Employee no" + emp.eno);
 S.O.P ("Employee name" + emp.ename);
 S.O.P ("Employee salary" + emp.esal);
}
}
```

else

```
{
```

```
 S.O.P ("Employee no: does not exist");
}
```

```
}
```

```
 S.O.P ("update another record (yes/no)");
 String choice = stdin.readLine();
 if (!choice.equalsIgnoreCase(s))
 break;
 }
}
```

28-6-08

The following practical ex: demonstrate how to design the Entity Beans in the EJB app'n to insert a record into the DB table called Employee by passing the obj of Employee Entity class from the client JVM to the server JVM.

insertempbypassingemp → META-INF → persistence.xml  
↓

- (1) Employee.java      3, MyBeanImpl.java
- (2) MyBeanInterface.java      4, TestClient.java
- (3) IndigoProperties.properties

### Employee.java

```
package entity;
import javax.persistence.*;
" java.io.*";
```

① Entity public class Employee implements Serializable  
{

② Id public int empId;  
public String ename;  
public float esal;

}

### MyBeanInterface.java

```
package entity;
import javax.ejb.*;
```

① Remote public interface MyBeanInterface

{

```
 public void insertEmployee(Employee emp);
```

}

### MyBeanImpl.java

```
package entity;
import javax.ejb.*;
" " persistence.*;
```

① Stateless public class MyBeanImpl implements  
MyBeanInterface

{

```
(@PersistenceContext EntityManager em;
public void insertEmployee(Employee emp)
{
 em.persist(emp);
}
}
```

### TestClient.java

```
import entity.MyBeanIface
 " entity.Employee
 " javax.naming.*;
 " java.util.*;
 " io.*;

public class TestClient
{
 P.S.V.M (String args[])throws Exception
 {
 Properties p = new Properties();
 p.load(new FileInputStream("indiProperties.properties"));
 InitialContext ic = new InitialContext(p);
 MyBeanIface mbe = (MyBeanIface) ic.lookup
 ("entity.MyBeanIface");
 Employee emp = new Employee();
 DataInputStream stdin = new DataInputStream
 (stdin);
```

```

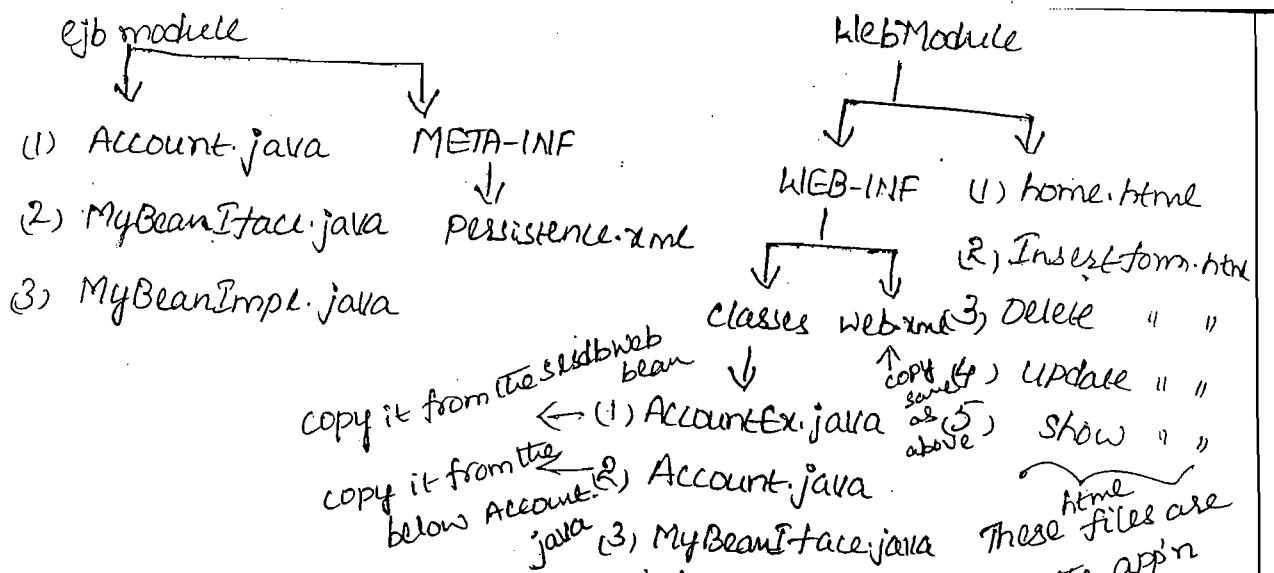
while(true)
{
 S.O.P ("Enter Employee no:");
 emp.eno = Integer.parseInt(stdin.readLine());
 S.O.P ("Enter Employee name");
 emp.ename = stdin.readLine();
 S.O.P ("Enter Employee salary");
 emp.esal = float.parseFloat(stdin.readLine());
 mbi.insertEmployee(emp);
 S.O.P ("Record inserted successfully");
 " (" Insert another record (yes/no)");
 String choice = stdin.readLine();
 if(!choice.equalsIgnoreCase("yes"))
 break;
}
}
}

```

→ The following practical ex: the communication b/w the web app'n & EJB app'n at the server to insert the record, to delete the record, to update the record into the DBtable called account with the column names accno, accname, accbal

Bankingweb ejb app'n





### Account.java

```
package entity;
import javax.persistence.*;
" java.io.*;
```

@Entity public class Account implements Serializable  
{

```
 @Id public int accno;
 public String accname;
 public float accbal;
```

}

### MyBeanIface.java

```
package entity;
import javax.ejb.*;
" annotation.*;
```

@Remote public interface MyBeanIface

{

```
public boolean insert (int accno, String accname,
 float accbal);

public boolean delete (int accno);

public boolean update (int accno, float amt);
public float getBalance (int accno);
```

}

### MyBeanImpl.java

```
package entity;
import javax.ejb.*;
" " . persistence.*;
```

① Stateless public class MyBeanImpl implements  
MyBeanIface

{

② PersistenceContext EntityManager em;  
public boolean insert (int accno, String accname,  
 float accbal)

{

```
 Account acc = new Account();
 acc.accno = accno;
 acc.accname = accname;
 acc.accbal = accbal;
 em.persist (acc);
 return true;
```

}

```
public boolean delete (int accno)
{
 boolean exist = false;
 try
 {
 Account acc = (Account) em.find (Account.class,
 accno);
 em.remove (acc);
 }
```

Where remove is the method of entity Manager Interface obj, this method will delete the respective record from the DB-table called account with respective the accno available in the respective a/c entity class obj, if there is no such record existing in the DB-table then this method will raise an exception.

```
exist = true;
```

```
}
```

```
catch (Exception e)
```

```
{
```

```
 exist = false;
```

```
}
```

```
return exist;
```

```
}
```

```
public boolean update (int accno, float amt)
```

```
{
```

```
 Account acc = (Account) em.find (Account.class,
 accno);
```

```
 if (acc != null)
```

```
{
```

```

acc.accbal = acc.accbal + amt;
return true;
}
return false;
}

public float getBalance(int accno)
{
 float bal = 0;
 Account acc = (Account) em.find(Account.class, accno);
 if(acc!=null)
 {
 bal = acc.accbal;
 }
 return bal;
}

```

~~30-6-09~~

### Entity Relations

The relationship of entities i.e. records in any two database tables can be defined with one of the following relationships i.e.

- (1) one-to-one
- (2) one-to-many (or)
- (3) many-to-one
- (4) many-to-many.

\* In this Entity relationships, one entity obj belongs to one DB table will have the relationship with a single entity object or multiple entity obj's belongs to some other DB table.

\* For ex: the relationship b/w customer & loan can be represented as one-to-one entity relationship, where one customer entity obj references to only one loan entity obj.

\* These customer, loan relationships can be defined either as unidirectional or bidirectional

\* These entity relationships are expressed in JPA i.e. (Java persistence API) with the following metadata annotations i.e.

### Relationship

### Annotation

one-to-one

ⓐ One-to-one ⓐ OneToOne

one-to-many

ⓐ OneToMany

many-to-one

ⓐ ManyToOne

many-to-many

ⓐ ManyToMany

### one-to-one relationship

The ⓐ OneToOne metadata annotation is used to represent uni or bi-directional one-to-one relationship b/w two entity classes in the ejb app'n.

The following 2 tables of data demonstrate this one-to-one relationship

### MyCustomer Table Data:

|             | (@) Id<br>cno | cname  | caddr      | clno | (@) Join column<br>clno |
|-------------|---------------|--------|------------|------|-------------------------|
| primary key | 101           | Ramesh | Ameerpet   | 1010 |                         |
|             | 102           | Suresh | Sanatnagar | 1020 |                         |
|             | 103           | Kiran  | Panjagutta | 1030 |                         |

Foreignkey

- \* We should treat this MyCustomer table as the owner table bcoz the foreignkey called clno stands for customer loan num: is available in this table.

### Myloan Table Data

| (@) Id<br>lno | primary key<br>lamt |
|---------------|---------------------|
|---------------|---------------------|

|      |       |
|------|-------|
| 1010 | 10000 |
| 1020 | 20000 |
| 1030 | 30000 |

- \* We should treat this Myloan table as non owner table bcoz there is no foreignkey available in this table.
- \* The above 2 tables of data demonstrate one to one relationship from mycustomer table to myloan table bcoz one of the entity ie. record of mycustomer table is having the relationship with only one entity of myloan table.
- \* Similarly one of the entity of Myloan table is having relationship with only one entity of mycustomer table.

\* We should define two entity classes in the EJB app'n to represent this one-to-one bidirectional relationship b/w mycustomer table & myloan table data as shown below

\* (1) Entity class for mycustomer table \*

(a) Entity

(a) Table (name = "mycustomer")

public class customer implements Serializable

{

(a) Id public int cno;

public String cname;

public String caddr;

public int clno;

(a) OneToOne

(a) JoinColumn (name = "clno")

public loan lr;

}

\* This entity class called customer is defined to map the owner table called mycustomer available at the db.

\* Where lr is the attribute of this customer entity class references to only one loan entity class

Obj.

\* Where `@JoinColumn` metadata annotation is used to specify the foreign key available in this current table called `mycustomer`.

\* (2) Entity class for myloan table

`@Entity`

`@Table(name = "myloan")`

public class Loan implements Serializable

{

`@Id public int lno;`  
`public float lamt;`

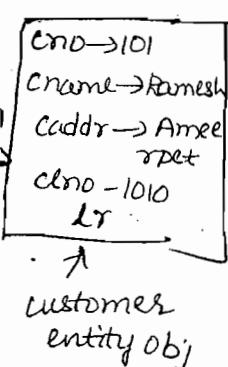
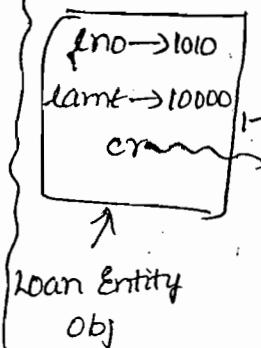
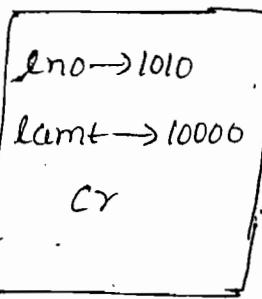
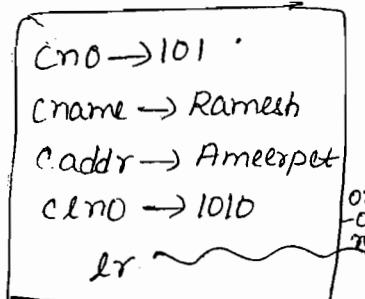
`@OneToOne(mappedBy = "lr") public Customer cr;`

}

\* This Entity class called `Loan` is defined to map the non-owner table called `myloan` available at the DB.

\* Where `mappedBy` is used to specify the reference field name i.e. attribute name available in the owner table entity class i.e. `Customer` class. Which is mapping the foreign key in the owner table i.e. `lno`

\* The following diagram demonstrate this one-to-one bidirectional relationship b/w the entity obj's belongs to MyCustomer and MyLoan tables available at the DB.



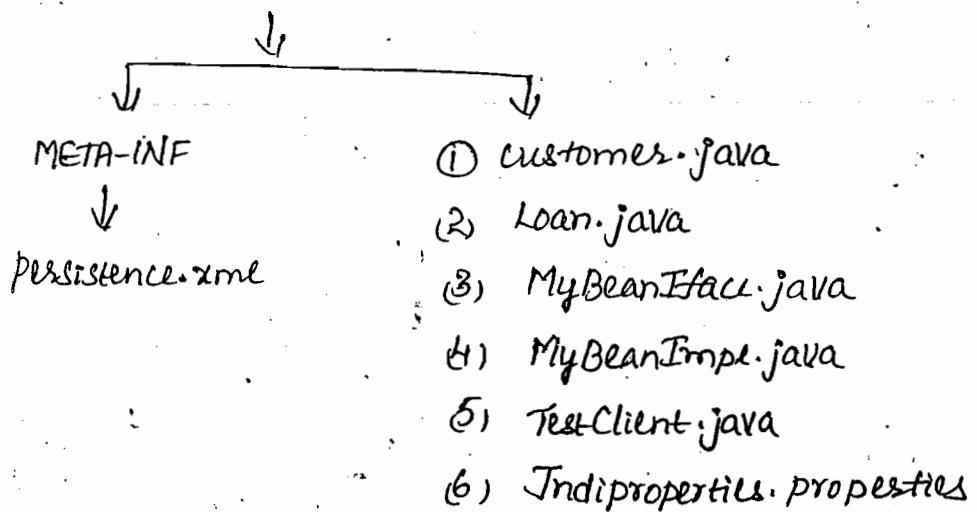
customer entity obj

① Relationship from customer entity obj  
to loan entity obj (one-to-one)

② Relationship from loan  
entity obj to customer  
entity obj (one-to-one)

The following practical ex. demonstrate this oneToOne bidirectional relationship b/w the entities of the two DB tables  
ie. mycustomers & myloan

### OneToOne



### customer.java

```

package entity;
import javax.persistence.*;

```

① Entity ① Table(name = "mycustomers") public class

Customer implements Serializable

{

① Id public int cno;  
public String cname;  
public String caddr;  
public int cno;

① OneToOne ① JoinColumn(name = "clno")

public Loan lr;

}

### Loan.java

```
package entity;
import javax.persistence.*;
" java.util.*;
```

① Entity ① Table(name = "myloan") public class Loan

implements Serializable

{

① Id public int lno;  
public float lamt;

① OneToOne(mappedBy = "lr") public Customer cr;

}

### MyBeanInterface.java

```
package entity;
import javax.ejb.*;
```

(②) Remote public interface MyBeanInterface

{

    public Customer getCustomerWithLoan(int cno);

    public Loan getLoanWithCustomer(int lno);

}

MyBeanImpl.java

package entity;

import javax.ejb.\*;

    // persistence.\*;

(③) Stateless public class MyBeanImpl implements  
                        MyBeanInterface

{

    @PersistenceContext EntityManager em;

    public Customer getCustomerWithLoan(int cno)

{

        Customer c = em.find(Customer.class, cno);

        return c;

}

    public Loan getLoanWithCustomer(int lno)

{

        Loan l = em.find(Loan.class, lno);

        return l;

}

## TestClient.java

```
import entity.Customer;
 " . Loan;
 " . MyBeanInterface;
 javax.naming.*;
 java.util.*;
 " . PO.*;

public class TestClient
{
 p.s.v.m (String args[]) throws Exception
 {
 Properties p = new Properties();
 p.load (new FileInputStream ("jndi.properties.
 properties"));

 InitialContext ic = new InitialContext (p);
 MyBeanInterface mbi = (MyBeanInterface) ic.lookup
 ("entity.MyBeanInterface");

 DataInputStream stdin = new DataInputStream
 (System.in);

 S.O.P ("Enter customer no:");
 int cno = Integer.parseInt(stdin.readLine ());
 Customer c = mbi.getCustomerWithLoan (cno);
 if (c != null)
 {
 S.O.P ("customer details are ...");
 S.O.P ("customer no:" + c.cno);
```

```
S.O.P ("Customer name" + c.cname);
S.O.P ("Customer address" + c.caddr);
S.O.P ("Loan alc.no:" + c.lc.eno); (2) (c.ceno);
S.O.P ("Loan amt: Rs." + c.lc.lamt); (2) (c.lamt)
}
else
{
 S.O.P ("Customer no: does not exist");
}

S.O.P ("Enter loan no:");
int lno = Integer.parseInt (stdin.readLine ());
Loan l = mbi.getLoanWithCustomer (lno);
if (l == null)
{
 S.O.P ("loan details are:");
 S.O.P ("loan no:" + l.lno);
 S.O.P ("loan amt: Rs." + l.lamt);
 S.O.P ("customer no:" + l.cr.cno);
 S.O.P ("Customer name:" + l.cr.cname);
 S.O.P ("Customer address" + l.cr.caddr);
}
else
{
 S.O.P ("loan no: does not exist");
}
```

1-7-09

## ManyToOne & OneToMany Relationship

- \* The (@)ManyToOne or (@)OneToMany meta data annotation are used for uni & bi-directional one-to-many & many-to-one relationship respectively.
- \* The following 2 tables of data demonstrate this many-to-one & one-to-many bi-directional relationship.

Myemp table data

| <u>(@) Id</u> | <u>primarykey</u> | <u>ename</u> | <u>sal</u> | <u>(@) JoinColumn (Foreignkey)</u> |
|---------------|-------------------|--------------|------------|------------------------------------|
| 101           |                   | Ramesh       | 10000      | 1010 *                             |
| 102           |                   | Suresh       | 20000      | 1020 #                             |
| 103           |                   | Kiran        | 30000      | 1010 *                             |
| 104           |                   | Ramarao      | 40000      | 1020 #                             |
| 105           |                   | AppaRao      | 50000      | 1030                               |

We should treat this myemp table as owner table bcoz the foreignkey called edno stands for employee deptno: is available in this table.

Mydept table data

| <u>(@) Id</u> | <u>(P.K.)</u> | <u>dname</u> | <u>loc</u>  |
|---------------|---------------|--------------|-------------|
|               | 1010          | Sales        | Ameerpet    |
|               | 1020          | Production   | Sanathnagar |
|               | 1030          | Marketing    | Yerragadda  |

We should treat this mydept table as non-owner-table bcoz there is no foreignkey available in this table.

- \* The above two tables of data demonstrate many-to-one relationship from myemp table to mydept table bcoz many of the entities ie records of myemp table are having the relationship with only one entity of mydept table.
- \* These tables demonstrate the one-to-many relationships from mydept table to myemp table bcoz one of the entity of mydept table is having relationship with many entities of myemp table.
- \* We should define two entity classes for this many-to-one & one-to-many bi-directional relationship b/w myemp table & mydept table data as shown below

(1) Entity class for myemp table:

① Entity ② Table(name = "myemp") public class  
Employee implements Serializable

{

③ Id public int emp;  
public String ename;  
public float sal;  
public int edno;

④ ManyToOne ⑤ JoinColumn(name = "edno")  
public Department dr;

}

one  
-02  
- (2) Entity class for mydept table

① Entity ② Table (name = "mydept") public class  
Department implements Serializable

{

③ Id public int dno;  
public String dname;  
" " dloc;

④ OneToMany (mappedBy = "dr", fetch = FetchType.EAGER)  
public Collection<Employee> empsr;

}

\* Where fetch = FetchType.EAGER is used to provide the info to the EJB container to search the entire owner table to select multiple entities from the owner table.

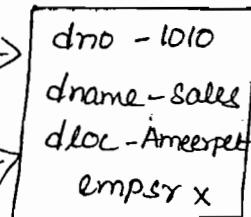
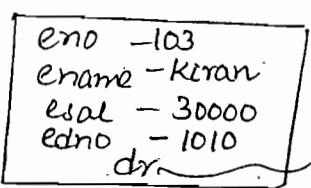
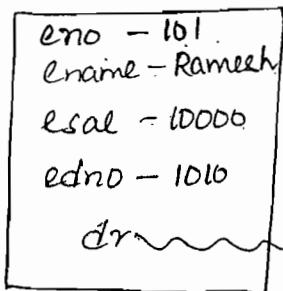
\* Where empsr is the attribute of this dept class references to a collection of employee entity Obj's

→ The following diagram demonstrate this many-to-one & one-to-many bi-directional relationship b/w myemp & mydept table data.

(1) ManyToOne Relationship b/w myemp table & mydept table

Note Many of employee entity class Obj's i.e. two employee entity class Obj are referencing to only one dept entity class Obj, so this relationship is called as many-to-one relationship.

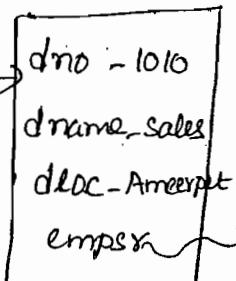
These are Employee Entity class obj's



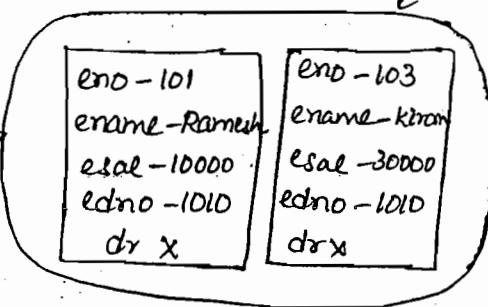
← dept entity class obj

Q) One To Many Relationship b/w mydept & myemp table.

Dept Entity class obj



collection of employee entity class obj's



Note One of the dept entity class obj is referencing to a collection of employee entity class obj's so this relationship is called as one-to-many relationships.

\* The following practical ex: demonstrate this many-to-one & one-to-many bi-directional relationship b/w myemp table & mydept table available in the DB

2/7/09

manytoone

META WEB-INF  
↓  
persistence.xml

1) Employee.java      4) MyBeanImpl.java  
2) Department.java      3) TestClient.java  
3) MyBeanInterface.java      6) JndiProperties.properties

## Employee.java

```
package entity;
import javax.persistence.*;
import java.io.*;

@Entity @Table(name = "myemp") public class Employee implements Serializable
{
 @Id public int eno;
 public String ename;
 @Column(name = "esal")
 public float esal;
 @Column(name = "edno")
 public int edno;

 @ManyToOne @JoinColumn(name = "edno")
 public Department dr;
}
```

## Department.java

```
package entity;
import javax.persistence.*;
import java.io.*;
import java.util.*;

@Entity @Table(name = "mydept") public class Department implements Serializable
{
 @Id public int dno;
 public String dname;
 @Column(name = "dloc")
 public String dloc;
```

① OneToMany (mappedBy = "dr", fetch = FetchType.EAGER)

```
public Collection<Employee> empsr;
```

```
}
```

### MyBeanInterface.java

```
package entity;
import javax.ejb.*;
" java.util.*;
```

② Remote public interface MyBeanInterface

```
{
```

```
public Employee getEmployeeWithDepartment(int empno);
" Department getDepartmentWithEmployees(int dno);
```

```
}
```

### MyBeanImpl.java

```
package entity;
import javax.ejb.*;
" javax.persistence.*;
" java.util.*;
```

③ Stateless public class MyBeanImpl implements  
MyBeanInterface

```
{
```

④ PersistenceContext EntityManager em;

```
public Employee getEmployeeWithDepartment(int empno)
```

```
{
```

```
P).
Employee emp = (Employee) em.find (Employee.class, eno);
return emp;
```

```
}
```

```
public Department getDepartmentWithEmployees (int dno)
{
```

```
Department d = em.find (Department.class, dno);
return d;
```

```
}
```

```
}
```

### TestClient.java

```
import entity.MyBeanInterface;
" " . Employee;
" " . Department;
" javax.naming.*;
" java.util.*;
" " . PO.*;
```

```
public class TestClient
```

```
{
```

```
public void main (String args[]) throws Exception
```

```
{
```

```
Properties p = new Properties();
```

```
p.load (new FileInputStream ("jndiproperties.
properties"));
```

```
InitialContext ic = new InitialContext (p);
```

```
MyBeanInterface mbi = (MyBeanInterface) ic.lookup("entity.
MyBeanInterface");
```

```
DataInputStream stdin = new DataInputStream(System.
in);
System.out.print("Enter employee no:");
```

```
int eno = Integer.parseInt(stdin.readLine());
```

```
Employee emp = mbi.getEmployeeWithDepartment(eno);
```

```
if(emp != null)
```

```
{
```

```
System.out.println("Employee details are . . .");
```

```
" " " no:"" + emp.eno);
```

```
" " " name" + emp.ename);
```

```
" " " salary" + emp.esal);
```

```
" " " Dept no:" + emp.dr.dno);
```

```
" " " name" + emp.dr.dname);
```

```
" " " location" + emp.dr.dloc);
```

```
}
```

```
else
```

```
{
```

```
System.out.println("Employee no: does not exist");
```

```
}
```

```
System.out.print("Enter dept. no:");
```

```
int dno = Integer.parseInt(stdin.readLine());
```

```
Department d = mbi.getDepartmentWithEmployee(dno);
```

```
if (d != null)
```

```
{
```

```
 s.o.p (" dept details are ");
 " (" " no:" + d.dno);
 " (" " name" + d.dname);
 " (" " location" + d.dloc);
```

```
Collection<Employee> cer = d.empsr;
```

```
s.o.p (" Employees in the dept are ");
```

```
for (Employee e : cer)
```

```
{
```

Where e is the reference variable of Employee class, this variable automatically references to all the individual Employee obj's one by one which are available in the respective collection of Employee obj's, so this for loop will display the details of all the Employees available in the respective collection of Employee obj's.

```
 s.o.p (" Employee no:" + e.eno);
 " (" " name" + e.ename);
 " (" " salary" + e.esal);
```

```
}
```

```
2
```

```
else
```

```
{
```

```
 s.o.p (" dept no: does not exist ");
```

```
3
```

```
? 3
```

3709

## Many-To-Many Relationship

The (@)ManyToMany meta data annotation is used for uni and bi-directional many-to-many relationship.

The following 2 tables of data demonstrate this many-to-many relationship.

### MyStudents Table Data

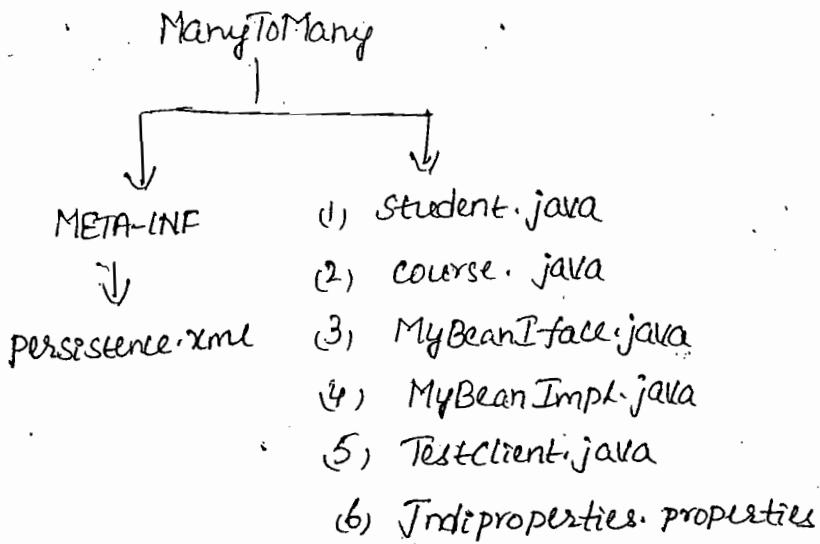
| <u>(@)Id</u> | <u>Sno</u> | <u>sname</u> |
|--------------|------------|--------------|
| 101          |            | Ramesh       |
| 102          |            | Suresh       |
| 103          |            | Kiran        |
| 104          |            | Anil         |

We can treat this MyStudents table as owner table.

### MyCourses Table Data

| <u>(@)Id</u> | <u>cno</u> | <u>cname</u> |
|--------------|------------|--------------|
| 1010         |            | Computers    |
| 1020         |            | Maths        |
| 1030         |            | Economics    |
| 1040         |            | History      |

We can treat this MyCourses table as non-owner table.



### Student.java

```

package entity;
import javax.persistence.*;
 " java.io.*;
 " " .util.*;

@Entity @Table(name = "mystudents") public class
Student implements Serializable
{
 @Id public int sno;
 " String sname;

 @ManyToMany(fetch = FetchType.EAGER)
 @JoinTable(name = "mystudents-mycourses")
 public collection<course> courses;
}

```

### Course.java

```

package entity;
import javax.persistence.*;
 " java.io.*;

```

my students.

having the relationship with many of the entities of  
similarity many of the entities in my courses are  
the relationship with many of the entities with  
both many of the entities of my students are having  
many-to-many relationship between my students & my courses

The above tables of date demands that to

specified in course entity class

specified in student entity class, and is the 3d attribute  
where course is the difference attribute now

specified in student entity class.

specified in course entity class, and is the 3d attribute  
where student is the difference attribute now

1030 104

1020 104

1040 103

1010 103

1040 102

1030 102

1020 102

1030 101

1010 101

course →

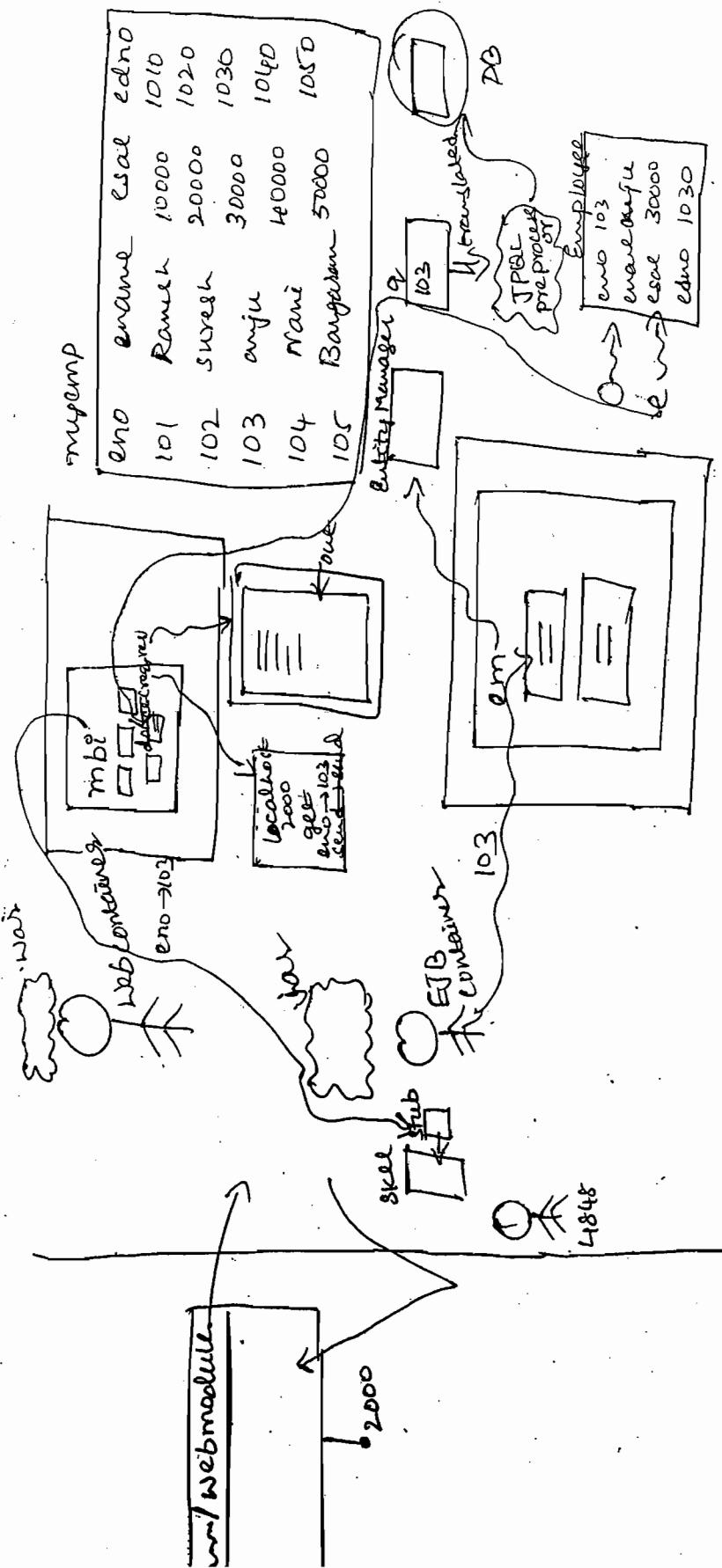
Student - sno Course - sno

Joint table called MyStudents - MyCourses table due

-shape as shown below

two tables to represent this many-to-many relation

Note we should create one joint table for the above



## web.xml

```
<Web-app>
< servlet >
< servlet-name > first < /servlet-name >
< servlet-class > EmpServlet < /servlet-class >
< /servlet >
< servlet >
< servlet-name > second < /servlet-name >
< servlet-class > LimitServlet < /servlet-class >
< /servlet >
< servlet-mapping >
< servlet-name > first < /servlet-name >
< url-pattern > / myfirstservlet < /url-pattern >
< /servlet-mapping >
< servlet-mapping >
< servlet-name > second < /servlet-name >
< url-pattern > / mysecondservice < /url-pattern >
< /servlet-mapping >
< welcome-file-list >
< welcome-file > home.html < /welcome-file >
< /welcome-file-list >
< /Web-app >
```

```
out.println ("Employee no:" + e.eno);
" ("

");
" (" Employee name" + e.ename);
" ("

");
" (" Employee salary" + e.esal);
" ("

");
" (" Employee dept no:" + e.edno);
" ("

");
count++;
```

{

The value of count is '0' if the above business logic method returns a collection of null Employee class obj's i.e. collection of no Employee obj's.

```
if (count == 0)
```

{

```
out.println ("Employees does not exist within the
given limit");
```

```
" ("

");
```

{

```
out.println (" HOME ");
" (" </center> </body> </html>");
```

{

{

```
public class LimitServlet extends HttpServlet
{
```

② EJB MyBeanInterface mbi;

```
public void doGet(HttpServletRequest req,
 " " Response res) throws ServletException
{
```

```
 res.setContentType ("text/html");
```

```
 PrintWriter out = res.getWriter();
```

```
 float lsal = Float.parseFloat(req.getParameter
 ("llimit"));
```

```
 float hsal = Float.parseFloat(req.getParameter
 ("hlimit"));
```

```
 Collection<Employee> cer = mbi.getEmployeesWithin
 limit(lsal, hsal);
```

This will call the respective Business logic method available in the respective bean component so that it will return a collection of Employee class obj's or collection of null obj's

```
 out.println ("<html> <body bgcolor=orange> ..
 <center>");
```

```
 " ("list of Employee are");
```

```
 " ("

");
```

```
 int count = 0;
```

```
 for (Employee e : cer)
```

```
{
```

```

out.println("

");

 " ("Employee salary" + e.esal);

 " ("

");

 " ("Employee dept no:" + e.edno);

 " ("

");

 }

 else

 {

 out.println(" Employee no: does not exist");

 " ("

");

 }

 out.println(" HOME ");

 " ("<center> <body> </body>");

}
}

```

### LimitServlet.java

The url pattern mapping of this HttpServlet class must be given as /mysecondservice, this will receive the limitform request from the client browser along with 3 parameters & values i.e. llimit, hlimit, send as get request.

```

import entity.MyBeanInterface;

 " .Employee;

 " javax.ejb.*;

 " .SERVLET.*;

 " " .http.*;

 " java.*;

```

```
public class EmpServlet extends HttpServlet
```

```
{
```

```
① EJB MyBeanInterface mbi;
```

```
public void doGet(HttpServletRequest req,
Http " Response res) throws ServletException
```

```
{
```

```
res.setContentType ("text/html");
```

```
PrintWriter out = res.getWriter();
```

```
int eno = Integer.parseInt (req.getParameter ("eno"));
```

```
Employee e = (Employee) mbi.getEmployee (eno);
```

Where mbi is the attribute of this servlet class which is referencing to the stub obj. of the EJB component which is injected into this servlet component

This will call the respective business logic method of the respective bean component available at the EJB app'n so that it will return one Employee class obj along with the details of the respective Employee no, if such Employee no: does not exist in the DB table then it will return null value.

```
out.println ("<html> <body bgcolor=cyan>
<center>");
```

```
if (e != null)
```

```
{
```

```
out.println ("Employee details are");
```

```
" ("

");
```

```
" ("Employee no" + e.eno);
```

```
" ("

");
```

```
" ("Employee name" + e.ename);
```

### Limitform.html

```
<html>
<body bgcolor=cyan>
<center>
<form method=get action=/webmodule/mysecond servletl_limith_limit
```

### EmpServlet.java

The url pattern mapping of this HttpServlet class must be given as /myfirst servlet, this will receive &employee select form request from the client browser with 2 parameters & values i.e. end, send as get request.

```
import entity.MyBeanInterface;
" " .Employee;
" javax.ejb.*;
" " .servlet.*;
" " " http.*;
```

## home.html

```
<html>
<body bgcolor = wheat>
<center>
 Main Menu

 SELECT EMPLOYEE

 SELECT EMPLOYEES WITHIN
 SALARY LIMIT
</center>
</body>
</html>
```

## Selectform.html

```
<html>
<body bgcolor = yellow>
<center>
<form method = get action = "/webmodule/myfirstserv
 -let>
 Enter Employee no:
<input type = text name = eno>

<input type = submit name = send value = SEND>
</form>
</center>
</body>
</html>
```

```
try
```

```
{
```

```
 O = q.getSingleResult();
```

This will raise an exception if there is no such Employee no: exist in the respective DBtable i.e. myemp table.

```
}
```

```
catch (Exception e)
```

```
{
```

```
 return null;
```

```
}
```

```
Employee e = (Employee) O;
```

```
return e;
```

```
}
```

```
public Collection<Employee> getEmployeesWithinLimit.
```

```
(float lowlimit, float highlimit)
```

```
{
```

```
Query q = em.createQuery("Select e from Employee e
where e.salary Between :minsalary
AND :maxsalary");
```

```
q.setParameter("minsalary", lowlimit);
```

```
q.setParameter("maxsalary", highlimit);
```

```
return q.getResultList();
```

```
}
```

```
}
```

## MyBeanInterface.java

```
package entity;
import javax.ejb.*;
 " java.util.*;
```

(@) Remote public interface MyBeanInterface

{

```
public Employee getEmployee(int eno);
public Collection<Employee> getEmployeesWithinLimit
 (float lowlimit, float highlimit);
```

}

## MyBeanImpl.java

```
package entity;
import javax.ejb.*;
 " " . persistence.*;
 " " . util.*;
```

(@) Stateless public class MyBeanImpl implements
 MyBeanInterface

{

(@) Persistence Context EntityManager em;

```
public Employee getEmployee(int eno)
```

{

```
Query q = em.createQuery("Select e from
 Employee e where e.eno = ?1");
```

```
q.setParameter(1, eno);
```

```
Object o;
```

3

① Entity ② Table (name = "mycourses") public class  
Course implements Serializable

{

③ Id public int cno;

→ String cname;

④ ManyToMany (mappedBy = "courseer", fetch = FetchType.EAGER)

public collection<Student> Students;

}

### MyBeanInterface.java

57 pg  
package entity;  
import javax.ejb.\*;  
" java.util.\*;

⑤ Remote public interface MyBeanInterface

{

public Student getStudentWithCourses(int sno);  
" Course getCourseWithStudents(int cno);

}

### MyBeanImpl.java

package entity;  
import javax.ejb.\*;  
" java.util.\*;  
" javax.persistence.\*;

⑥ Stateless public class MyBeanImpl implements  
MyBeanInterface

{

~~TYPE~~  
~~ER~~

(a) PersistenceContext EntityManager em;

public Student getStudentWithCourses (int sno)

{

    Student s = em.find (Student.class, sno);

    return s;

}

public Course getCourseWithStudents (int cno)

{

    Course c = em.find (Course.class, cno);

    return c;

}

}

### TestClient.java

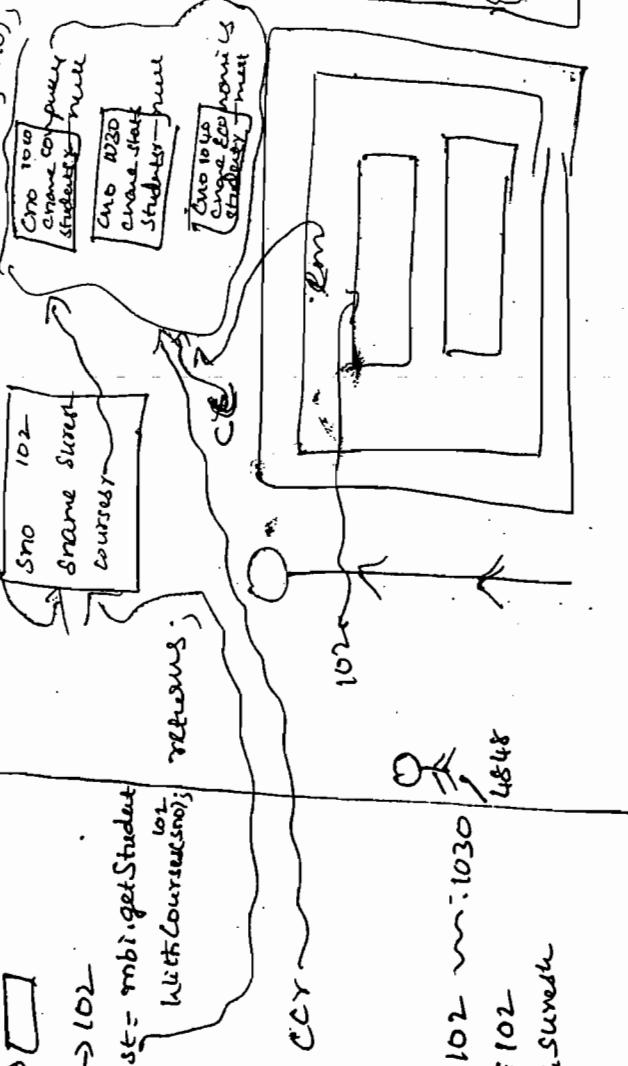
```
import entity.MyBeanInterface;
" " .course;
" " .Student;
" " java.io.*;
" " javax.naming.*;
" " java.util.*;
public class TestClient
{
 public static void main (String args[]) throws Exception
 {
```

```
Properties p = new Properties();
p.load(new FileInputStream ("jndiproperties.properties"));
InitialContext ic = new InitialContext(p);
MyBeanIface mbi = (MyBeanIface) ic.lookup ("entity.
 MyBeanIface");
DataInputStream stdin = new DataInputStream (System.
 .in);
s.o.p ("Enter student no:");
int sno = Integer.parseInt (stdin.readLine ());
Student st = mbi.getStudentWithCourse (sno);
if (st != null)
{
 s.o.p ("Student details are:");
 s.o.p (" " + "no:" + st.sno);
 s.o.p (" " + "name" + st.sname);
 Collection<Course> ccr = st.courses;
 s.o.p ("Student courses details are");
 s.o.p
 for (Course c:ccr)
 {
 s.o.p ("Course no:" + c.cno);
 s.o.p (" " + "name" + c cname);
 }
}
```

```
else
{
 S.O.P ("Student no: does not exist");
}

S.O.P ("Enter course no:");
int cno = Integer.parseInt (stdin.readLine ());
Course c = mbi.getCourseWithStudents (cno);
if (c != null)
{
 S.O.P ("course details are");
 S.O.P ("course no:" + c.cno);
 S.O.P ("course name:" + c cname);
 Collection<Student> csr = c.students;
 S.O.P ("course students details are:");
 for (Student s: csr)
 {
 S.O.P ("student no:" + s.sno);
 S.O.P ("student name:" + s.sname);
 }
}
else
{
 S.O.P ("course no: does not exist");
}
```

student<sup>s</sup> = em. find (Studentic Class<sup>202</sup>, sno)



四

$\text{SnO} \rightarrow \text{LO}_2$

Student  $\leftarrow$  mbig.getStudent  
with  $\{$  Course  $\}$ ;

mysteries

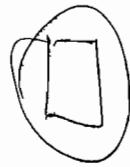
| Sno | name   |
|-----|--------|
| 101 | Ramesh |
| 102 | Deepak |
| 103 | Kiran  |

my course  
one

|         |                      |
|---------|----------------------|
| 100     | one hundred          |
| 1000    | one thousand         |
| 10000   | one ten thousand     |
| 100000  | one hundred thousand |
| 1000000 | one million          |

### Mysteries - My courses

| Studentino | Corsettino |
|------------|------------|
| 101        | 1020       |
| 101        | 1040       |
| 102        | 1070       |
| 102        | 1030       |
| 102        | 1040       |
| 103        | 1030       |



6

米

米

六

$\sim: 102 \sim: 1030$   $\sim: 102$   $4548$

: 102 ~  
: 102 ~  
: 102 ~

~~6.7.09~~  
JPQL

## (Java Persistence Query Language)

- \* JPQL is used to create queries to retrieve the data from the DB tables.
- \* The JDBC SQL query will return one ResultSet interface obj along with the values of DB records.
- \* Whereas JPQL query will return collection of Java obj's.
- \* In JDBC SQL query we should use collection interface obj to create Statement interface obj, whereas in JPQL query we should use EntityManager interface obj to create Query Interface obj.
- \* In JDBC SQL query we will send the required SQL code directly to the DB through Statement interface obj, whereas in JPQL query we will send JPQL code to the EntityManager interface obj through query interface obj, so that the entity manager will translate the JPQL code into the required SQL code & then sent to the DB.
- \* The following are the some of JPQL queries that can be used in the EJB app's to retrieve the data from the DB tables i.e.

1) Query q = em.createQuery("select e from Employee e");  
return q.getResultList();

Where Query is the predefined interface available in javax.persistence package, where createQuery is the method of EntityManager interface obj that will automatically create one Query interface obj along with the respective JPQL code.

Where `getResultSet` is the method of `Query` interface obj that will execute the respective query & return a collection of `Employee` class obj's or a collection of null obj's

(2) Query q = em.createQuery("select e.eno from Employee e");  
return q.getResultSet();

This will return a collection of Integer class obj's along with the employee no's available in the respective DB table.

(3) Query q = em.createQuery("Select e.eno, e.ename  
from Employee e");  
return q.getResultSet();

This will return a collection of obj's array along with the Employee no's, Employee names available in the respective DB table.

(4) Query q = em.createQuery("Select e from Employee  
e where e.eno = ?");

q.setParameter(1, 50);

Object o = q.getStringSingleResult();

This will raise an exception if there is no such record available in the DB-table that satisfies the specified condition

Employee e = (Employee)o;

where ?1 is called as numbered parameters.

5) Query q = em.createQuery("select e from Employee  
e where e.esal between :minsal  
-ary AND :maxSalary");

q.setParameter("minsalary", 10000);

q.setParameter("maxsalary", 20000);

return q.getResultList();

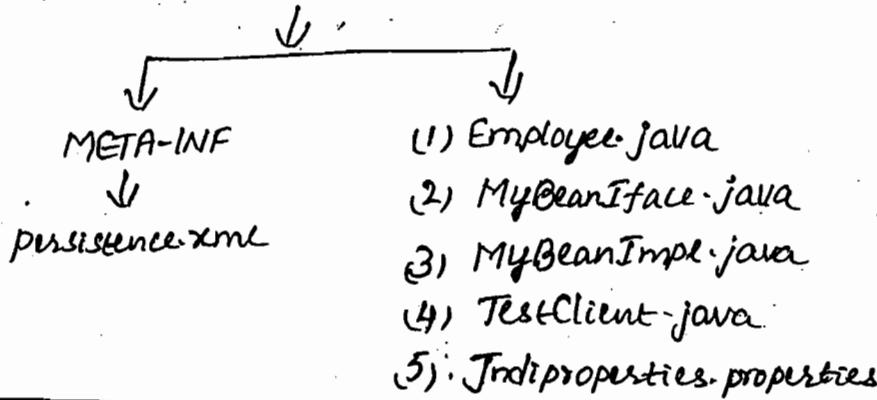
This will return a collection of Employee class  
Obj's.

Where minsalary, maxsalary are called as named  
parameters.

Note Each JPQL query is translated into SQL query  
by the JPQL preprocessor and then sent to the DB,  
where the JPQL preprocessor is supplied by Vendor of  
the app'n server.

→ The following practical ex: demonstrate how to design the  
bean components in the EJB app'n to retrieve the data  
from the DB-table called myEmp by using JPQL  
query language.

JPQL select



## Employee.java

```
package entity;
```

```
import javax.persistence.*;
" java.io.*;
```

```
@Entity @Table(name = "myemp") public class
```

```
Employee implements Serializable
```

```
{
```

```
 @Id public int eno;
 public String ename;
 public float sal;
 public int edno;
```

```
}
```

## MyBeanInterface.java

```
package entity;
```

```
import javax.ejb.*;
" java.util.*;
```

```
@Remote public interface MyBeanInterface
```

```
{
```

```
 public Collection<Employee> getAllEmployees();
```

```
 public Collection<Integer> getAllEmpnos();
```

```
 public Collection<Object[]> getAllEmpnosWithNames
```

```
();
```

## MyBeanImpl.java

```
package entity;
```

```
import javax.ejb.*;
 " persistence.*;
 " java.util.*;
```

(a) Stateless public class MyBeanImpl implements  
MyBeanInterface

{

```
@PersistenceContext EntityManager em;
public Collection<Employee> getAllEmployees()
{
 Query q = em.createQuery("select e from
 Employee e");
 return q.getResultList();
}
public Collection<Integer> getAllEmpnos()
{
 Query q = em.createQuery("select e.eno from
 Employee e");
 return q.getResultList();
}
public Collection<Object[]> getAllEmpnosWithNames()
{
 Query q = em.createQuery("select e.eno,
 e.ename from Employee e");
 return q.getResultList();
}
```

{

## TestClient.java

```
import entity.MyBeanInterface;
// " + Employee;
// javax.naming.*;
// java.util.*;
// " + EO.*;

public class TestClient
{
 public static void main (String args[]) throws Exception
 {
 Properties p = new Properties();
 p.load (new FileInputStream ("jndi.properties"));
 InitialContext ic = new InitialContext (p);
 MyBeanInterface mbi = (MyBeanInterface) ic.lookup
 ("Entity.MyBeanInterface");
 Collection<Employee> cer = mbi.getAllEmployees();
 System.out.println ("list of all the Employees");
 for (Employee e:cer)
 {
 System.out.println (e.geteno());
 System.out.println (e.getename());
 // System.out.println (e.getesal());
 System.out.println (e.getedno());
 System.out.println ("....");
 }
 }
}
```

```
Collection<Integer> cir = mbi.getAllEmpnos();
```

```
S.O.P ("List of all the Employee no.s");
```

```
for (Integer i: cir)
```

```
{
```

```
 S.O.P (i);
```

```
}
```

```
Collection <Object[]> coar = mbi.getAllEmpnoWith
Names();
```

```
S.O.P ("list of the Empno's, Names");
```

```
for (Object[] oa: coar)
```

```
{
```

```
 S.O.P (oa[0] + "E" + oa[1]);
```

```
}
```

```
}
```

Q/P list of all the Employee

101

Anil

list of all the Empno's

101

102

103

104

list of all the Empno's with names

101 Anil

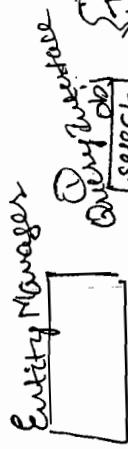
102

103

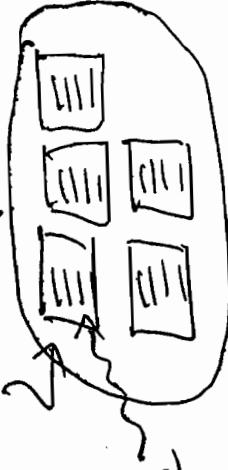
104

mbirey

Entity Manager



Collection < Employee >  
cse = mbirey.getallEmployees();



Collection < Integer >

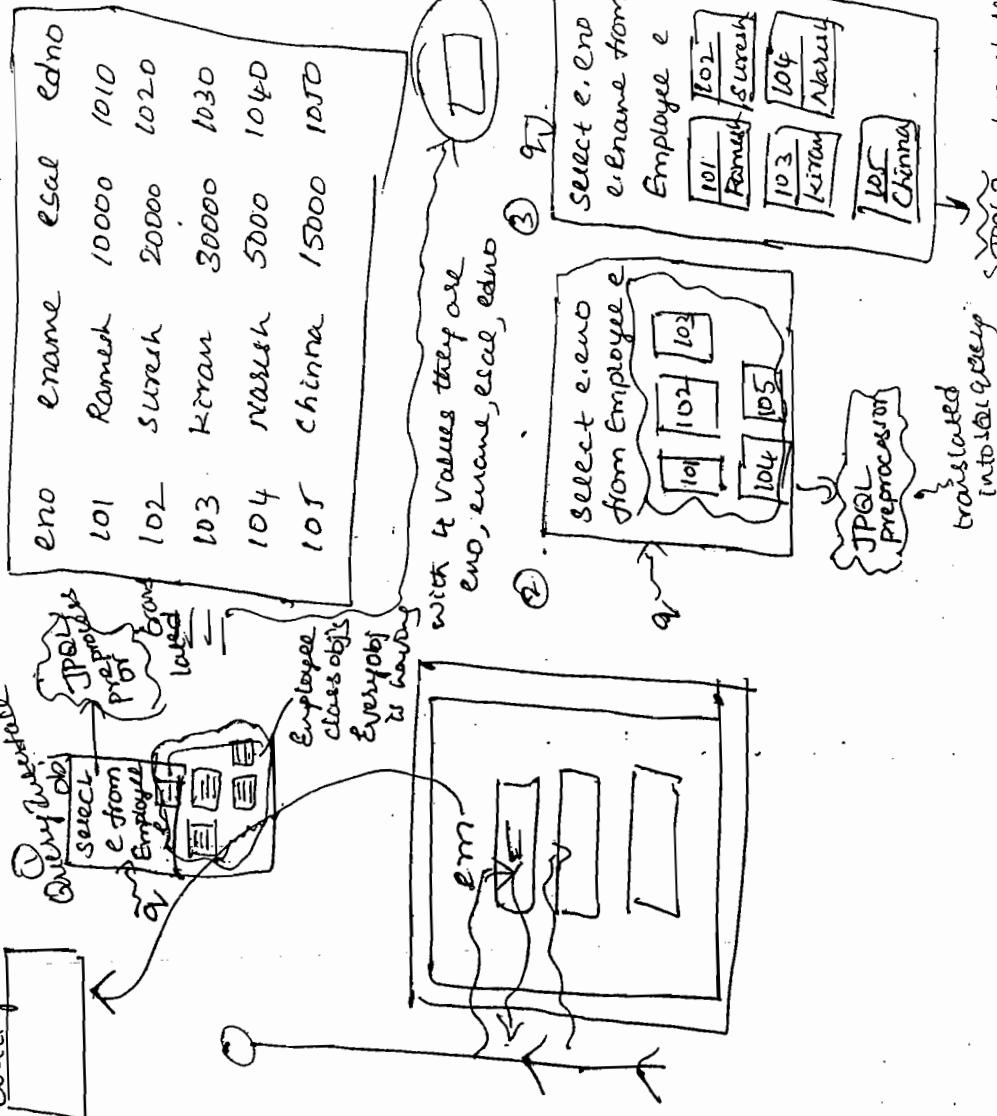
cis = mbirey.getAllEmpNo();  
cis = [101, 102, 103, 104]



Collection < Object[] > →  
coar = mbirey.getAllEmployees();  
coar = [Employee Name: Ramesh, eno: 101, esal: 10000, edno: 1010]  
or  
coar = [Employee Name: Suresh, eno: 102, esal: 20000, edno: 1020]  
or  
coar = [Employee Name: Kiran, eno: 103, esal: 30000, edno: 1030]  
or  
coar = [Employee Name: Naray, eno: 104, esal: 50000, edno: 1040]  
or  
coar = [Employee Name: Chinna, eno: 105, esal: 15000, edno: 1050]



$$da[0] = 101$$



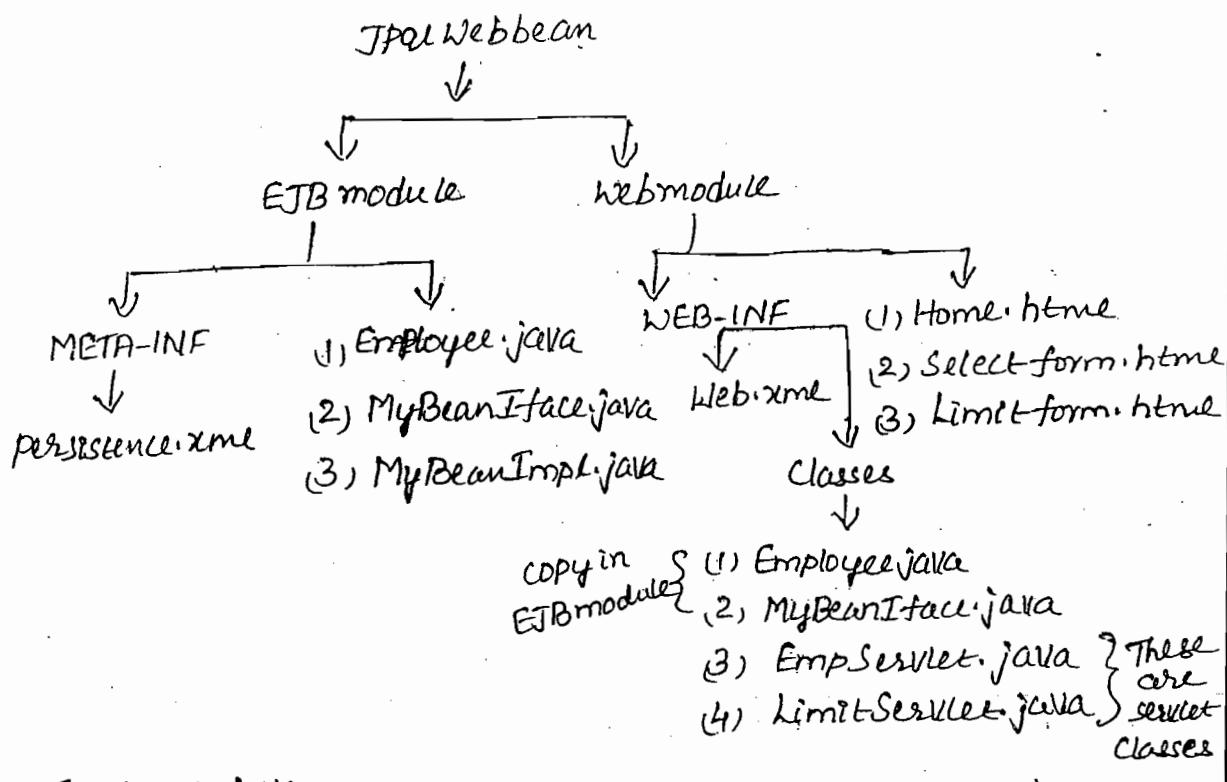
JPQL untranslated  
into SQL query  
DB

↓

117.

7-7-09

→ The following practical Ex: demonstrate the communication b/w <sup>the</sup> Web app'n & the EJB app'n by using JPQL code in the bean component to retrieve the data from the DB table called myemp.



### Employee.java

```
package entity;
import javax.persistence.*;
import java.io.*;

@Entity @Table(name = "myemp") public class Employee
 implements Serializable
{
 @Id public int eno;
 String ename;
 float basic;
 int edno;
```

## SavingsAccount.java

```
package entity;
```

```
import javax.persistence.*;
" java.io.*;
```

```
@Entity public class SavingsAccount implements
```

Serializable

```
{
```

```
@Id public int saccno;
```

```
public String saccname;
```

```
" float saccbal;
```

```
}
```

## CurrentAccount.java

```
package entity;
```

```
import javax.persistence.*;
```

```
" java.io.*;
```

```
@Entity public class CurrentAccount implements
```

Serializable

```
{
```

```
@Id public int caccno;
```

```
public String caccname;
```

```
public float caccbal;
```

```
}
```

## MySavingsBeanInterface.java

```
package entity;
```

```
import javax.persistence.*;
```

② Local public interface MySavingsBankInterface

۳

```
public void deduct (int saccno, float amt);
```

## MySavings Bean Impl. java

```
package entity;
import javax.ejb.*;
" " " persistence.*;
```

④ Stateless public class MySavingsBeanImple implements MySavingsBeanIface

۸

④ Persistence Context EntityManager em;

public void deduct (int sacno, float amt)

۸

SavingsAccount sacc = em.find(SavingsAccount.class, saccno);

$$\text{sacc.saccbal} = \text{sacc.saccbal-amt};$$

3

3

## MyCurrentBeanIface.java

package entity;

```
import javax.ejb.*;
```

(a) Local public interface MyCurrentBeanFace

۸۷

```
public void deposit(int accno, float amt);
```

3

### MyCurrentBeanImpl.java

```
package entity;
import javax.ejb.*;
 " . persistence.*;
```

② Stateless public class MyCurrentBeanImpl implements  
MyCurrentBeanInterface

{

③ PersistenceContext EntityManager em;

```
public void deposit(int accno, float amt)
```

{

```
 CurrentAccount cacc = em.find(CurrentAccount.
 class, accno);
```

```
 cacc.caccbal = cacc.caccbal + amt;
```

}

}

### MyTransferBeanInterface.java

```
package entity;
```

```
import javax.ejb.*;
```

④ Remote public interface MyTransferBeanInterface

{

```
 public boolean transferAmount(int saccno,
 int caccno, float amt);
```

}

### MyTransferBeanImpl.java

```
package entity;
```

```
import javax.persistence.*;
```

```
import javax.persistence.*;
 " " . transaction.*;
 " " . annotation.*;
```

② Stateless ② TransactionManagement (Value = Transaction  
ManagementType.BEAN)

Public class MyTransferBeanImpl implements  
MyTransferBeanInterface

{

The above ② TransactionManagement metadata  
annotation which is available in javax.transaction  
package is used to provide the info to the EJB container  
that the transactionManagement service available at  
app'n server is used in the current bean class.

② PersistenceContext EntityManager em;

② EJB MySavingsBeanInterface msbi;

This is used to inject the stub of the  
respective local bean component into this Mainbean  
component.

② EJB MyCurrentBeanInterface mcbi;

② Resource UserTransaction ut;

This is used to inject the UserTransaction  
interface obj available at the resource ctr of the  
app'n server into this mainbean component.

```
public boolean transferAmount(int saccno, int caccno,
float amt)
```

{

```
boolean transferdone = false;
```

```
try
```

```
{
```

```
ut.begin();
```

```
msbi.deduct(saccno, amt);
```

This will call the respective deduct business logic method available in the "localbean component", if there is no such savings A/c no. exist in the DBtable then it will raise an exception. i.e. It will receive an exception.

```
mcbi.deposit(caccno, amt);
```

This will call the respective bco2 its the business logic method available in the "another local bean component", if there is no such current a/c no. exist in the DB table then it will receive an exception.

```
ut.commit();
```

This is executed if there are no exceptions are raised in this try block so that the above 2DB operations are committed into the respective DB tables.

```
transferdone = true;
```

```
}
```

```
catch(Exception e1)
```

```
{
```

```
e1.printStackTrace();
```

```
try
```

```
{
```

```
ut.rollback();
```

This is executed when there is an

exception is raised in the above try block so that some of  
the successfully performed operations which are available  
in the buffer are canceled so that there is no data  
inconsistency in the respective DB tables.

```
 }
catch(Exception e2)
{
}
}
}
}
return transferdone;
}
}
```

### transferform.html

```
<html>
<body bgcolor = "cyan">
<center>

<form method = "get" action = "/Webmodule/myservlet">
Enter savings acc no:
<input type = "text" name = "saccno" >

Enter current acc no:
<input type = "text" name = "caccno" >

Enter transfer amt:
```

one of  
Table

```
<input type="text" name="amt">

<input type="submit" name="send" value="SEND">
</form>
</center>
</body> </html>
```

### TransferServlet.java

The url pattern mapping of this HttpServlet class must be /myservlet, this will receive the transferform request with 3 parameters & values i.e. sacno, cacno, amt, send as get request.

```
import entity.MyTransferBeanInterface;
```

```
" javax.ejb.*;
" " .Servlet.*;
" " " .http.*;
" java.io.*;
```

```
public class TransferServlet extends HttpServlet
```

```
{
```

```
② EJB MyTransferBeanInterface mtbi;
```

This is used to inject the stub of the respective main bean component available at the EJB container into this servlet component.

```
public void doGet(HttpServletRequest req,
" " " Response res) throws
```

SE, IOException

```
{
```

```
res.setContentType("text/html");
```

```
PrintWriter out = res.getWriter();
int saccno = Integer.parseInt(req.getParameter("saccno"));
int caccno = Integer.parseInt(req.getParameter("caccno"));
float amt = Float.parseFloat(req.getParameter("amt"));
out.println("<html><body bgcolor=wheat><center>");
"

");
```

```
if (mtbi.transferAmount(saccno, caccno, amt) == true)
{
```

This will call the respective business logic method available in the respective mainbean component at the EJB container of the app'n server.

```
out.println(" Amt transferred successfully");
"

");
```

```
" click
me for one more transaction ");
```

```
}
```

```
else
```

```
{
```

```
out.println(" Amt transfer failed");
```

```
"

");
```

```
" click
me for one more transaction ");
```

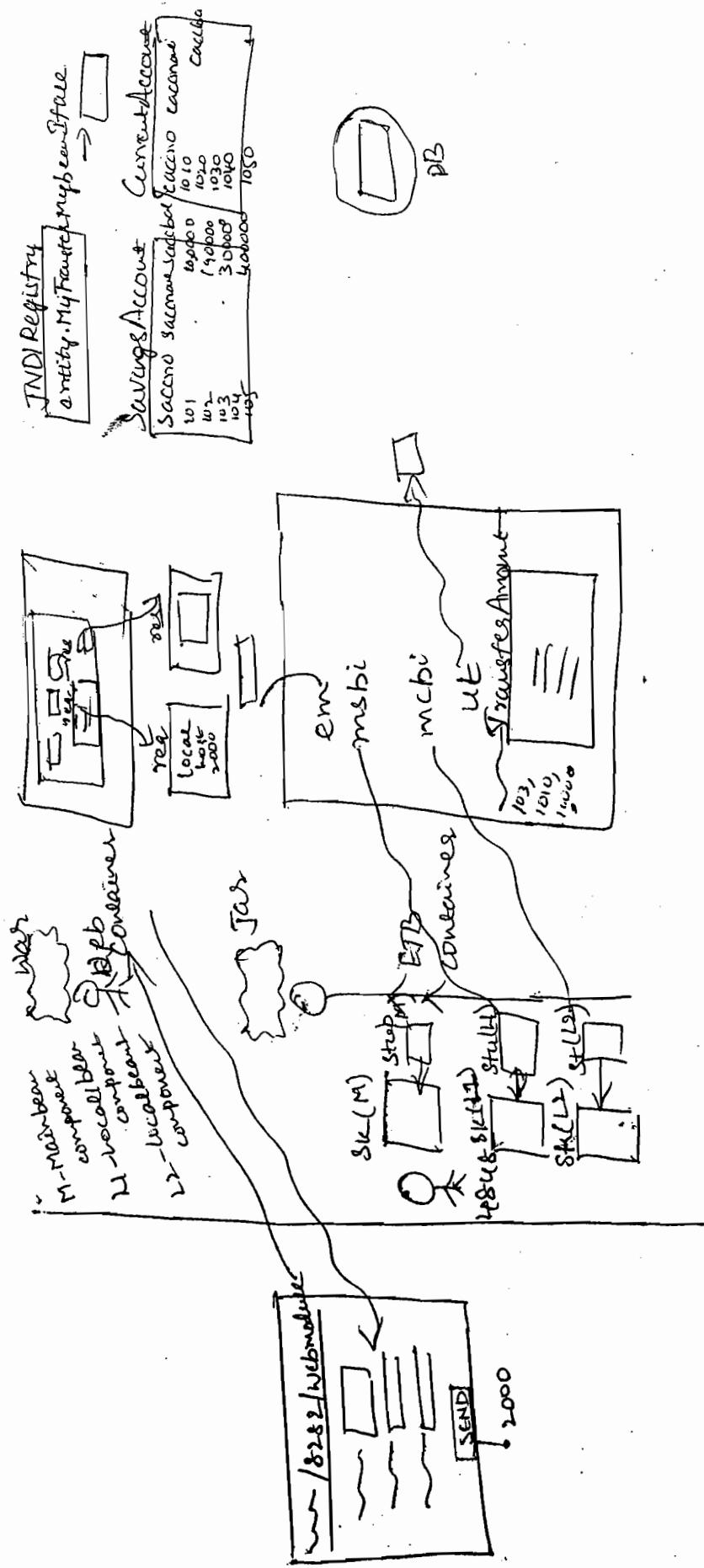
```
}
```

```
out.println("</center></body></html>");
```

```
}
```

## Web.xml

```
<web-app>
 <servlet>
 <servlet-name> First </servlet-name>
 < " - class > Transfer service </servlet-class >
 </servlet>
 <servlet-mapping>
 <servlet-name> first </servlet-name>
 <url-pattern> /myservlet </url-pattern>
 </servlet-mapping>
 <welcome-file-list>
 <welcome-file> transfer form.html </welcome-file>
 </welcome-file-list>
</web-app>
```



10-7

\*

\*

81

\* 10.10.10

10.10.10

8-7-09

## Database Transaction Management

- \* The default mode of the connection established to the DB from the EJB app'n is called as auto-commit mode. In this auto-commit mode, the execution result of every SQL command that we send to the DB is automatically committed i.e. automatically reflected into the DB tables.
- \* In this auto-commit mode we can't cancel the executed results at the DB.
- \* If we want to have the option of either committing the result or canceling the result at the DB then the connection mode has to be changed to non auto-commit mode.
- \* In this non-auto-commit mode the DBEngine will allocate one buffer for the app'n so that every executed SQL command result is temporarily stored into this buffer at the DB.
- \* If we want to commit all the results of the buffer into the physical DB tables then we should send the commit signal to the DB from the EJB app'n.
- \* If we want to cancel all the results of the buffer then we should send one 'rollback signal' to the DB so that the DBEngine will clear the buffer without committing those results into the physical DBtables.
- \* A set of DB operations are called as a single DB transaction.

\* We want to perform all the DB operations or none of the DB operations available in the DB transactions so that there is no data inconsistency in the DB tables.

\* This process of performing all the operations are non of the operations of the DB transaction in the app'n is called as DBtransaction management.

\* This transaction mgmt is only possible in nonauto-commit mode.

\* If we want to manage the transactions at the DB from the EJB app'n then we should inject UserTransaction interface obj available at Resource of the app'server into the bean component by using the following metadata annotation. i.e.

(a) Resource UserTransaction ut;

\* We should use the following javacode in the business logic methods to change the connection mode into non auto commit mode i.e. ut.begin(); Where begin is the method of UserTransaction interface obj that will change the connection mode to the DB as non auto commit mode so that the DBEngine will allocate to one buffer for the EJB app'n

\* We should use the following javacode send the commit signal to the DB i.e.

ut.commit();

Where commit is the method of UserTransaction interface obj that will send one commit signal to the DB so that the DBEngine will commit all the results of the buffer into the DB tables & then the buffer is cleared.

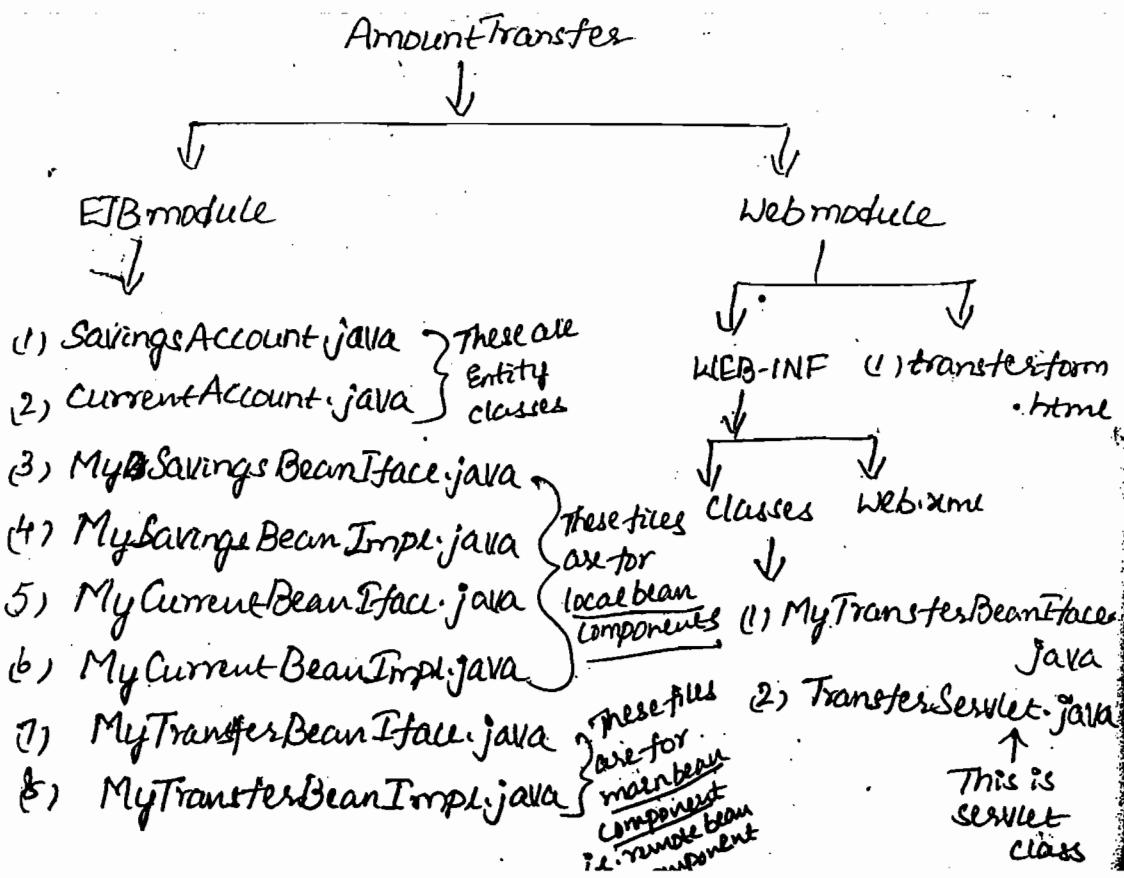
\* We should use the following java code to cancel the results of the buffer.

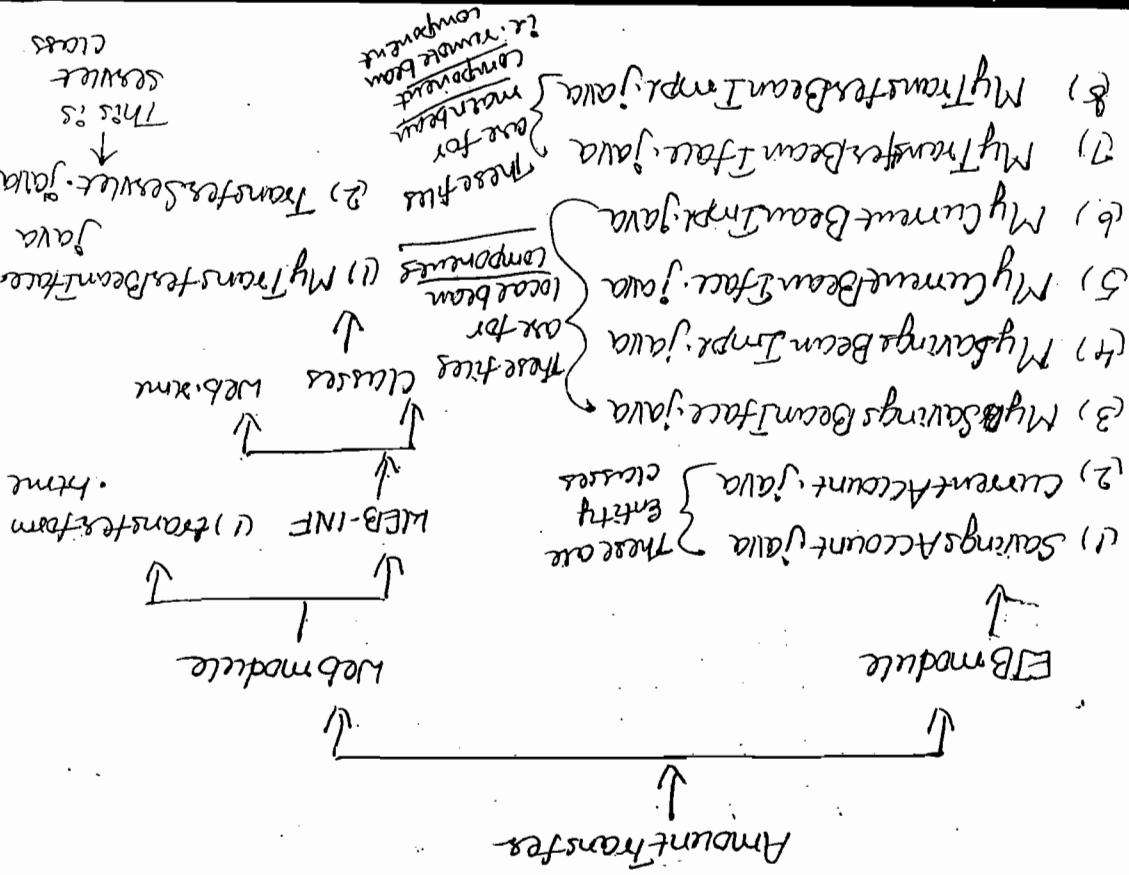
```
ut.rollback();
```

Where rollback is the method of UserTransaction interface obj that will send one rollback signal to the DB so that the DBEngine will cancel the results of the buffer by clearing the buffer without committing those results into the DB table.

→ The following practical ex: demonstrate how to manage the DB transactions in the EJB app'n by injecting UserTransaction interface obj available at the Resource of the app'n server into the bean component, this app'n is the communication b/w the Web app'n & the EJB app'n to transfer some amount from the DB table called Saving Account into another DB table called Current Account

C  
=





Saving Account into another DB table called CurrentAccount  
 to transfer some amount from the DB table called  
 to the communication bin in the web appn of the ETB appn  
 of the appn server into the bean component, this appn  
 uses transaction interface of available at the Resource  
 the DB transactions in the ETB appn by injecting  
 ← The following practical ex: demonstrate how to manage

These results into the DB table.  
 of the buffer by clearing the buffer without committing  
 to the DB so that the DB engine will cancel the results  
 interface of fact will send one roll back signal

What rollback is the method of user transaction

at rollback)

result of the buffer

\* We should use the following java code to control the

10-7-09

## Java Message Service (JMS)

- \* JavaMessageService is the service provided by the app'n server to create, send, receive the msgs asynchronously by the Java app's
- \* Asynchronous means that the receiver does not have to actively request for the msg to receive it.
- \* There are 2 types of msgservices provided by the server ie.
  - (1) point-to-point msg service
  - (2) publish subscribe "

Most of the app'n servers will provide both of this types of msgservices.

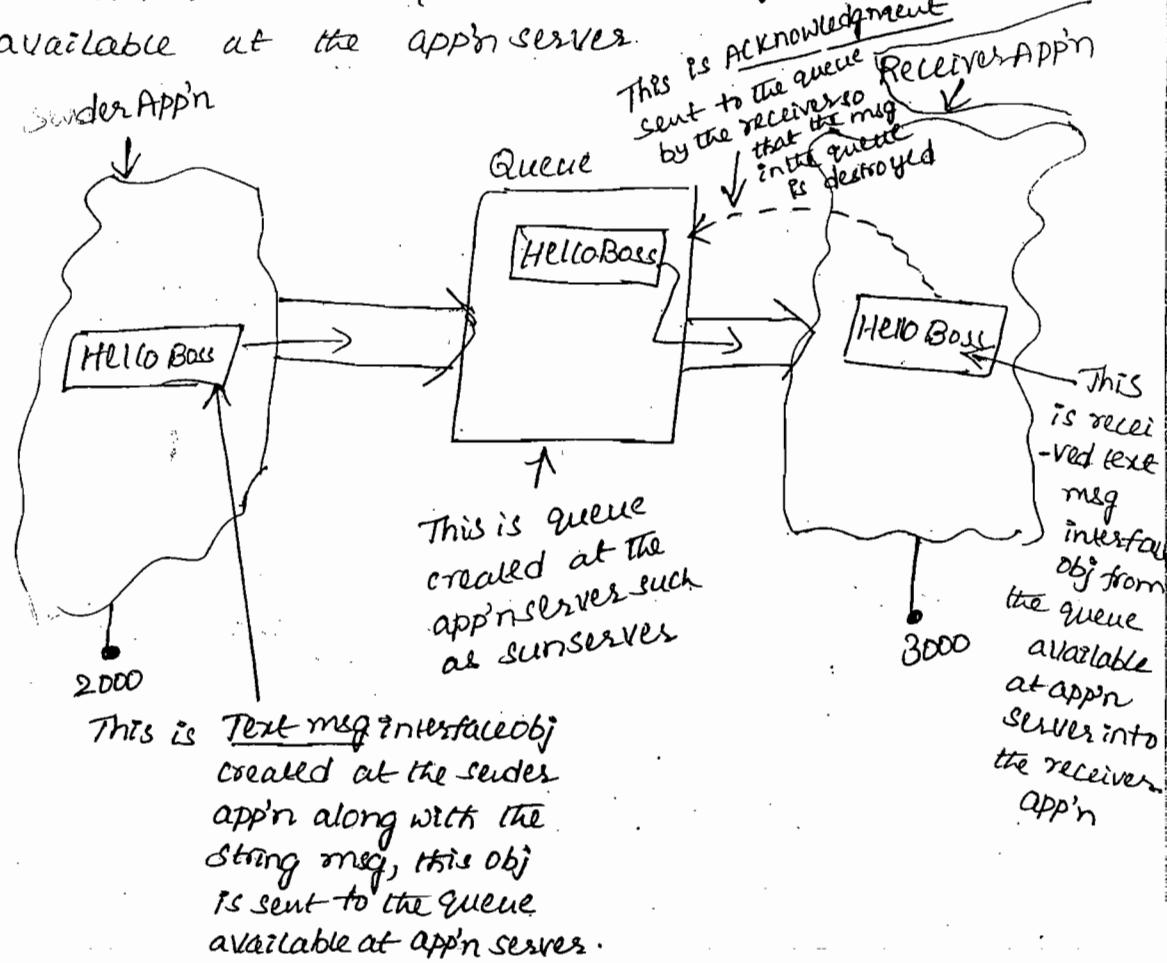
### point-to-point msg service:

- \* This point-to-point msgservice is based on queues, the characteristic of the queue is FIFO(First InFirstOut) ie. the msg leave from the queue in the same order that it is stored.

- \* All the msgs that are sent to the queue by the sender are available at server until they are consumed by the receiver, which means that the messagequeue available at the app'nservr will act as the common point for the msgs b/w the sender & the receiver.

- \* The types of msg obj's that can be sent in this JMS service are text msg, obj msg, Stream msg

→ The following diagram demonstrate this Asynchronous communication b/w the sender & receiver through the queue available at the appn server.



We should remember the following 3 points in this point-to-point msg service by using the queue appn server.

- 1) Each msg. can be consumed by a single consumer which means there can be many consumers i.e. many receivers are connected to the same queue, but only one consumer will get the individual msg.
- 2) The consumer automatically acknowledge the receipt of the msg to the server so that the msg is automatically destroyed from the queue.

more 3) sending & receiving the msgs are not time dependent,

which means the sender can send the msg at any time,  
the receiver can receive the msg at any later time.

Procedure at the sender app'n to send the msgs to the queue available at the server

\* We should follow the following procedure at the sender app'n to send the msgs into the queue which is already created at the app'n server i.e.

(1) We should create one Queue Connection Factory at the app'n server such as sunappnServer with some JNDI name such as myqueuefactory.

(2) We should create one queue at the appnserver with some JNDI name such as myqueue.

(3) We should get the stub obj of Queue Connection Factory into the sender app'n by looking into the JNDI Registry of the appnserver such as

QueueConnectionFactory qcf = (QueueConnectionFactory)  
ic.lookup ("myqueuefactory");

(4) We should create the QueueConnection interface obj such as

QueueConnection qc = qcf.createQueueConnection();

where createQueueConnection is the method of QueueConnectionFactory interface obj, this method automatically create one obj of Queue Connection interface available in javax.jms package.

(5) We should create the QueueSession interface obj such as

QueueSession qs = qc.createQueueSession(false, Session.AUTO\_ACKNOWLEDGE);

12-7-09

(6) We should get the Stub obj of the Queue available at the server by looking into the JNDI registry of the appn server such as

javax.jms.Queue q = (javax.jms.Queue) ic.lookup("myqueue");

(7) We should create the QueueSender interface obj by connecting to the Stub obj. of the Queue such as

QueueSender qsender = qs.createSender(q);

(8) We should establish the connection from the QueueSender interface obj to the Queue available at the server such as

qc.start();

(9) We should create the TextMessage interface obj such as

TextMessage tm = qs.createTextMessage();

(10) We should store the required string message into the TextMessage interface obj such as

tm.setText("Hello, how are you!");

(11) We should send the TextMessage interface obj to the Queue at the server through the QueueSender

interface obj such as

qsender.send(tm);

Note

- \* Now the respective TextMessage interface obj along with the String message is stored into the Queue available at the Server so that the message can be received by the receiver at any time.

NOTE: All the above interfaces are available in javax.jms package

procedure at the receiver to receive the msgs from the Queue

- (\*) We should follow the following procedure at the receiver app'n to receive the msgs from the queue available at the app'n server.

\* All the above 6 steps are same even at the receiver

- (J) We should create the QueueReceiver interface obj by

connecting to the stub obj of the Queue such as

QueueReceiver qreceiver = qs.createReceiver(q);

- (\*) We should attach one MessageListener interface obj to the QueueSender interface obj

qreceiver.setMessageListener(new

Note Where Listener is the class implementing MessageListener interface with the following public void onMessage(Message

This is TextMessage interface obj created at the publisher app'n and the queue created directly at the publisher app'n components at the app'n

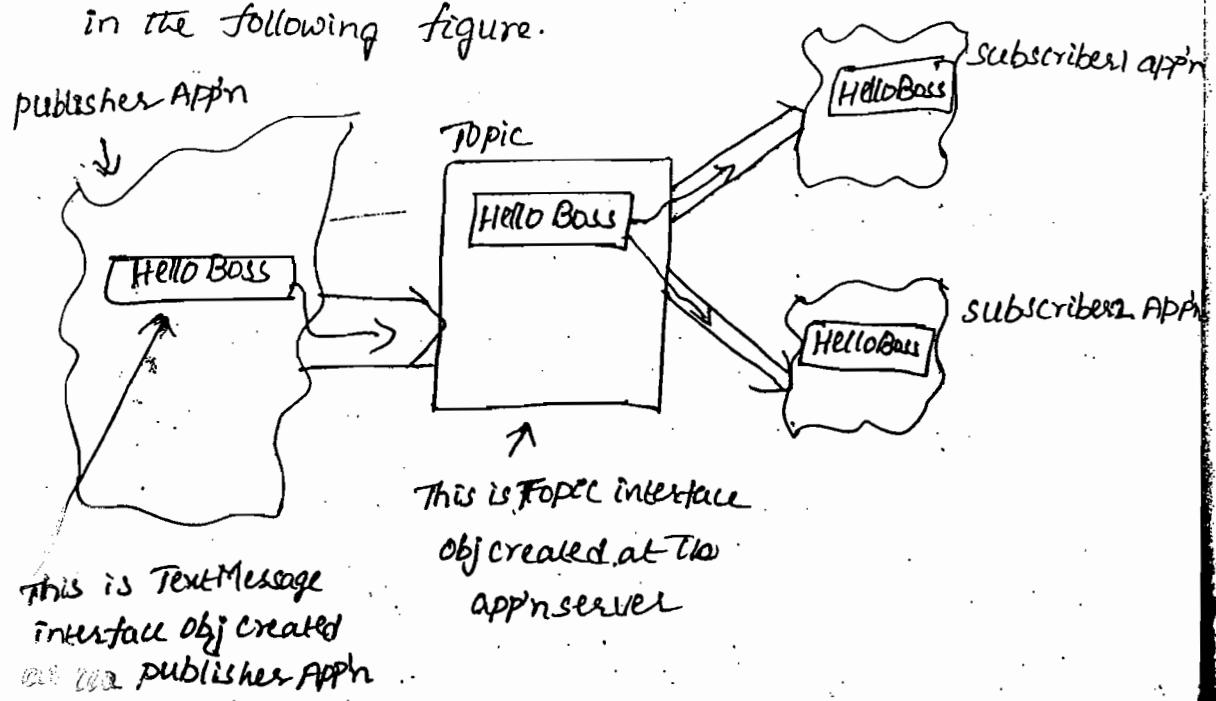
9) We should establish the connection from the QueueReceiver interface obj to the Queue available at the server such as

```
 qc.start();
```

All the messages from the Queue are automatically received into onMessage() of the respective message listener obj.

### Publish subscribe Message service

- \* This publish subscribe Msgservice is used to have the multiple recipients for the same msg, which means that the same msg can be received by the multiple receivers called as subscribers.
- \* The messages are send i.e. published & the msgs are received i.e. subscribed through the topic interface obj which is already created at the app'n server.
- \* These topics are similar to the Queues except that multiple receivers can share the same msg as shown in the following figure.



Note The copy of same TextMessage is received by the both subscribers. Which are connected to the same topic available the app'n server.

\* We should remember the following points in this publisher-subscriber msgservice

(1) Each message can be received by the multiple receivers i.e. Subscribers.

(2) There are two types of subscribers can be connected to the topic available at the server i.e. nondurable subscribers & durable subscribers, where the non-durable "must be actively connected with the topic to get the msgs from the topic, whereas the durable subscribers can be connected to the topic at anytime & get all the msgs from the topic.

Note The procedure to send the msg from the publisher app'n to the topic at the server, the procedure to receive the msg from the topic to subscriber app'n is similar to the above steps in point-to-point msg service except that we should create TopicConnectionFactory at the app'nservice with some JNDI name such as mytopicfactory, we should create one Topic interface obj at the server with some JNDI name such as mytopic.

→ The following practical ex: demonstrate how to send the msgs directly from the sender app'n into the Queue created at the app'nservice & how to receive the msgs directly from the same queue available at the app'nservice into the receiver appn without using any bean components at the appn.

## jmsqueue



- (1) Sender.java
- (2) Receiver.java
- (3) Listener.java
- (4) JndiProperties.java ~~properties~~

### Sender.java

This is the sender java app'n which will directly send the TextMessages to the Queue already created at the SunApp'nServer so that any receiver app'n is expected to receive these msgs at any later point of time from the same Queue available at the SunApp'nServer.

```
import javax.jms.*;
" " . naming.*;
" " . util.*;
" " . io.*;

public class Sender
{
 p.s.v.m(String args[]) throws Exception
 {
 Properties p = new Properties();
 p.load(new FileInputStream("jndiProperties.
 properties"));
 InitialContext ic = new InitialContext(p);
 QueueConnectionFactory qcf = (QueueConnection
 Factory) ic.lookup("myqueue
 factory");
```

```
QueueConnection qc = qcf.createQueueConnection();
QueueSession qs = qc.createQueueSession(false,
 Session.AUTO_ACKNOWLEDGE);
javax.jms.Queue q = (javax.jms.Queue) ic.lookup("myqueue");
```

Where fully qualified ~~or~~ name is specified for the Queue interface bcoz the same Queue interface is also available in java.util package.

```
QueueSender qsender = qs.createSender(q);
qc.start();
TextMessage tm = qs.createTextMessage();
DataInputStream stdin = new DataInputStream(System.in);
while(true)
{
```

```
S.O.P("Enter the msg to the msgQueue");
String msg = stdin.readLine();
tm.setText(msg); // stored inside Textmsg interface obj
qsender.send(tm);
S.O.P("Another msg (yes/no):");
String choice = stdin.readLine();
if(!choice.equalsIgnoreCase("yes"))
 break;
```

```
}
```

```
S.O.P("All ur msgs are sent to the queue of appn
server").
```

```
S.O.P ("Receiver is expected to get these msgs at
any time");
```

```
S.O.P ("Bye Bye");
```

```
System.exit(0);
```

```
}
```

```
}
```

Receiver.java This is the receiver app'n that will receive  
the msgs from the same Queue available at the app'n  
server at any time.

```
import javax.jms.*;
" . naming.*;
" . java.util.*;
" . io.*;
public class Receiver
{
```

```
p.s.v.m (String args[]) throws Exception
```

```
{
```

```
Properties p = new Properties();
```

```
p.load(new FileInputStream("Jndiproperties.
properties"));
```

```
InitialContext ic = new InitialContext(p);
```

```
QueueConnectionFactory qcf = (QueueConnectionFactory)
ic.lookup("myqueuefa
ctory");
```

```
QueueConnection qc = qcf.createQueueConnection();
```

```
QueueSession qs = qc.createQueueSession (false,
Session.AUTO_ACKNOWLEDGE);
```

```
javax.jms.Queue q = (javax.jms.Queue) ic.lookup
("myqueue");
```

```
QueueReceiver qreceiver = qs.createReceiver(q);
```

```
qreceiver.setMessageListener (new Listener());
```

```
qs.start();
```

```
SOP ("Receiver Started");
```

```
}
```

```
}
```

Listener.java, This is the userdefined class by implementing MessageListener interface to receive the msgs automatically from the Queue available at the app'n server, this class obj is attached as MessageListener obj to the QueueReceiver interface obj which is connected to the Queue of the app'n server.

```
import javax.jms.*;
public class Listener implements MessageListener
{
 public void onMessage (Message m)
 {
```

[Hello Boss]

↓  
[How are you]

This is the implemented abstract method of MessageListener interface with the required implementation, this method is automatically executed whenever the msg interface obj is received from the Queue of app'n server.

try

TextMessage tm = (TextMessage)m;

This will type cast the received msg interface obj. into Textmsg interface obj.

String receivedmsg = tm.getText();  
S.O.P (receivedmsg);

}

catch (Exception e)

{

}

}

[Here we no need of jar file bcoz we are not using Bean component]

O/P

> java sender

Enters the msg to the message

Hello boss

Another msg (yes/no): yes

Enters the msg to the message.

How r u?

Another msg (yes/no): No

open another cmd prompt for receiver

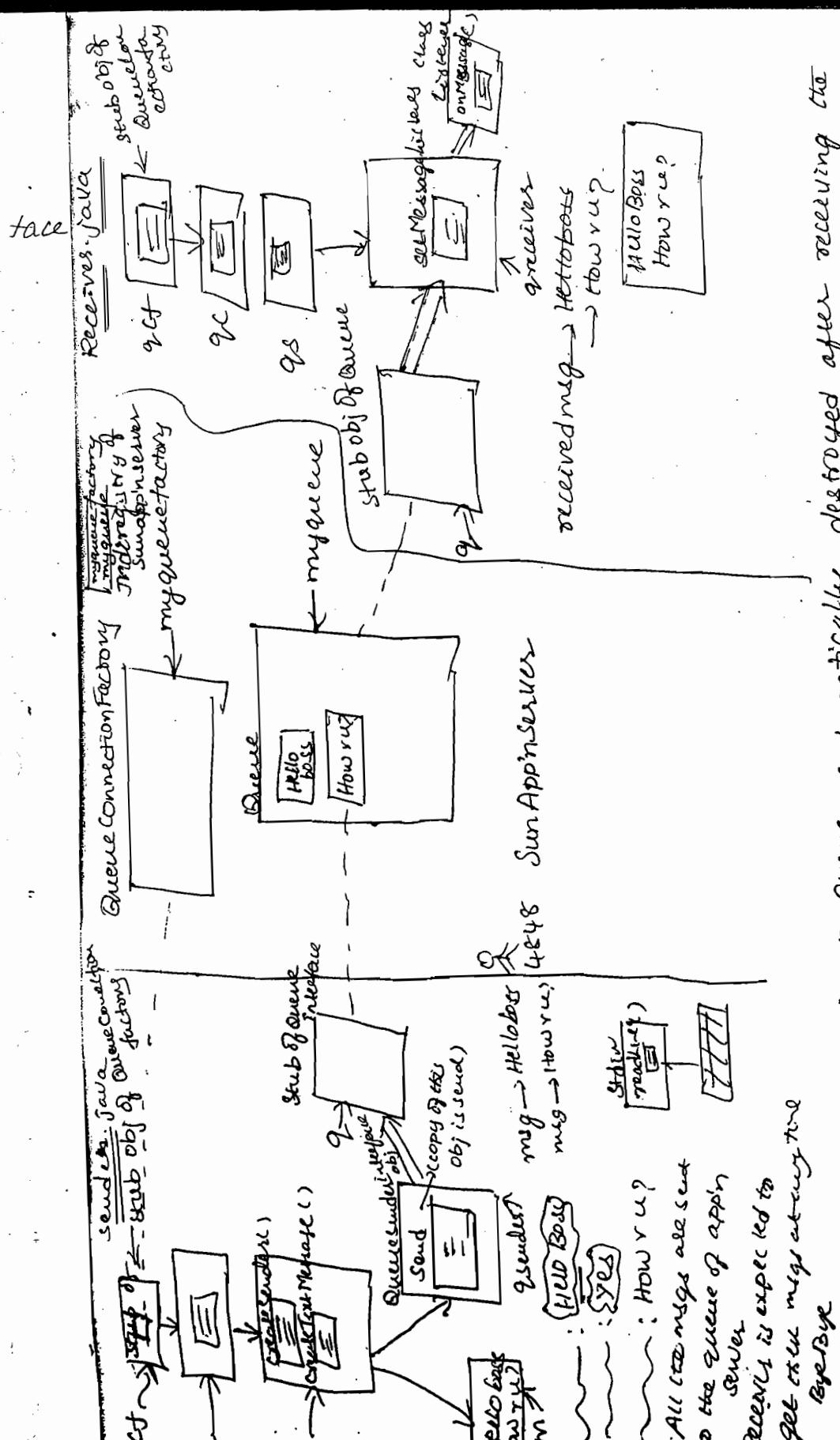
> java receiver

get object of queuefactory  
get object of queueconnection  
get object of queue

QueueConnectionFactory

QueueConnection

get object of queue



ByeBye : main in the Queue automatically destroyed after receiving the  
event in the message queue

13-7-09

The following practical ex: demonstrate publisher, subscriber message service by using TopicConnectionfactory, topic interface obj's created at Sun App'n Server, where publisher App'n will send the msgs to the topic available at the server & the non-durable, durable subscribers will receive those msgs from the same Topic available at the server.

jmsTopic  
↓

- (1) publisher.java
- (2) Non-durable subscriber.java
- (3) Durable " "
- (4) Listener.java
- (5) JndiProperties.properties

### publisher.java

```
import javax.jms.*;
" " . naming.*;
" " . util.*;
" " . io.*;
public class Publisher
{
 public void main (String args[]) throws Exception
 {
 Properties p = new Properties();
 p.load (new FileInputStream ("jndiProperties
 . properties"));
 InitialContext ic = new InitialContext (p);
 TopicConnectionFactory tcf = (TopicConnectionFactory)
 ic.lookup ("mytopicfactory");
```

```
TopicConnection tc = tcf.createTopicConnection();
TopicSession ts = tc.createTopicSession(false,
 Session.AUTO_ACKNOWLEDGE);
Topic t = (Topic) tc.lookup("mytopic");
TopicPublisher tpbr = ts.createPublisher(t);
tc.start();
TextMessage tm = ts.createTextMessage();
DataInputStream stdin = new DataInputStream(System.in);
while (true)
{
 System.out.println("Enter the msg to the topic at the server");
 String msg = stdin.readLine();
 tm.setText(msg);
 tpbr.send(tm);
 System.out.println("Another msg yes/no");
 String choice = stdin.readLine();
 if (!choice.equalsIgnoreCase("yes"))
 break;
}
System.out.println("All ur msgs are sent to the topic of msgserver");
System.out.println("Subscribers are expected to get these msgs");
System.out.println("Bye Bye");
System.exit(0);
```

## ~~14-709~~ NonDurableSubscriber.java

This is NonDurableSubscriber APP'n to receive the msgs from the TOPIC available at the Sun App'n Server, this Non-durable subscriber must be connected to the TOPIC to receive the msgs. Otherwise it can't receive the earlier msgs which are already available in the TOPIC of the server.

```
import javax.jms.*;
 " . naming.*;
 java.util.*;
 " . Po.*;
```

```
public class NonDurableSubscriber
{
```

P.S.V.M (String args[]) throws Exception  
{

```
Properties p = new Properties();
```

```
p.load(new FileInputStream("jndi.properties.properties"));
```

```
InitialContext ic = new InitialContext(p);
```

```
TopicConnectionFactory tcf = (TopicConnectionFactory)ic.lookup
("mytopicfactory");
```

```
TopicConnection tc = tcf.createTopicConnection();
```

```
TopicSession ts = tc.createTopicSession(false, Session.
AUTO_ACKNOWLEDGE);
```

```
Topic t = (TOPIC) tc.lookup("mytopic");
```

```
TopicSubscriber tsbr = ts.createSubscriber(t);
```

```
tsbr.setMessageListener(new Listener());
```

```
tc.start();
```

S.O.P ("Non Durable subscriber is started");

۳

3.

## DurableSubscriber.java

This is DurableSubscriber App'n to receive the msgs from the Topic available at SunApp'nServer, this DurableSubscriber can be connected to the Topic along with some clientId name to the topic at any time so that it can receive all the earlier msgs already available in the topic at the server.

```

import javax.jms.*;
// " . naming.*;
// java.util.*;
// " . PO.*;

public class DurableSubscriber
{
 public void main(String args[]) throws Exception
 {
 Properties p = new Properties();
 p.load(new FileInputStream("jndi.properties.proper-
ties"));
 InitialContext ic = new InitialContext(p);
 TopicConnectionFactory tcf = (TopicConnectionFactory)ic.
 lookup("mytopicfactory");
 TopicConnection tc = tcf.createTopicConnection();
 tc.setClientID("c1");
 }
}

```

Where `setClientID()` is the method of `topicConnection` interface obj that will store the

respective ClientID name i.e. 'c1' into the respective TopicConnection interface obj.

This ClientID name is sent to the server at the time of establishing the connection so that the Sun App'n Server will maintain this ClientID name along with the pointer to topic at the server side so that all the msgs available in the topic are sent to the respective client.

```
TopicSession ts = tc.createTopicSession(false, Session.AUTO_ACKNOWLEDGE);
```

```
Topic t = (TOPIC) ic.lookup("mytopic");
```

```
TopicSubscriber tsbr = ts.createDurableSubscriber(t, "c1")
```

This will create an Obj of TopicSubscriber interface by connecting to the stub Obj of the Topic by storing the respective ClientID name i.e. 'c1' into the respective TopicSubscriber interface obj so that it will collect all the msgs available in the topic against to the respective Client-ID name i.e. 'c1'.

```
tsbr.setMessageListener(new Listener());
```

```
tc.start();
```

```
s.o.p("Durable subscriber is started");
```

```
}
```

```
}
```

### Listener.java

```
import javax.jms.*;
```

```
public class Listener implements MessageListener
```

```
{
```

```

public void onMessage(Message m)
{
 try
 {
 TextMessage tm = (TextMessage)m;
 String receivedmsg = tm.getText();
 S.O.P(receivedmsg);
 }
 catch(Exception e)
 {
 e.printStackTrace();
 }
}

```

practical procedure for the previous app'n called 'jmsqueue'

#### classpath settings

set path = D:\sun\jdk16\bin\;

set classpath = D:\sun\jdk16\lib\tools.jar; D:\sun\APPSever\lib\javaee.jar; D:\sun\APPServer\lib\appserver-rt.jar; D:\sun\APPServer\lib\appserver-admin.jar; D:\sun\APPServer\lib\install\application\jmsra\imqjmsra.jar;;

JAVA\_HOME = D:\sun\jdk16

#### practical procedure

(1) Creation of QueueConnectionFactory at Sun app'n server.

openserver admin console



Click on Resources



Click on JMS resources



Click on connection factories



Click on new



give the required JNDI name such as myqueuefactory



Select the resource type as javax.jms.QueueConnection



Factory

Click on OK

Now QueueConnectionFactory is created successfully  
at the server with the respective JNDI name.

## (2) creation of Queue at Sun app'n server

After clicking the resources



Click on JMS resources



Click on Destination resources



Click on new



give the required JNDI name such as myqueue



give the physical destination name as also same  
name as myqueue



Select the resource type as javax.jms.Queue



for topic app'n, for durable subscriber do the following.

Click on add property



give the name as clientID & the value as C1



click on OK



click on OK

NOW Queue is created successfully at the server  
with the respective JNDI name.

3) compile the java files to generate the classpath files

such as

> java c \*.java

(4) create & first execute the <sup>Receiver</sup> sender app'n such as

> java Receiver [ctrl c is for closing the Receiver]

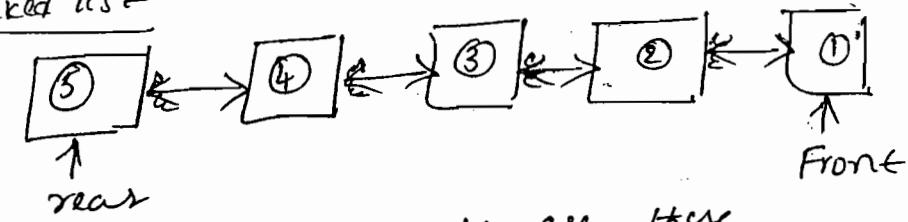
(5) Execute the sender app'n such as

> java sender

Queue - FIFO  
Stack - LIFO

Note one obj internally point to another obj is called as

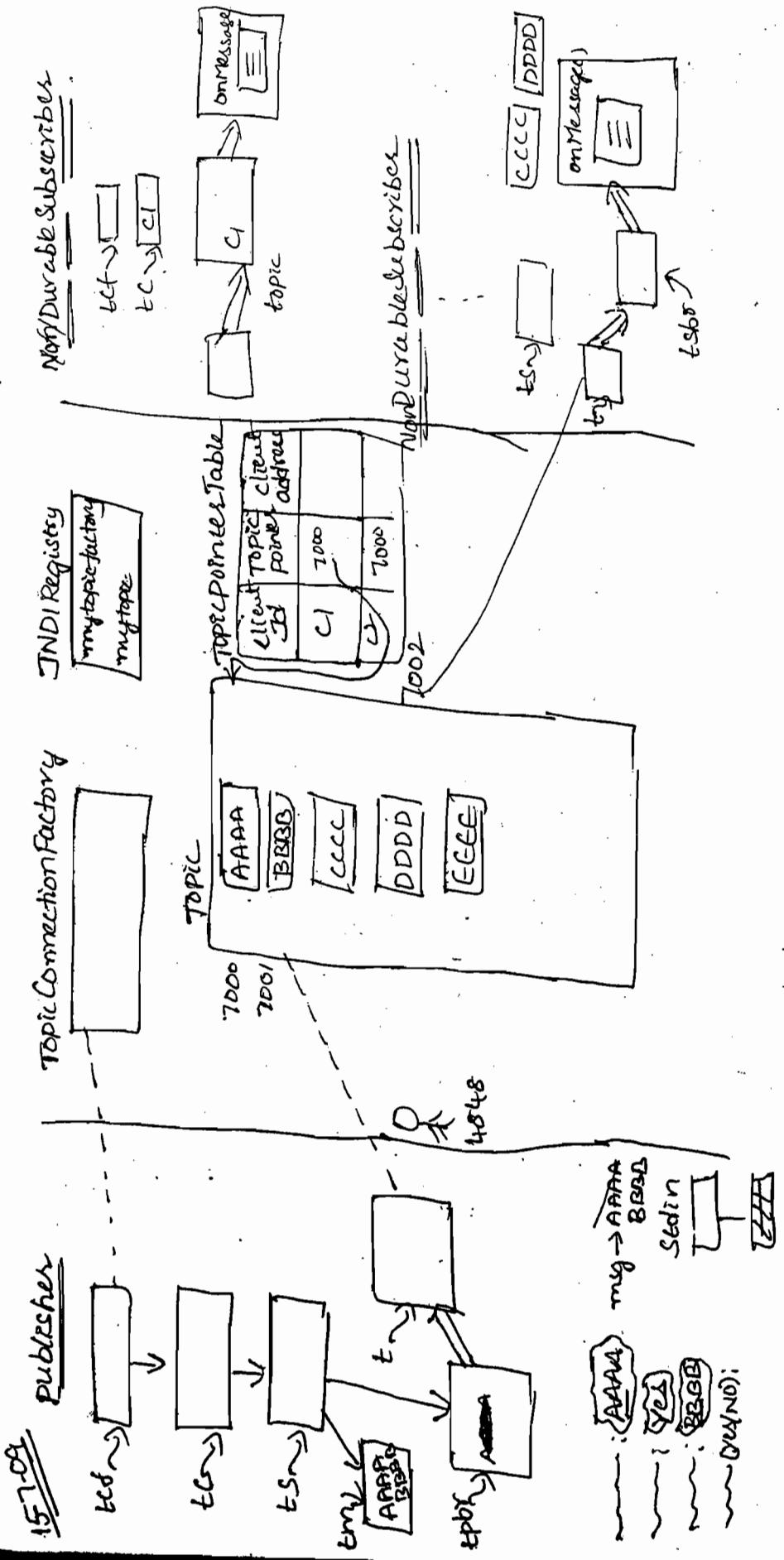
Linked list



We have used 2 points are there

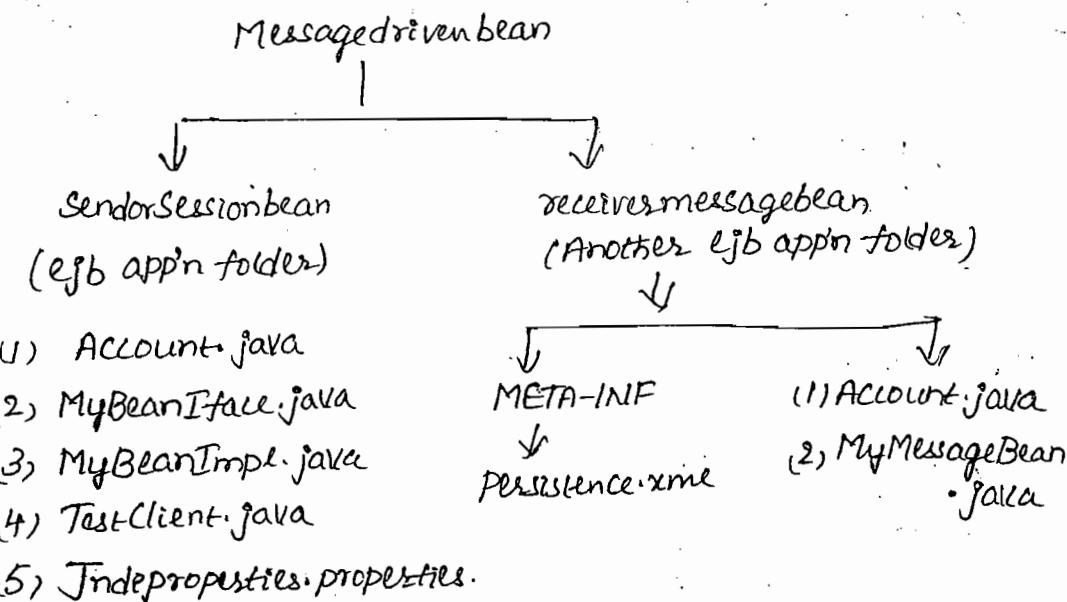
For Front = rear = 1 If Queue is one obj  
if " " = null " " is empty:

No pointer is for Non-durable subscriber  
pointer is only for durable subscriber like C1, C2 etc



5-7-09

The follow



→ The above practical ex: demonstrate how to design the bean components in the EJB app'n to store the obj msg into the queue & to receive the obj msgs from the Queue which is already creating available at the app'n server.  
Hierarchy of the app'n is as above.

### Account.java

```
package entity;
import javax.persistence.*;
import java.io.*;
```

① Entity public class Account implements Serializable

{

```
 @Id public int accno;
 " String accname;
 " float accbal;
```

}

### MyBeanIface.java

```
package entity;
import javax.ejb.*;
```

① Remote public interface MyBeanInterface

{

```
public void insert(int accno, String accname,
float accbal);
```

}

### MyBeanImpl.java

```
package entity;
import javax.ejb.*;
" " .jms.*;
" " .annotation.*;
```

② Stateless public class MyBeanImpl implements  
MyBeanInterface

{

③ @Resource(mappedName = "myqueuefactory")

```
QueueConnectionFactory qcf;
```

This will inject the stub obj of QueueConnection  
Factory which is already created at the app server with  
the JNDI name as myqueuefactory into this bean  
component.

④ @Resource(mappedName = "myqueue") Queue q;

```
QueueConnection qc;
```

```
QueueSession qs;
```

```
QueueSender qsender;
```

```
public void insert(int accno, String accname,
float accbal)
```

{

```
try {
```

{

```
Account acc = new Account();
acc.accno = accno;
acc.accname = accname;
acc.accbal = accbal;
qc = qcf.createQueueConnection();
qs = qc.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
qsender = qs.createSender(q);
qco.start();
ObjectMessage om = qs.createObjectMessage();
om.setObject(acc);
qsender.send(om);
qc.close();
```

```
}
```

```
catch(Exception e)
```

```
{
```

```
}
```

```
}
```

```
}
```

TestClient.java This is the clientside java appn

```
import entity.MyBeanIface;
" javax.naming.*;
" java.util.*;
" " .io.*;
```

```
public class TestClient
```

```
{
```

p.s.v.m (String args[]) throws Exception

```
{
```

```

Properties p = new Properties();
p.load(new FileInputStream ("jndiproperties.properties"));
InitialContext ic = new InitialContext (p);
MyBeanIface mbi = (MyBeanIface) ic.lookup(
 "entity.mybeanIface");
DataInputStream stdin = new DataInputStream (System.in);
while (true)
{
 System.out.print ("Enter acc no:");
 int accno = Integer.parseInt (stdin.readLine ());
 System.out.print ("Enter acc name:");
 String accname = stdin.readLine ();
 System.out.print ("Enter acc bal:");
 float accbal = Float.parseFloat (stdin.readLine ());
 mbi.insert (accno, accname, accbal); [This will call Business
 logic method]
 System.out.print ("Insert another record yes/no");
 String choice = stdin.readLine ();
 if (!choice.equalsIgnoreCase ("yes"))
 break;
}
}

```

MyMessageBean.java This is the message driven bean class

defined by the user in this EJB appn by implementing  
MessageListener interface method i.e. onMessage()  
 so that this bean component automatically receive the obj

msgs which are available in the queue & they are automatically inserted into the DB-table called account, this bean class obj is automatically created at the time of deploy ing the jar file of this EJB appn.

```
package entity;
import javax.persistence.*;
" " . ejb.*;
" " . jms.*;
" " . annotation.*;
```

### ② MessageDriven/activation Config = {

④ ActivationConfigProperty (propertyName = "destinationType",  
propertyValue = "javax.jms.Queue") }, mappedName  
= "myqueue")

public class MyMessageBean implements MessageListener  
{

④ PersistenceContext EntityManager em;

public void onMessage(Message m)

{

try

{

ObjectMessage om = (ObjectMessage) m;

Account acc = (Account) om.getObject();

em.persist(acc);

s.o.p ("Record inserted successfully in the DB table");

}

catch (Exception e)

{

}

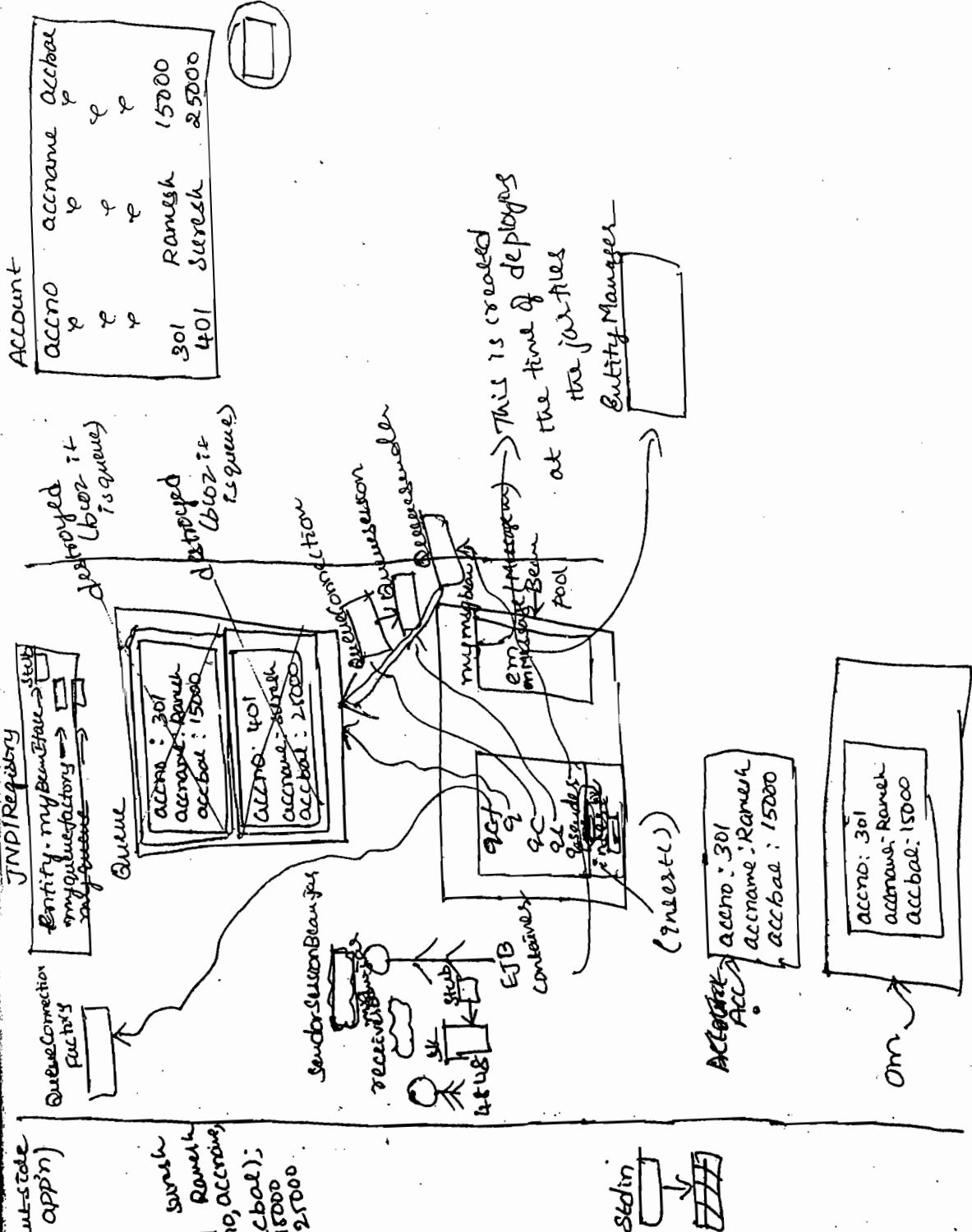
Note The above metadata annotation called `@MessageDriven` is used to provide the info to the EJB container that this bean component is MessageDriven components so that the container automatically create this bean class obj in the bean pool at the time of deploying the jar file of this EJB app'n which means the client need not call any business logic methods of this bean component.

Where the above metadata annotation called

`@ActivationConfigProperty` is used provide the info to the EJB container about the JNDI name of the Queue which is already created & available at \$ App'n Server along with the type of the Queue so that this bean component automatically get all the obj message which are available in the Queue with the JNDI name as myqueue.

> java -d . \*.java [entity] [inside this package]  
> jar -cvf senderSessionbean.jar [if the package name is there we can use -d otherwise need no]  
> java -d . \*.java  
> jar -cvf receiverMessagebean entity META-INF  
> deployment [can create a table] [receiver messagebean.jar]  
> javac TestClient.

mappedName is used as we can look into the JNDI registry.



- ~: 301
- ~: Rameh
- ~: 15000
- ~(yes/no): (y/n)
- ~: 401
- ~: Sarah
- ~: 25000
- ~(yes/no): (y/n)

17-7-04

## JAVA TIMERS

- \* One of the service provided by the app'n server for the ejb components is called TimerService
- \* We can use this timer service in the bean classes of the ejb app'n alk to app'n req's.
- \* If we want to use this TimerService in the bean classes then we should inject the TimerService interface obj. available at the resource of the app'n server into the bean component by using the following metadata annotation i.e.

@Resource TimerService ts;

- \* We should create the Timer interface obj in the business logic methods of the bean component by setting the required time in number of milliseconds such as

No Incomplete

```
javax.ejb.Timer t = ts.createTimer(30000, null);
```

- \* Where createTimer is the method of TimerService interface obj that will automatically create one obj of Timer interface available in javax.ejb package by setting the timer obj with the specified time.

- \* We can pass any Serializable obj to the Timer interface obj in place of null value alk to app'n req's.

- \* We can perform any unspecified func in the bean class after expiry of the time which is set in the

Timer interface Obj such as

@TimeOut public void addBonus(javax.ejb.Timer t)  
{}

Some required implementation

}

\* The above userdefined method called addBonus is automatically executed by the EJB container along with the Timer interface obj after expiry of 30 seconds of time that we have set in the Timer interface obj.

\* We can also set the Timer interface obj with the repetative time such as

javax.ejb.Timer t = ts.createTimer(40000, 20000, null);

Where 40000 is initial time, 20000 is the repetitive time i.e. snooze time.

This will set the initial time in the Timer obj with 40 seconds, After expiry of this time it is automatically set the timer with 20 seconds

\* If we want to stop this repetitive timer then we should use the following java code such as

Collection<javax.ejb.Timer> ctr = ts.getTimers();

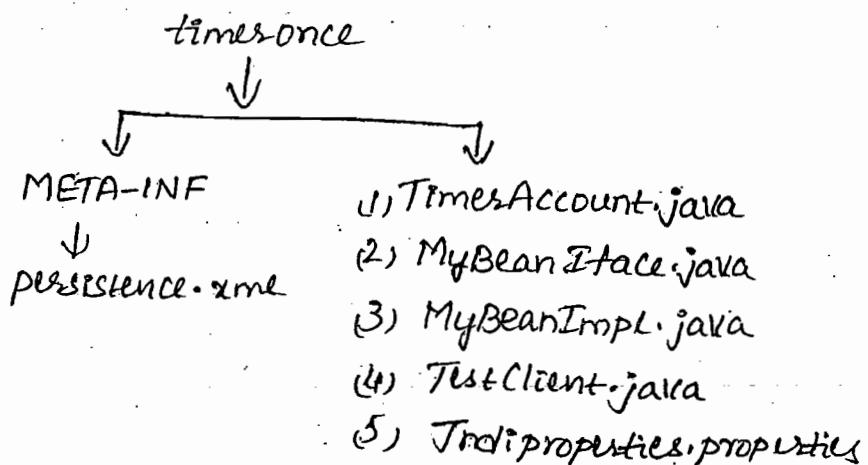
Where getTimers is the method of Timer service interface obj. that will return the reference of all the Timer obj's i.e. Collection of Timer obj's which are already created in the respective TimerService interface obj.

for (javax.ejb.Timer t:ctr)  
{}

t.cancel();

Where cancel is the method of Timer interface obj that will cancel the respective timer obj which means the respective timer obj is destroyed.

→ The following practical ex: demonstrate how to use the TimerService available at the sun appn server in the bean components of the EJB appn so that some of the userdefined Business Logic methods of the bean component are automatically executed by the EJB container at the specific duration of time intervals.



### TimesAccount.java

```
Package entity;
import java.io.*;
" javax.persistence.*;
```

① Entity public class TimesAccount implements Serializable

{

② Id public int tacno;  
" String tacctype;

public String tacname;  
" float tacbal;  
}  
" Obj  
ane

### MyBeanIface.java

package entity;  
import javax.ejb.\*;  
" java.util.\*;

④ Remote public interface MyBeanIface  
{

public void insert(int tacno, String tacctype,  
String tacname, float tacbal);

public void startTimer(int time);

}

### MyBeanImpl.java

package entity;  
import javax.ejb.\*;  
" javax.persistence.\*;  
" " . annotation.\*;  
" java.util.\*;

④ Stateless public class MyBeanImpl implements  
MyBeanIface  
{

④ PersistenceContext EntityManager em;

④ Resource TimerService ts;

public void insert(int tacno, String tacctype,  
String tacname, float tacbal)

S

Te

```
TimerAccount tacc = new TimerAccount();
```

```
tacc.taccno = taccno;
tacc.tacctype = tacctype;
tacc.taccname = taccname;
tacc.taccbal = taccbal;
em.persist(tacc);
```

```
}
```

```
public void startTimer(int time)
{
```

```
 javax.ejb.Timer t = ts.createTimer(time, null);
```

```
}
```

```
@Timeout public void addBonus(javax.ejb.Timer t)
```

```
{
```

```
Query q = em.createQuery("select tacc from
TimerAccount tacc where tacc.tacctype
= 'savings'");
```

```
Collection<TimerAccount> ctaccr = q.getResultList();
```

```
for(TimerAccount tacc: ctaccr)
```

```
{
```

```
 tacc.taccbal = tacc.taccbal + 1000;
```

This changed value i.e. updated value is automatically reflected into the corresponding DBtable called TimerAccount.

```
}
```

```
}
```

```
}
```

## TestClient.java

```
import entity.MyBeanInterface;
 " " TimerAccount;
 " javax.naming.*;
 " java.util.*;
 " " IO.*;
public class TestClient
{
 p.s.v.m (String args[])throws Exception
 {
 Properties p = new Properties();
 p.load(new FileInputStream("jndi.properties",
 properties));
 InitialContext ic = new InitialContext(p);
 MyBeanInterface mbi = (MyBeanInterface) ic.lookup(
 "entity.MyBeanInterface");
 }
 DataInputStream stdin = new DataInputStream
 (System.in);
 while(true)
 {
 S.O.P("do u want to insert the record (yes/no)");
 String choice = stdin.readLine();
 if(!choice.equalsIgnoreCase("yes"))
 break;
 S.O.P("Enter ac no");
 int tacno = Integer.parseInt(stdin.readLine());
 S.O.P("Enter ac type (savings/current)");
 }
}
```

```
String tacctype = stdin.readLine();
S.O.P ("Enter a/c name");
String taccname = stdin.readLine();
S.O.P ("Enter a/c bal");
float tacbal = Float.parseFloat(stdin.readLine());
mbi.insert (taccno, tacctype, taccname, tacbal);
S.O.P ("Record inserted successfully");
```

{

while (true)

{

S.O.P ("do u want to create the timeset server (yes/no)");

String choice = stdin.readLine();
if (!choice.equalsIgnoreCase ("yes"))
 break;

S.O.P ("Enter the time to set in the timer (in milli
seconds");

int time = Integer.parseInt(stdin.readLine());

mbi.startTimer (time);

S.O.P ("timer started");

{

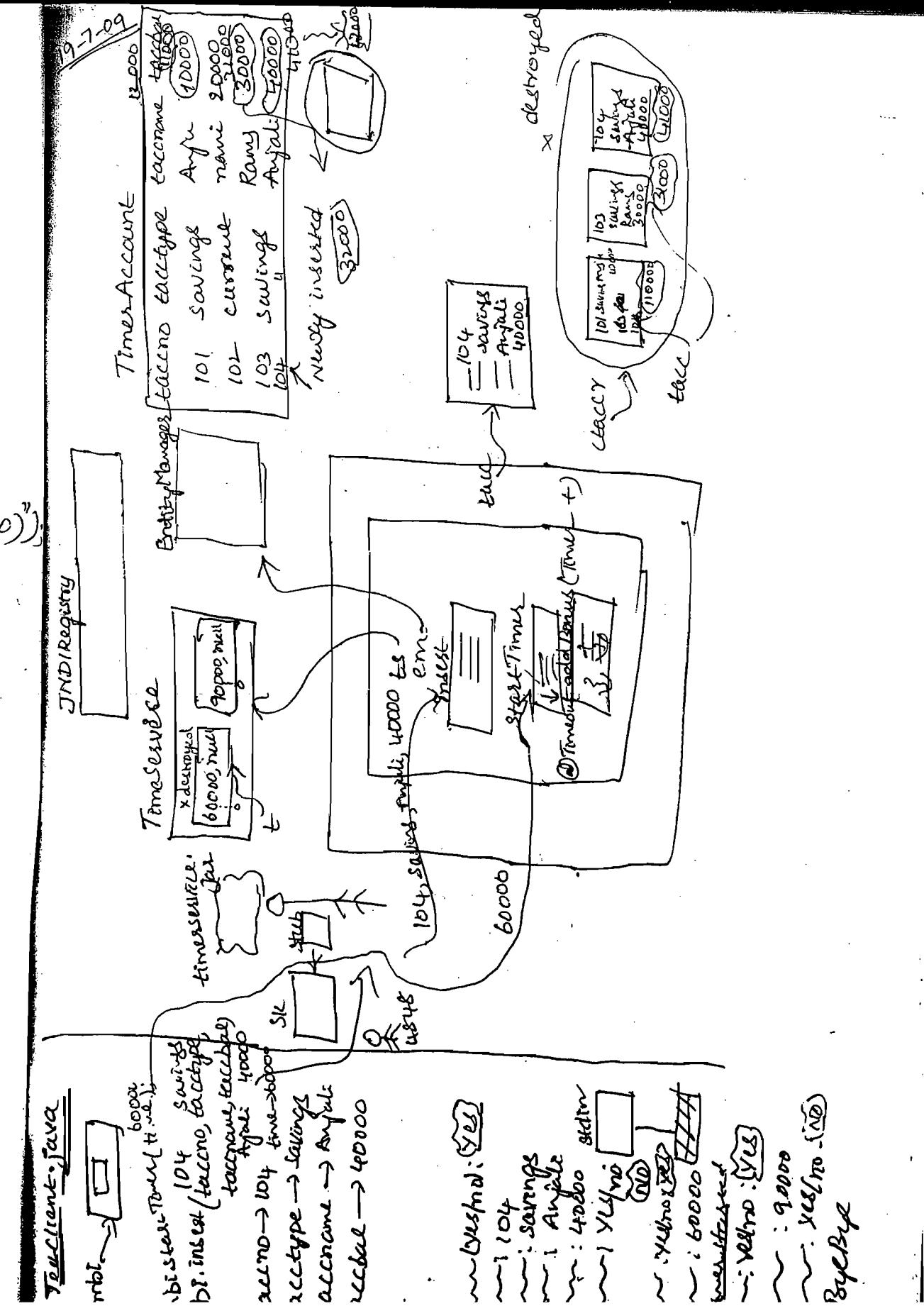
S.O.P ("Bye Bye");

{

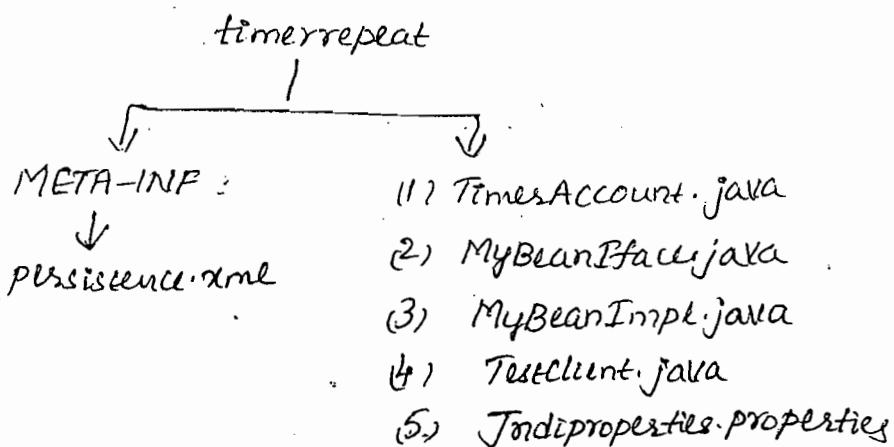
{

JNDI Registry

Account.java



→ The following practical ex: demonstrate how to create the Timer by setting the repetitive time i.e. snoozetime at the app'n server.



### TimerAccount.java

```
package entity;
import javax.persistence.*;
" java.io.*;
```

② Entity public class TimerAccount implements Serializable

{

③ Id public int facno;  
public String tacctype;  
" " tacccname;  
" float tacbal;

}

### MyBeanIface.java

```
package entity;
import javax.ejb.*;
" java.util.*;
```

④ Remote public interface MyBeanIface

```
{
 public void insert (int tacno, String tacctype,
 String taccname, float tacbal);
 public void setBonusAmount (float amt);
 // " StartRepetitiveTimer (int initTime,
 // int repeatTime);
 // " stop " " ();
}
```

### MyBeanImpl.java

```
package entity;
import javax.ejb.*;
// " persistence.*;
// " annotation.*;
// java.util.*;
```

② Stateless public class MyBeanImpl implements

MyBeanIface

③ PersistenceContext EntityManager em;

④ Resource TimerService ts;

float bonusamt;

public void insert (int tacno, String tacctype,  
 String taccname, float tacbal)

{

TimerAccount tacc = new TimerAccount();

tacc.tacno = tacno;

tacc.tacctype = tacctype;

tacc.tacname = tacname;

```
tacc.taccbal = taccbal;
em.persist(tacc);
}

public void setBonusAmount(float amt)
{
 Bonusamt = amt;
}

public void startRepetativeTimer(int initime,
 int repeatime)
{
 javax.ejb.Timer t = ts.createTimer(initime,
 repeatime, null);
}

public void stopRepetativeTimes()
{
 Collection<javax.ejb.Timer> ctr = ts.getTimers();
 for(javax.ejb.Timer t:ctr)
 {
 t.cancel();
 }
}

① Timeout public void addBonus(javax.ejb.Timer t)
{
 Query q = em.createQuery("select tacc from
 TimeAccount tacc where tacc.taccType
 = 'savings'");
```

```
Collection<TimerAccount> ctaccr = q.getResults();
for(TimerAccount tacc: ctaccr)
{
 tacc.taccbal = tacc.taccbal + bonusamt;
}
}
```

### TestClient.java

```
import entity.MyBeanInterface;
 " . TimerAccount;
 " javax . naming.*;
 " java . util.*;
 " " . io.*;

public class TestClient
{
 p.s.v.m (String args[]) throws Exception
 {
 Properties p = new Properties();
 p.load(new FileInputStream ("jndiproperties.
 properties"));
 InitialContext ic = new InitialContext (p);
 MyBeanInterface mbi = (MyBeanInterface) ic.lookup
 ("Entity.MyBeanInterface");
 DataInputStream stdin = new DataInputStream
 (System.in);
```

while (true)

{

S.O.P ("Do you want to insert the record (yes/no)");

String choice = stdin.readLine();

if (!choice.equalsIgnoreCase("yes"))

break;

S.O.P ("Enter a/c no.");

int taccno = Integer.parseInt(stdin.readLine());

S.O.P ("Enter a/c type");

String tacctype = stdin.readLine();

S.O.P ("Enter a/c name");

String tacctname = stdin.readLine();

S.O.P ("Enter a/c bal.");

float taccbal = Float.parseFloat(stdin.readLine());

mbi.insert(taccno, tacctype, tacctname, taccbal);

S.O.P ("Record inserted successfully");

}

S.O.P ("Enter the bonus amt to be added in for the savings a/c holders");

float amt = Float.parseFloat(stdin.readLine());

mbi.setBonusAmount(amt);

S.O.P ("Enter the initial time to set in the timer");

int inittime = Integer.parseInt(stdin.readLine());

S.O.P ("Enter the repetitive time to set in the timer");

int repeattime = Integer.parseInt(stdin.readLine());

mbi.startRepetativeTimer(inittime, repeattime);

```

S.O.P ("Repetative timer started");
while(true)
{
 S.O.P ("Do u want to stop the timer (yes/no)");
 String choice = stdin.readLine();
 if(choice.equalsIgnoreCase("yes"))
 {
 mbi.StopRepetativeTimer();
 break;
 }
}
S.O.P ("Bye Bye");
}
}

```

O/P do u want insert the record: yes  
 Enter the a/c no: 107  
 " " a/c type: Anjalik Savings  
 " " " name: Anjali  
 " " " bal: 60000  
 Do u want to insert the record: no  
 Enter the bonus amt to added 30000  
 " " Initial timer : 60000(m-s)  
 Repetative " : 20000(m-s)  
 Do u want to stop the timer : 20000(m-s) (yes/no): yes  
 Bye Bye.

## Final Registry



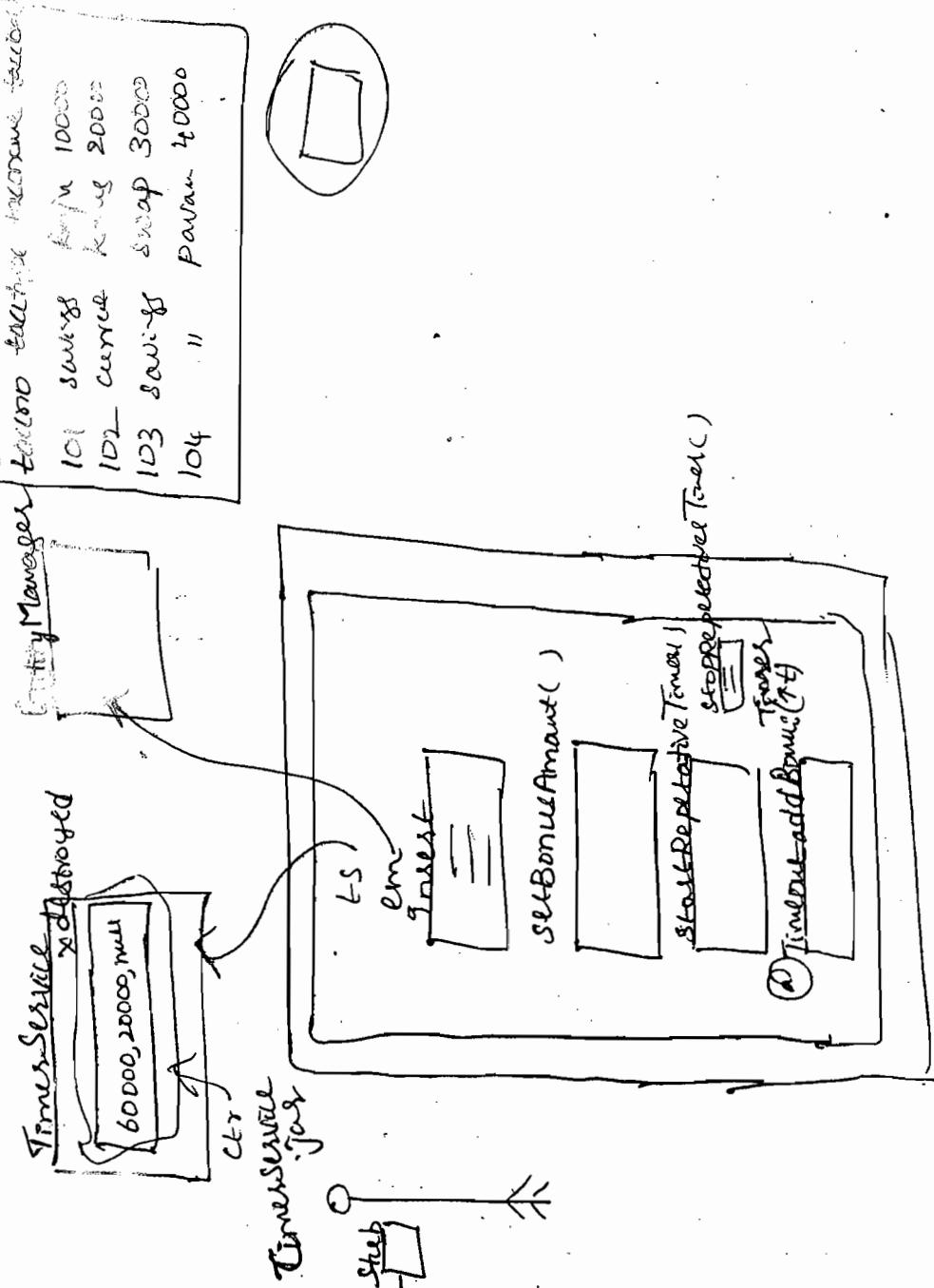
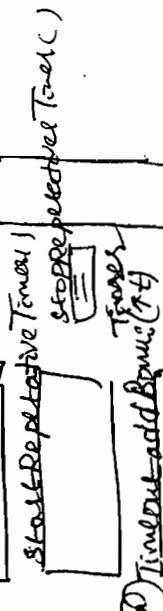
Initial Value

amt → 3000  
initTime → 20000  
Repetitive → 20000

InterestRepetitive  
(60000, 20000)

amt: 3000  
: 60000  
: 20000  
:(y/n): yes

4848



20-7-09

## → Java Authentication & Authorization services

### \* Authentication

Authentication is the procedure or mechanism used in several side ejb app'n to check for the validation of the required client.

- \* Sometimes we don't want to give access permission to the EJB app'n for every required client.
- \* When the client make a request to the serverside ejb app'n along with some username & password then we should check for the validation of the client with the avail at serverside so that if the client is valid then we give the access permission to the app'n otherwise we will reject the client request.
- \* We should store a list of valid user names & their valid passwords at serverside. so that the ejb container automatically check this authentication whenever the request is received from the client.
- \* If the client is valid client then the client address details username, password are automatically stored into the SessionContext interface obj & the permission is given to access the app'n otherwise the container automatically send an exception to the requested client.
- \* The EJB container will provide this authentication & authorization services so that we can use these services in the EJB component of the EJB app'n acc to app'n requirements.

## Authorization

Authorization is the procedure & mechanism is used to check whether a particular user can access to a particular resource or business logic method available in the bean component.

- \* This authorization will come into the picture in the EJB app'n after the user authentication.
- \* Sometimes we don't want to give the access permission to the business logic method of the ejb component to all authenticated clients, in such case we should use this authorization service in the bean component of the app'n.
- \* This authorization required code can be implemented in the bean class by the user at server side or this can be done automatically by the EJB container by defining the required roles at the EJB container of the app'n server.
- \* If we want to implement this authorization in the bean class then we should use the following code in the bean implementation class i.e.

(a) Resource Session Context sc;

```
public int sub(int f1, int f2)
```

```
{
```

```
Principal p = sc.getCallerPrincipal();
```

```
String uname = p.getName();
```

```
if (uname.equalsIgnoreCase("Ramesh") || uname.
equalsIgnoreCase("suresh"))
```

```
{}
```

```
 return fno-sno; (-'minus)
}
throw new EJBException ("User is invalid user to access
this sub business logic method");
}
```

Where `getCallerPrincipal` is the method of `SessionContext` interface obj that will return an obj of `Principal` interface available in `java.security` package along with the requested client user name available in `SessionContext` interface obj against to the client address details. i.e. Client hostname, Client port no:

\* Where `getName` is the method of `Principal` interface obj that will return the client user name available in the obj as a string.

Jaas

→ The following practical ex: demonstrate how to use the authentication procedure to check for the validation of the client to provide the access permission to the server side of the EJB app'n & how to use this authorization procedure in the Business logic methods of the Bean component to give the access permission to the Business logic method for a set of authenticated clients.

jaasone  
↓

1. MyBeanIface.java
2. MyBeanImpl.java
3. TestClient.java

## MyBeanIface.java

```
package jaas;
import javax.ejb.*;
@Remote public interface MyBeanIface
{
 public int add(int fno, int sno);
 public int sub(int fno, int sno);
 public int mul(int fno, int sno);
}
```

## MyBeanImple.java

```
package jaas;
import javax.ejb.*;
// java.security.*;
// javax.annotation.*;
@Stateless(mappedName = "beansecurity") public class
 MyBeanImple implements MyBeanIface
{
```

@Resource SessionContext sc;

public int add(int fno, int sno)

{

return fno+sno;

}

The user at serverside is not return any authorization code in the above add business logic method, so the EJB container will give the permission to all authenticated clients to access this add business logic method which means

all authenticated clients are the authorized clients to access this add business logic method.

```
public int sub(int fno, int sno)
```

```
{ Principal p = sc.getCallerPrincipal(); }
```

```
String uname = p.getName();
```

```
If (uname.equalsIgnoreCase("Rameesh")) || uname.equalsIgnoreCase("surest"))
```

```
{
```

```
return fno - sno;
```

```
}
```

```
throw new EJBException("Sorry, u are not authorized
user in to access this sub business logic method");
```

```
}
```

The user at serverside is return the authorization code in the above sub business logic method to give the access permission to this business " " for two authenticate-d clients ie. Rameesh, surest

```
public int mul(int fno, int sno)
```

```
{
```

```
Principal p = sc.getCallerPrincipal();
```

```
String uname = p.getName();
```

```
If (uname.equalsIgnoreCase("Dinesh"))
```

```
{
```

```
return fno * sno;
```

```
}
```

```
throw new EJBException("Sorry u are not authorized
```

```
user in to access ... business method");
```

```
2
```

```
11.
```

The user at serverside is return authorization code  
in the above mentioned business logic method to give the access  
permission to only one authenticated client i.e. direct

When we use Weblogic 10 App's services to deploy the jar  
file of this EJB app'n then the Stub obj of the bean compo-  
ment is automatically binded into JNDI registry along  
with the specified JNDI name # & the Remote interface  
name, which means this bean component is binded into the  
JNDI registry with the following JNDI name ie.

"beansecurity #<sup>java:comp</sup>MyBeanInterface"

### TestClient.java

```
import java.MyBeanInterface;
import javax.naming.*;
import java.util.*;
import java.io.*;

public class TestClient
{
 public static void main(String args[])
 {
 String uname = null;
 String upass = null;
 try
 {
 DataInputStream stdIn = new DataInputStream
 (System.in);
 System.out.println("Enter username");
 s.o.p ("lnln Enter username");
 }
 }
}
```

```
uname = stdin.readLine();
uname = uname.toUpperCase();
```

Where `toUpperCase` is the method of `String` class that will convert all the characters of the respective username into uppercase character & return as a string.

```
s.o.p ("lnln Enter user pwd");
```

```
upass = stdin.readLine();
```

```
Hashtable ht = new Hashtable();
```

This will create an obj of `Hashtable` class available in `java.util` package along with 2cols ie. `hashkey`, `hashvalue`.

```
ht.put (Context.INITIAL-CONTEXT-FACTORY, "weblogic.jndi.
WLInitialContextFactory");
```

Where `put` is the method of `Hashtable` class that will store the respective property name, the respective web logic app'n server property value into the Hashkey, Hashvalue <sup>column</sup> of the respective hash table.

```
ht.put (Context.PROVIDER-URL, "t3://localhost:7001");
```

```
ht.put (Context.SECURITY-PRINCIPAL, uname);
```

```
ht.put (Context.SECURITY-CREDENTIALS, upass);
```

```
InitialContext ic = new InitialContext(ht);
```

This will create an obj of `InitialContext` class along with the respective hashtable obj so that connection is established from the client to the web logic app'n server JNDI registry along with the respective username & password values available in the respective hashtable so that the EJB container at the web logic app'n server will check for the validation of the client i.e. authentication of the client.

- \* If the client is not a valid client then the server will automatically send an exception to this requested client.
- \* We are not created JNDI properties file at the client side bcoz we want to send the client username & client pwd dynamically to the weblogic app'n server by creating hash table class obj

```

s.o.p ("In\n u are successfully authtenticated");
MyBeanInterface mbi = (MyBeanInterface) ic.lookup("beansecu-
 -rity #jboss.MyBeanInterface");
s.o.p ("In\n Enter first no:");
int fno = Integer.parseInt(stdin.readLine());
s.o.p ("In\n Enter second no:");
int sno = Integer.parseInt(stdin.readLine());
int addresult = mbi.add(fno, sno);
s.o.p ("The result of addition is "+addresult);
try
{
 int subresult = mbi.sub(fno, sno);
 s.o.p ("In\n The Result of subtraction is "+subresult);
}
catch(Exception e)
{
 s.o.p ("In\n");
 s.o.p (e.getMessage());
}
try
{

```

```

 int mulresult = mbi.mul(fno,sno);
 s.o.p("In\nThe result of multiplication is "+mulresult);
}
catch(Exception e)
{
 s.o.p("In");
 s.o.p(e.getMessage());
}

try
{
 //try (outer block)
 catch (Exception e)
 {
 s.o.p("In authentication failed\nfor the username"
 +uname);
 s.o.p("Bye Bye");
 }
}

```

21-7-09  
practical procedure for the above app'n called jackson by  
using weblogic 10 app'n server.

### class path settings

set path = D:\sun\jdk16\bin\;

set classpath = D:\bea\10\wlservr-10.3\servers\lib\weblogic.jar\;

JAVA\_HOME = D:\sun\jdk16

## practical procedure

(1) compile the java files to generate the class files with the package & generate the jar file for the ejb appn

2) Creation of weblogic 10 server domain

Start



programs



oracle weblogic



oracle weblogic server 10g R3



tools



Configuration Wizard



Click on next



again Click on next



Give the username as Weblogic & password also as Weblogic



click on next, next, next



Give the required domain name such as maidu



Click on create



Click on done

(6)

Now domain created successfully with the specified name.

(3) Start the weblogic10 app'n server domain

Start



programs → oracle weblogic → user proj's → main  
(domain name) → start admin server for weblogic  
server domain

Now the server domain is running at localhost

7001 port no:

(4) open the server console

→ open the browser with the following URL  
<http://localhost:7001/console>

→ give the username as weblogic & password also  
as weblogic

→ click on login

Now console opened successfully

(5) create the a list of authenticated users at the server

→ click on security realms on server console

→ " " myrealm (don't click on new)

→ " " users & groups

→ " " new

→ Give the required username such as ramesh &  
password as ramesh123

→ click on OK

Now the new authenticated user is added. We can  
add any no. of required users with the same procedure.

(6) Deploy the jar file of the EJB app'n.

→ click on deployments

→ " " install

→ " " upload the files

→ " " browse & select the jar file of the ejb  
app'n.

Now the jarfile of the ejb app'n deployed successfully

7) Execute the client app'n & enjoy the o/p

→ The following practical ex: demonstrate how to perform authentication & authorization in the EJB app'n by providing its permission to creating the roles at the Web logic app'n server to access the business logic methods for the respective users included in the respective roles of the EJB app'n.

jaastwo

↓

- (1) MyBeanIface.java
- (2) MyBeanImpl.java
- (3) TestClient.java

Note

Where MyBeanIface.java, TestClient.java are exactly the same app'n i.e. same java files of the previous app'n called jaaseone.

MyBeanIface Impl.java

```
package jaas;
import javax.ejb.*;
" java.security.*;
" " annotation.*;
```

② Stateless(mappedName = "beansecurity")

```
public class MyBeanImpl implements MyBeanIface
{
```

③ Resource SessionContext sc;

```
public int add(int fno, int sno)
```

{

```
 return fno+sno;
```

}

```
public int sub(int fno, int sno)
```

{

```
 if(sc.isCallerInRole("subrole"))
```

{

Where `isCallerInRole` is the method of `SessionContext` interface obj that will check whether the requested client username which is available in `SessionContext` interface obj is available in the list of users in the Role called `subrole` which is already created at the weblogic app'n server, if the client username is available in the subrole then it will return true otherwise return false.

```
return fno - sno;
```

{

```
throw new EJBException("sorry u are not authorized
user in to access this sub business logic
method");
```

{

```
public int mul(int fno, sno)
```

{

```
if (sc.isCallerInRole ("mulrole"))
```

{

```
return fno * sno;
```

{

```
throw new EJBException ("sorry u are not authorized
user in to access this mul business logic
method");
```

{

### Practical procedure

In addition to the practical procedure of the previous app'n called `Jaasone`, do the following

- (1) Create the roles at the app'n server

- 22/
- open server console
  - click on security realm
  - " " myrealm
  - " " roles & policies
  - " " + of global roles
  - " " roles
  - " " new
  - give the name of the role as subrole
  - click on OK
  - " " subrole
  - " " addconditions
  - select user from the list
  - Click on next
  - give the user argument name as ramesh
  - Click on add
  - give another name as suresh
  - Click on add
  - like this add required no.of users in the respective sub role
  - click on finish

Now the role called subrole is created  
with the specified no.of users.

Like this create another role called mul  
role with the required no.of users

(2) Deploy the jar file of the ejb app'n

NOTE Before clicking the finish button make sure that  
the selected option is customroles

22/7/09

JAVA MAIL e-mail is one of oldest networked computer technologies which is used for sending & receiving the msgs over the n/w

\* sending & receiving the msgs in the n/w is very simple, where we should open a connection i.e. socket for communication, transferring the data, & then close the connection with a small set of commands.

\* The Java mail API will provide a set of classes for the implementation of msg based systems in the n/w.

\* The transport protocols that are used for these e-mail are

(1) Simple Mail Transfer Protocol i.e. SMTP protocol

(2) Post Office Protocol Version 3 i.e. POP3 protocol

(3) Internet Message Access Protocol i.e. IMAP protocol

The protocol i.e. used for packaging the msg content is called as Multipurpose Internet Mail Extension Protocol i.e. MIME protocol.

The SMTP protocol is designed for the delivery of msgs i.e. sending the msgs to a mailservers, which means this protocol can't be used for reading the mail msgs.

The POP3 protocol is designed to read the mail msgs i.e. to receive the mail msg from the mail server just like opening the mailbox at real post office, collect & download e-mails, store it locally & remove them from the server.

- \* So the SMTP protocol will send the mail msgs to the mailserver, the POP3 protocol will read the mail msgs from the mailserver
- \* The IMAP is another protocol that may e-mail servers use, in this protocol all msgs are stored in the server, which means the user machines need not download the msg
- \* Internet mail is fundamentally based on ASCII text, does not permit non ASCII data to be used in the msg, later on attaching the non ASCII files to the mail msgs features is introduced by encoding binary files into ASCII code, this ASCII code is transported through the network, at receiving side the received msg is decoded back into the native binary files.
- \* This translation & transmission rules are defined with MIME protocol which is used <sup>for</sup> packaging the msg.
- \* The Java API provides a rich set of classes & interfaces to support this e-mail communication in the n/w.
  - The following practical ex: demonstrate how to send the mail msgs to the mailserver such as James MailServer and how to receive the mail msgs from the same mail server.

Note The James MailServer. Whenever we start on the localhost then it is going to run at the following port no's. i.e:

- (1) SMTP Port no: is 25
- (2) POP3 " " " 110
- (3) Manager " " " 4555

→ He should set the following class path settings to execute the following. Practical Ex:

```
set PATH = D:\sun\jdk16\bin;;
set CLASSPATH = D:\sun\jdk16\lib\tools.jar; E:\naidujzee
|naidumail\mailpath\folder\activation.jar; E:\naidujzee\
naidu \| \| \| mail-1-3.jar; D:\sun\app
Server\lib\javamail.jar;;
JAVA_HOME = D:\sun\jdk16;
```

MAIL  
↓

- (1) sendmail.java
- (2) Receivemail.java
- (3) MyAuthenticator.java

Sendmail.java This is the appn to send the mail msg  
to the James Mail Server which is already running at  
localhost, 25 SMTP port no.

```
import javax.mail.*;
" " " " " " " " " " " " " " " ;
" " " " " " " " " " " " " " " ;
" " " " " " " " " " " " " " ;
public class sendMail
{
 p.s.v.m (String args[]) throws Exception
 {
 DataInputStream stdIn = new DataInputStream
 (System.in);
```

```
System.setProperty ("mail.smtp.host", "localhost");
" " " (" " " " , "25");
```

```
Session sn = Session.getDefaultInstance (System.getProperties (), null);
```

Where `getDefaultInstance` is the static method of `Session` class available in `javax.mail` package that will return an obj of same session class along with the respective property names, property values ie. MailServer host name, mail server smtp port no: available in the `System` class.

```
Transport t = sn.getTransport ("smtp");
Message m = new MimeMessage (sn);
s.o.p ("lnln Enter the senders name");
String sndrname = stdin.readLine ();
m.setFrom (new InternetAddress (sndrname + "@localhost"));
s.o.p ("lnln Enter the recipients name");
String rcptname = stdin.readLine ();
m.setRecipient (Message.RecipientType.To, new InternetAddress (rcptname + "@localhost"));
m.setSentDate (new Date ());
m.setSubject ("Hello");
String totalmsg = "";
while (true)
{
 s.o.p ("lnln Enter the mail msg");
 String msg = stdin.readLine ();
 total msg = totalmsg + "\n" + msg;
 s.o.p ("lnln Do u want to add one more msg
(yes/no)");
```

```

String choice = stdin.readLine();
if (!choice.equalsIgnoreCase ("Yes"))
 break;
}
m.setText (totalmsg);
t.send(m);
S.O.P ("In\n all ur mses are sent to In the mail
server");
S.O.P ("In\n Bye Bye");
}
}

```

### ReceiveMail.java

This is the appn at receiver side to receive the  
 mail mses from the James MailServer already running  
 at local host, 110 pop3 port no: after authentication.

```

import javax.mail.*;
import java.io.*;
public class ReceiveMail
{
 P.S.V.M (String args[])
 {
 try
 {
 System.setProperty ("mail.pop3.host", "localhost");
 " " " " , "110");
 Session sm = Session.getDefaultInstance (System.

```

Where MyAuthenticator is the user defined class to read the recipient user names & passwords from the keyboard of the computer system & to pass these details to the mail servers so that the James Mail server will check for authentication of the connected user with the list of authenticated usernames & pwds already available at James Mail Server

```
Store st = sm.getStore("POP3");
st.connect();
Folder f = st.getFolder("INBOX");
f.open(Folder.READONLY);
Message m[] = f.getMessages();
FileOutputStream fout = new FileOutputStream
("messages.txt");
PrintStream ps = new PrintStream(fout);
S.O.P ("lnln ur received megs are");
int count=0;
for(int i=0; i<m.length; i++)
{
 S.O.P ("ln");
 ps.println ("lnln");
 S.O.P ("Message"+(i+1));
 ps.println ("Message"+(i+1));
 S.O.P ("From" + m[i].getFrom()[0]);
 ps.println (" " " " " ");
 S.O.P ("sentDate" + m[i].getSentDate());
 ps.println (" " " " " ");
 S.O.P ("Subject" + m[i].getSubject());
 ps.println (" " " " " ");
```

```

 s.o.p ("\\n\\n");
 ps.println ("\\n\\n");
 s.o.p ("Message" + m[i].getContent());
 ps.println ("Message" + m[i].getContent());
 s.o.p ("_____");
 ps.println ("_____");
 count++;
}
s.o.p ("\\n\\n ur total msgs are" + count);
ps.println (" " " " " + " ");
}
catch (Exception e)
{
 s.o.p ("you are not valid user to receive the mail msgs");
 return;
}
s.o.p ("\\n\\n -have a nice day");
}
}

```

### MyAuthenticator.java

This is the userdefined class by extending authenticator class available in javax.mail package to read the recipients usernames & pws from the keyboard of the computer system.

```

import java.io.*;
u: javax.mail.*;

```

```
public class MyAuthenticator extends Authenticator
{
```

```
 public PasswordAuthentication getPasswordAuthentica
 -tion()
```

This is <sup>the</sup> overridden method of the super-class  
i.e. authenticator class, this method is automatically  
executed by the Mail server to check for the authenti  
cation of the recipient usernames & PWDs with the  
available data at serverside.

```
try
{
```

```
 DataInputStream stdin = new DataInputStream(System.
 in);
 System.out.println("Enter recipient name");
 String rcptname = stdin.readLine();
 System.out.println("Enter password");
 String rcptpass = stdin.readLine();
 return new PasswordAuthentication(rcptname,
 rcptpass);
```

3 This will return an obj of PasswordAuthentication  
class along with the respective recipient name & Pwd.

```
catch(Exception e)
```

```
{
```

```
}
```

```
return null;
```

```
}
```

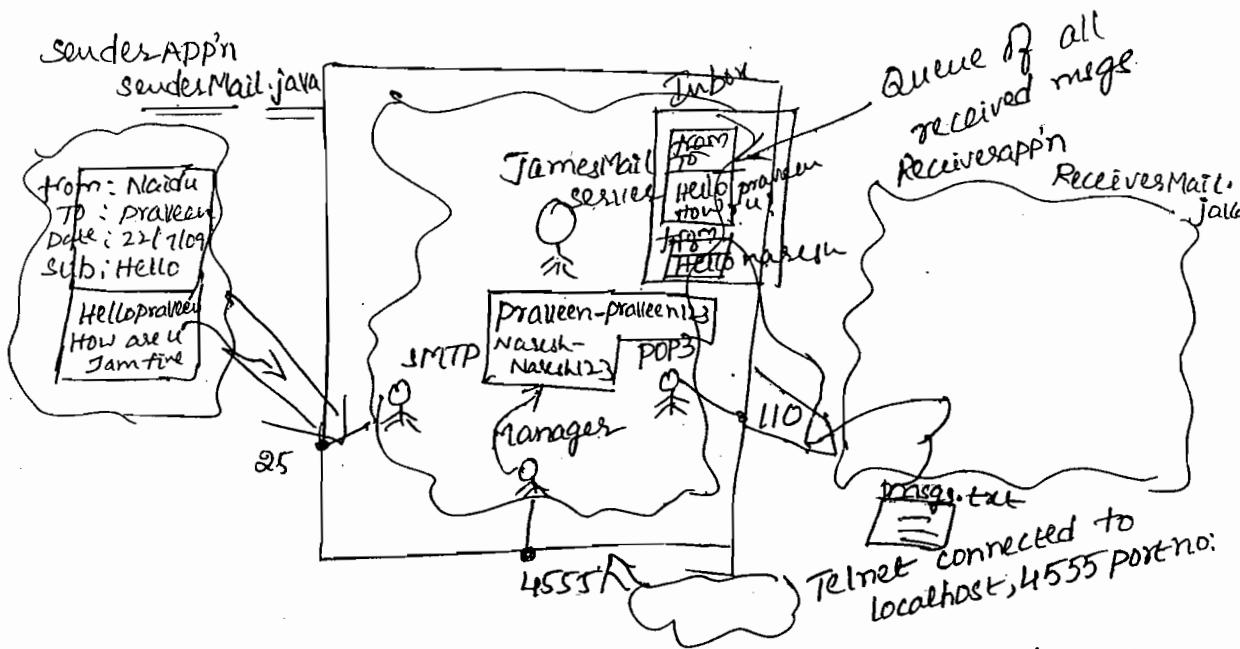
```
}
```

## practical procedure

- (1) compile the java files with the following command.

javac \*.java

- (2) we should start James Mail Server [t: James\bin\run mail click]



- (3) Adding i.e. storing a list of Valid Authenticated username & pWds at JamesMailServer which is already started

running through its Manager port no: i.e. 4555.

→ start → Run → telnet localhost 4555

Login id: root

password: root

Welcome root. Help for a list of cmds

listusers is the cmd is used to see all the list of valid users available at the mailserver.

user: Anjali } already existing usernames

user: chinna }

user: nani }

If we want to add new username & pWd

into the existing list available at the mailserver then

We should give the following command

addrser praveen praveen123 <|

User praveen added

addrser naresh naresh123 <|

User naresh added

- (4) ~~>~~ java Execute the sender side app'n to send the mail msgs such as

> java sendMail.

Enter the senders name: Naidu

" " recipients name: praveen

" " mail msg : Hello praveen....

Do u Want to add one more msg (yes/no); Yes

Enter the mail msg: How are u--.

Do u Want to add one more msg (yes/no); yes

Enter the mail msg: I am fine

Do u Want to add one more msg (yes/no); No

All ur msgs are sent to the mail server

Bye Bye

Enter the senders name: Kiran

" " recipients " : naresh

" " mail msg : Hai naresh--.

Do u Want to add one more msg (yes/no); ~~yes~~ no

All ur <sup>no</sup> msgs are sent to the mail server

Bye Bye.

- (5) Execute receiver side app'n to receive the mailmsg from the mail server after authentication is completed with the following command

> java ReceiveMail

Enter recipient's name: praveen

" Pwd : " 126

u are not valid user to receive

Have a nice day

sgs Enter recipient name: praveen

" Pwd : " 123

Your total mges are: 2

Message 1:

From: Ramesh @ localhost

Sentdate: Wed

Subject: Hello How r u?

Message 2:

From: Naidu @ localhost

Sentdate: Wed

Sub : I am fine

>

Messages.txt

Message 1

From: Ramesh @ localhost

Sentdate: Wed

Subject: Hello How r u?

Message 2:

From: Naidu @ localhost

~~to~~ Sentdate: Wed

Sub: I am fine

ed

