

# Introduction

**What are Enterprise Java Beans?**

**Glossary**

**Why use them? (Or why not.)**

**Enterprise Java Bean server features**

**Enterprise Java Bean roles**

**Enterprise Java Bean programming restrictions**

**How does it work (briefly)**

**Types of beans**

**An example**

**Deployment descriptor**

**Archive Files**

**J2EE Server**

# Glossary

- **J2EE - Java 2 Enterprise Edition -- EJB's + JSP's etc.**
- **JMS - Java Messaging Service**
- **JNDI - Java Naming and Directory Interface**
- **JSP - Java Server page. Here this could mean JSPs, servlets, and/or beans installed on the web server**
- **API - Application Programmer's Interface**
- **JVM - Java Virtual Machine**
- **SQL - Structured Query Language**
- **AWT - abstract window toolkit**
- **I/O - input output**
- **J2EE RI - J2EE reference implementation**

- **IDE - Interactive Development Environment**

## **What is an Enterprise Java Bean (EJB)?**

- **Special type of “bean”**
  - bean* - Just a Java class. Typically follows naming conventions for methods to create *properties*. May also implement class XXXBeanInfo to provide information about class.**
- **Enterprise Java Bean is a bean designed to run inside an EJB server.**
- **From the server, we get certain benefits. By the nature of the EJB and server interaction, we also have several limitations.**
- **EJB is supplemented by a *deployment descriptor*: an XML file which provides the container with information about how the bean is to be deployed.**

# Why use Enterprise Java Bean?

- EJB servers are expensive.

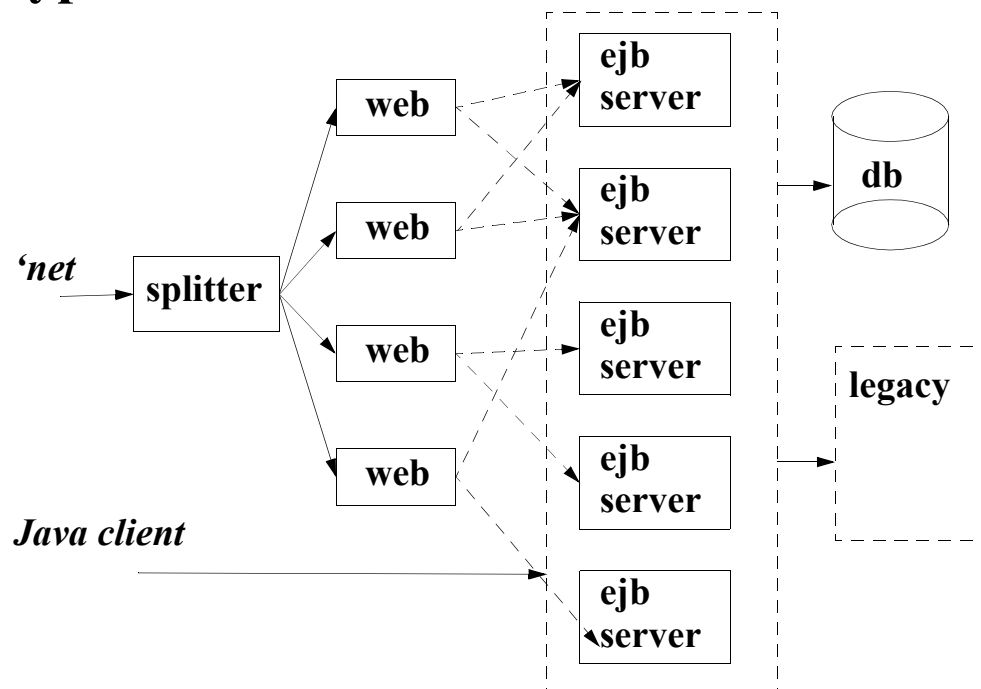
*Sun Reference Implementation (RI)* is free -- and worth every penny!

## Don't overspend on application tech

- Best used where scale of application -- primarily transaction rate -- warrants performance demands.
- For sites with less traffic, using HTML/JSP/servlets/beans with a database backend may be simpler, cheaper, and easier.

Heidi J. C. Ellis and Gerard C. Weatherby

## Typical EJB scenario



## EJB server characteristics

- **May be distributed over multiple hardware boxes**
  - Software manages -- location transparency
- **Provides services**
  - Naming**
  - Security**
  - Concurrency**
  - Transactions**
  - Persistence**
  - Distributed objects**
  - Asynchronous Messaging**
  - Interoperability**

Heidi J. C. Ellis and Gerard C. Weatherby

## Enterprise Java Bean roles

- **EJB specification defines five roles humans can play**
    - Enterprise Bean Provider*** - application domain expert who writes the Java code and the deployment descriptor. Work product is **ejb jar files**.
    - Application Assembler*** - assembles beans into larger units, perhaps with other components (e.g. JSP). Also produces **jar files**.
    - Deployer*** - installs the jar files in specific environment
- Note:** Whether these roles, especially the first two, will be performed by different people is questionable.
- Server provider*** - provides the environment to run EJBs in. (BEA, IBM, Sun. etc.)

***Container provider* - provides the thing the server runs in.**

**Note: As of EJB 2.0, there is no delineation of the interface between container and server. Right now, they're one thing.**

**We'll consider container and server synonyms.**

- **Details are in the EJB specification**

**Reference: 3.1 of ejb spec.**

## **What you can't do**

- **The EJB environment restricts what you're allowed to do. Specifically, you can not:**

**Have modifiable static fields (so make them final)**

**Use thread synchronization API or manipulate threads**

**Attempt to output to a display via AWT or read the keyboard**

**Access files and directories in the file system**

**Accept or listen on a socket**

**Alter Socket / stream handler factories**

**Use the reflection API or otherwise attempt to bypass security rules**

**Alter the JVM, class loader, security manager, alter pred-**

**finer I/O streams**

**Read or write file descriptors**

**Access security policy or related objects**

**Use the subclass / object substitution features of the Serialization protocol**

**Pass *this* as a parameter or return it. (Equivalent methods are provided in the J2EE API)**

**Reference: 24.1.2 of ejb spec.**

## **How does it work?**

- **In general each EJB class you write interacts with other EJBs via interfaces. Clients (JSP or standalone) also interact with EJBs this way.**
- **There is not a direct connection between calling code and the EJB implementation. The EJB server generates code which sits between client and EJB.**

**This code is where the server implements the required services.**

**Details necessary for the server to implement the code are contained in the deployment descriptor.**

## What kinds of EJBs are there?

- ***Entity Beans*** - represent persistence objects, typically stored in a database. They can be
  - Container managed** -- meaning the server generates the necessary SQL to insert/update/delete data.
  - Bean managed** -- the programmer writes the necessary code
- ***Session Beans*** -- represent an interaction or work process which lies on behalf on the client. They can be
  - Stateless** - no information is retained by the object between method calls
  - Stateful** - information is retained
- ***Message Beans*** -- process Java Message Service (JMS) messages

## Types of interfaces

- **Entity and Session Beans can support two kinds of client interfaces**
  - Home* interfaces**
    - Used to create, find and destroy bean instances**
    - Obtained via a name (JNDI)**
  - interface* (business interface)**
    - Provides business logic API**
    - Obtained from home interfaces**
- **Each type of interface has two versions**
  - Remote interfaces can be used anywhere**
  - Most flexible**

**Local interfaces can only be used inside the EJB server**

**More efficient**

- Type of interface determine by which interface is extended  
*EJBObject, EJBLocalObject, EJBHome, EJBLocalHome*
- Bean itself extends yet another interface *SessionBean* or *EntityBean*

**Note that, although the EJB implements the methods on a local or remote interface, it does *not* have an inheritance relationship with them**

**If you mess up, your Java code will probably compile but the server generated code fails to work right**

Heidi J. C. Ellis and Gerard C. Weatherby

## Example - Staff

- Consider the STAFF table used in Database Systems.

```
ENAME  NOT NULL  VARCHAR2(10) //primary key
ETITLE                VARCHAR2(10)
ESALARY               NUMBER(5)
DEPT                  VARCHAR2(4)
```

- First, let's define the remote interface

```
public interface StaffRemote extends javax.ejb.EJBObject {

    public String getName() throws java.rmi.RemoteException;

    public String getTitle() throws java.rmi.RemoteException;
    public void setTitle(String name) throws java.rmi.RemoteException;
    ...
}
```

**Note there's no *setName* 'cause name is the primary key.**

- **Next, let's define the (remote) home interface**

```
import javax.ejb.CreateException;
import javax.ejb.FinderException;
import javax.rmi.RemoteException;
public interface StaffHome extends javax.ejb.EJBHome {
    public StaffRemote create(String name)
        throws CreateException,RemoteException;
    public StaffRemote findByPrimaryKey(String name)
        throws FinderException,RemoteException; }
```

**create is used to make a new staff member**

**findByPrimaryKey looks up an existing staff member**

- **The next step is to implement the bean itself**

**There are a bunch of methods on the interface we don't care about and will implement as no-ops**

### **Common pattern in EJB**

Heidi J. C. Ellis and Gerard C. Weatherby

```
package edu.rh.ejb;
```

```
import javax.ejb.CreateException;
import javax.ejb.EJBException;
import javax.ejb.EntityContext;
```

```
public abstract class StaffBean implements javax.ejb.EntityBean {
    protected EntityContext context;
    public void setEntityContext(EntityContext c){
        context = c;
    }
    public void unsetEntityContext(){
        context = null;
    }
    public void ejbActivate(){ }
    public void ejbPassivate(){ }
    public void ejbLoad(){ }
    public void ejbStore(){ }
    public void ejbRemove(){ }
```

```

/** supports create call on Homeinterface*/
public String.ejbCreate(String name)
    throws CreateException
{
    setName(name);
    return null; //don't ask
}

public void.ejbPostCreate(String name) { }

/** CMP 2.0 persistence methods */
public abstract String getName( );
public abstract void setName(String name);
public abstract String getTitle( );
public abstract void setTitle(String name);
}

```

Heidi J. C. Ellis and Gerard C. Weatherby

## • Client code

```

import javax.naming.Context;
import javax.naming.InitialContext;
import javax.rmi.PortableRemoteObject;
import edu.rh.ejb.StaffHome; //etc.

public class StaffClient {
    public static void main(String[] args) {
        try {
            Context initial = new InitialContext();
            Object objref = initial.lookup("java:comp/env/ejb/StaffBean");
            StaffHome home = (StaffHome)PortableRemoteObject.narrow(objref,
                StaffHome.class);

            StaffRemote staff = home.findByPrimaryKey("LUKE");
            System.out.println("title is " + staff.getTitle());
        } catch (Exception ex) {
            System.err.println("Caught an exception:" + ex);
            ex.printStackTrace();    }    } }

```

## Deployment descriptors

- **Deployment descriptors are XML files.**
  - Two types: EJB specification defined**
  - Vendor specific**
- **Ways to edit XML files**
  - Use a text editor (error prone)**
  - Use a XML tool (e.g. freeware editor or commercial tool like XMLSpy)**
  - Use a J2EE server specific tool**
    - J2EE RI's deploytool generates necessary files**
    - J2EE aware IDE (Forte, Visual Age, JBuilder, Dreamweaver)**

Heidi J. C. Ellis and Gerard C. Weatherby

**For BEA, EJBGGen was recommended by one of their engineers.** (Insert commands, in the form of comments, directly into the Java source and run through the tool. Free.)

- **Sometimes editing XML directly is easier than using a tool**
  - e.g. Setting SQL when using the J2EE RI**

## Archive files

- **Java archive (jar) files are used to package data for deployment**
  - \*.ear - Enterprise Archive file
  - \*.war - Web archive file
  - \*.rar - resource adapter

Note that often we will have archives within archives.
- Can use jar or packager utilities.
  - Be aware of jar bug.
- Note also that WinZip can read jar files.

## RH J2EE server

- **RH has set up a J2EE RI server on machine “facweb”**
  - Only accessible from within the RH network**
  - currently ftp access only**
  - If possible, probably best to do some development on your own machine (download J2EEdk from Sun). Be aware you’ll need a lot of machine resources (20 MEG memory, 48 threads)**
  - Very touchy environment, especially when using deploytool.**
  - Validator not always correct.**
  - Backup your deployment files (\*.ear, et. al.) often!**
  - Must read Sun’s tutorial to learn how to use.**

# Introduction

- **CMP Entity bean lifecycle**

  - **Callback methods**

- **Deployment descriptors**

  - **Security**

  - **Transactions**

  - **Relationships**

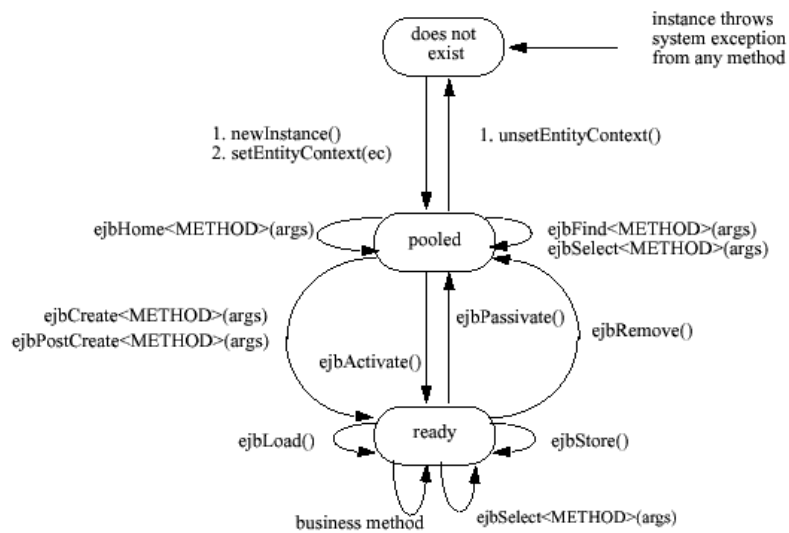
- **BMP Entity beans**

- **EJB base class services**

  - **client side**

  - **EJB side**

# Entity Bean Lifecycle



## Entity bean life cycle (container managed persistence)

- **Three states: does not exist**
- **Pooled - object has been created but does not represent a specific entity**
  - Allows server to optimize memory by not having to create/destroy object instances as client requests come and go**
  - Can service utility methods (home / find / select ) that don't rely on the object corresponding to a particular instance**
- **Ready - associated with a particular database instance (typically an RDBMS row)**
  - Container responsible for keeping the instance in sync with database representation**

## Callback methods

- **setEntityContext - makes the entity context available to the bean.**
  - Access to primary key, interface handles, security info**
- **unsetEntityContext - notification Entity Context no longer valid**
- **ejbCreate - developer responsible for setting database fields, via abstract setXXX methods, to create new instance.**
  - for each *createUvwXYZ* method on home interface, bean implements corresponding *ejbCreateUvwXYZ* method, with matching parameters, in bean class.**
- **ejbPostCreate - called after new row inserted into database. Developer may do additional processing. For example, addi-**

tional database activity (dependent tables) may be performed here.

- **ejbActivate** - specification says it gives bean “the chance to acquire additional resources that it needs” without giving us much clue as to what that might be. For entity bean, *ejbLoad* is more commonly useful.
- **ejbPassivate** - called to release those additional resources.
- **ejbRemove** - called when data is about to be removed from database (someone called remove on business or home interfaces, or due to cascade delete in container managed relationships). Should do any resource releasing that *ejbPassivate* does.
- **ejbLoad** - called after container has set persistent fields to match values in database. Can use to set dependent data, or a

transient representation of data.

- **ejbStore** - called immediately before container is about to store persistent fields in database. Can use to set persistent fields to match a transient representation.

## ejbStore / ejbLoad example

- Consider student table

```
SNO                CHAR(3)
SNAME              NOT NULL VARCHAR2(25)
```

...

- Name is stored as a single field
- We wish to model it as a first and last name
- On bean, define the usual abstract methods corresponding to the database table

```
public abstract String getNumber();
public abstract void setNumber(String number);
```

```
public abstract String getFullName();
public abstract void setFullName(String fullName);
```

- **Add private fields and additional methods to get / set first and last name:**

```
private String firstName;
private String lastName;
```

```
public String getFirstName(){
    return firstName;
}
```

```
public void setFirstName(String first) {
    firstName = first;
}
```

```
public String getLastName() {
    return lastName;
}
```

```
public void setLastName(String last) {
    lastName = last;
}
```

- **Put first and last set get functions on business interface. Do *not* put getFullName / setFullName on business interface.**

```
public abstract String getFirstName() throws RemoteException;
public abstract void setFirstName(String first) throws RemoteException;
```

```
public abstract String getLastName() throws RemoteException;
public abstract void setLastName(String last) throws RemoteException;
```

- **Split name in ejbLoad**

**Note this simple example doesn't handle suffixes, middle**

**initials, and titles.**

```

public void ejbLoad() {
    StringTokenizer st = new StringTokenizer(getFullName());
    switch (st.countTokens()) {
        case 0: ???
            firstName = new String("");
            lastName = firstName;
            break;
        case 1:
            firstName = new String("");
            lastName = st.nextToken();
            break;
        default:
            //simplistic
            firstName = st.nextToken();
            lastName = st.nextToken();
            break;
    }
}

```

- **Implement ejbStore to put pieces back together**

```

public void ejbStore () {
    if (firstName.length() > 0) {
        StringBuffer sb = new StringBuffer(firstName);
        sb.append(' ');
        sb.append(lastName);
        setFullName(sb.toString());
    } else {
        setFullName(lastName);
    }
}

```

- **Container will call ejbStore when it determines it needs to (e.g. at the end of a transaction).**

## Home, Find, and Select methods

- An entity bean may include general methods, than do not apply to a specific instance of a bean.
- A *home* method is written on a home interface, and implemented by the developer.

Method named *abcDef* on home interface must be implemented as *ejbHomeAbcDef* on bean, with matching parameters

- A *find* method is written on a home interface, returns one or more instances of the entity, and is implemented by the Container, using SQL specified by the developer.

Exception: *findByPrimaryKey* does not require SQL be specified

- A *select* method returns one or more instances of the entity, and is implemented by the Container, using SQL specified by the developer, but must *not* be exposed on client interfaces.

Used by other methods in the bean.

Operates in the transaction context of the calling code.

## Example home method

- On interface **StaffHome**:

```
public java.util.Collection getStaffNames ()
    throws RemoteException;
```

- On bean **StaffBean**:

```
static final int NAME_COLUMN = 1;
public java.util.Collection ejbHomeGetStaffNames()
    throws EJBException{
    try {
```

```

TreeSet set = new TreeSet();
InitialContext c = new InitialContext();
DataSource ds = (DataSource)c.lookup("jdbc/Oracle");
Connection conn = ds.getConnection("ejb","femke");
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery("Select ename from STAFF");
while (rs.next()){
    set.add(rs.getString(NAME_COLUMN));
}
conn.close();
return set;
} catch(Exception e){
    System.out.println("ejbHomeGetStaffNames caught" + e);
    e.printStackTrace();
    throw new EJBException("ejbHomeGetStaffNames",e);
}
}

```

## Deployment descriptor

- **Deployment is the XML file which provides information to the container about the beans in place**
- **DTD**

```
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise Java-Beans 2.0//EN" 'http://java.sun.com/dtd/ejb-jar_2_0.dtd'>
```

**Note the DTD is heavily commented with explanations of the elements**

- **Structure - *optional elements italicized*. # discussed later**
- **root - <ejb-jar>**

***description* - describes the beans contained herein**

***display-name* - for use by tools**

***small-icon, large-icon* - also for use by tools. Pictures.**

**enterprise-beans** - information about the beans

**relationships** - information regarding container managed relationships

**assembly-descriptor** - information about security and transactions

**ejb-client-jar** - class files necessary for a client to access beans (not supported by RI?)

- **enterprise-beans** - information about the beans present in the archive. one more more entity, session, or message elements
- **entity**

**description, display-name, small-icon, large-icon** - as before

**ejb-name** - required unique (within jar) name for the bean

**home, remote, local-home, local** - class names of interfaces.

**Entity and session bean must have either a local or remote interfaces, or both.**

**ejb-class** - name of the class with implements the bean

**persistence-type** - must be “Bean” or “Container”

**prim-key-class** - name of the class of the primary key

**reentrant** - “True” or “false” #

**cmp-version** - Version of CMP used. “1.x” or “2.x” (default)

**abstract-schema-name** - Name of the “schema” for this bean. How this maps to a database is vendor specific

**cmp-field** - fields to be managed by container

**primkey-field** - primary key of entity

**env-entry** - environmental (configuration) information readable by bean at runtime #

***ejb-ref*** - information about some other remote EJB used by this one via JNDI

***ejb-local-ref*** - same, except other EJB is local #

***security-role-ref*** - maps a security name used in code to security role defined on server

***security-identity*** - indicates whether method should execute as caller or some specific role

***resource-ref*** - some external resource used by bean, e.g. database connection

***resource-env-ref*** - some additional resource managed by the EJB server used by the bean

***query*** - queries for find methods (other than find by primary key)

## CMP 2.0 example deployment descriptor

```
<entity>
  <description>Academic department of university</description>
  <display-name>DepartmentBean</display-name>
  <ejb-name>DepartmentBean</ejb-name>
  <home>edu.rh.ejb.DepartmentHome</home>
  <remote>edu.rh.ejb.DepartmentRemote</remote>
  <local-home>edu.rh.ejb.DepartmentLocalHome</local-home>
  <local>edu.rh.ejb.DepartmentLocal</local>
  <ejb-class>edu.rh.ejb.DepartmentBean</ejb-class>
  <persistence-type>Container</persistence-type>
  <prim-key-class>java.lang.String</prim-key-class>
  <reentrant>False</reentrant>
  <cmp-version>2.x</cmp-version>
  <abstract-schema-name>Dept</abstract-schema-name>
  <cmp-field>
    <field-name>building</field-name>
  </cmp-field>
  <cmp-field>
```

```

    <field-name>departmentName</field-name>
  </cmp-field>
  <cmp-field>
    <field-name>room</field-name>
  </cmp-field>
  <cmp-field>
    <field-name>chairmanCode</field-name>
  </cmp-field>
  <primkey-field>departmentName</primkey-field>
  <security-identity>
    <use-caller-identity/>
  </security-identity>
</entity>

```

## Environment entries

- The bean developer can allow some behavior to be obtained from the deployment descriptor rather than hard-coded.
- Student set first name code:

```

public void setFirstName(String first) {
    Integer maxFirstName;
    try {
        InitialContext jctx = new InitialContext();
        maxFirstName = (Integer)jctx.lookup("java:comp/env/maxFirstName");
    } catch(Exception e) {
        throw new EJBException(e);}
    if (maxFirstName != null) {
        if (first.length() > maxFirstName.intValue())
            throw new EJBException("first name " + first
                + " length of " + first.length()
                + " characters exceeds allowed length "
                + maxFirstName.intValue());
    }
}

```

```

    } else {
        throw new EJBException("can't lookup maxFirstName");
    }
    firstName = first;
}

```

- **The value is specified via the env-entry portion of the EJB's deployment descriptor**

```

<env-entry>
  <description>maximum length of a first name</description>
  <env-entry-name>maxFirstName</env-entry-name>
  <env-entry-type>java.lang.Integer</env-entry-type>
  <env-entry-value>10</env-entry-value>
</env-entry>

```

## Reentrant?

- **Reentrant code is when EJB A calls code on EJB B which then calls a method on EJB A again**

**Prohibited for session beans**

**Only allowed on entity beans if specified by <reentrant> element**

**Not recommended, as it means more work for EJB container**

- **Making a simple method call to another method on the bean is *not* considered reentrant code and is a perfectly fine thing to do.**

## Referencing other beans

```

<ejb-local-ref>
  <ejb-ref-name>ejb/DepartmentBeanLocal</ejb-ref-name>
  <ejb-ref-type>Entity</ejb-ref-type>
  <local-home>edu.rh.ejb.DepartmentLocalHome</local-home>
  <local>edu.rh.ejb.DepartmentLocal</local>
  <ejb-link>DepartmentBean</ejb-link>
</ejb-local-ref>

```

- **Means this bean (StaffBean) references the Entity Bean with ejb-name “Department Bean” and calls it ejb/DepartmentBeanLocal. Here’s the code:**

```

public String getBuilding() throws EJBException {
    try {
        javax.naming.Context nameContext = new InitialContext();
        DepartmentLocalHome dhome = (DepartmentLocalHome)
            nameContext.lookup("java:comp/env/ejb/DepartmentBeanLocal");
        DepartmentLocal dept = dhome.findByPrimaryKey(getDepartment());

```

```

    return dept.getBuilding();

```

- **Note that “java:comp/env/” is automatically prepended to all calls in the EJB environment**

**Code calls java:comp/env/ejb/DepartmentBeanLocal  
*java:comp/env/* + “*ejb/DepartmentBeanLocal*” specified in  
 deployment descriptor**

## Assembly descriptor

- **assembly-descriptor element is parallel to enterprise-beans**

**Optional, as are all child elements:**

**security-role,  
method-permission,  
container-transaction,  
exclude-list**

- **security-role - describes a role used in method permissions**

```
<security-role>
  <description>Can update student</description> <!-- optional -->
  <role-name>registrar</role-name>
</security-role>
```

## Unchecked methods

- **method-permission - describes which roles may call a method**

```
<method-permission>
  <unchecked />
  <method>
    <ejb-name>StudentBean</ejb-name>
    <method-name>*</method-name>
  </method>
</method-permission>
```

- **Unchecked means no restrictions**
- **“\*” means all methods not otherwise specified**

## Restricted access

```

<method-permission>
  <role-name>registrar</role-name>
  <method>
    <ejb-name>StudentBean</ejb-name>
    <method-intf>Remote</method-intf> <!-- optional -->
    <method-name>setLastName</method-name>
    <method-params> <!-- optional -->
      <method-param>java.lang.String</method-param>
    </method-params>
  </method>

```

- **role-name** specifies which role can execute, can be more than one
- **Same method** may be on Local / Remote interfaces -- use **method-intf** to specify
- **Method name** may be overloaded -- use **params** to distinguish

## Transactions

```

<container-transaction>
  <method>
    <ejb-name>StudentBean</ejb-name>
    <method-name>getFirstName</method-name>
  </method>
  <trans-attribute>Required</trans-attribute>
</container-transaction>

```

- **method** same element as on permissions, same options interface, params or not, wildcard “\*”
- **transaction options**
  - NotSupported*- existing transaction suspended while in this method
  - Supports* - participates in transaction if one present, okay to be outside of transaction

***Required*** - must be part of transaction. If call is made outside of a transaction, a new one is automatically created

***RequiresNew*** - a new transaction is always automatically created -- current transaction, if any, is suspended

***Mandatory*** - must be part of transaction. If call is made outside of a transaction, exception is thrown (*TransactionRequiredException* or *TransactionRequiredLocalException* )

***Never*** - can not be part of a transaction (throws *RemoteException* or *EJBException*)

- For CMP 2.0, servers only required to support Required, RequiresNew, and Mandatory options.

## Don't call me

```
<exclude-list>
  <method>
    <ejb-name>StudentBean</ejb-name>
    <method-name>setFullName</method-name>
  </method>
</exclude-list>
```

- Specifies methods which may not be called

**Attempt results in**

Caught an exception:java.rmi.AccessException: CORBA NO\_PERMISSION 9998  
Maybe; nested exception is: ...etc

## Relationships

- The relationship of one entity to another can be either one or many
- One entity may know about the relationship or both may (unidirectional, bidirectional)
- Two entities involved in a relationship
  - 2 entities X 2 cardinalities X 2 directionality = 8 types of relationship
  - Many to one and one to Many bidirectional equivalent -> 7

## Relationship descriptor

```
<relationships>
  <description>Here are the relations</description> <!-- optional -->
  <ejb-relation>... <!-- one or more -->
</relationships>
```

- Each relationship consists of

```
<ejb-relation>
  <description>staff members belong to a department</description>
  <ejb-relation-name>belongs to</ejb-relation-name> <!--both optional -->
  <ejb-relationship-role>...<ejb-relationship-role>
  <ejb-relationship-role>...<ejb-relationship-role>
</ejb-relation>
```

- Each relationship role describes one end of the relationship. Together, they determine the total relationship

## Relationship role

```

<ejb-relationship-role>
  <description>Belongs to department</description>
  <ejb-relationship-role-name>StaffBean</ejb-relationship-role-name>
  <multiplicity>many</multiplicity>
  <cascade-delete/>
  <relationship-role-source>
    <ejb-name>StaffBean</ejb-name>
  </relationship-role-source>
  <cmr-field>
    <cmr-field-name>departmentObject</cmr-field-name>
  </cmr-field>
</ejb-relationship-role>

```

- *description*, *ejb-relationship-role-name* - optional
- *multiplicity* - “one” or “many”
- *cascade-delete* - optional, empty if present. This entity will automatically be deleted if the other entity is deleted. The

other entity must specify a cardinality of “one”

- *relationship-role-source* - name of a EJB specified in current deployment descriptor
- *cmr-field* - If present, implies this entity knows about the relationship. May have optional *description* element. Must have *cmr-field-name*; if *abcDef* is specified, class must have *setAbcDef* and *getAbcDef* methods.

If multiplicity of other entity in relationship is “many”, must have *cmr-field-type* element with value of either:

`java.util.Collection`

`java.util.Set`

Note that if both relational role elements have *cmr-fields*, relationship is bidirectional. If just one *cmr-field* present, relationship is unidirectional.

- **Notes on relationships in J2EE RI. The RI support for relationships seems poor.**

**When trying to model a one-to-many relationship implementation in the database with a foreign key, the implementation attempted to create a third table to link the two existing tables.**

**This is bad.**

**We're not planning on using container managed relationships as part of our exercises.**

## **Bean managed persistence**

- **Entity bean is which developer must provide manage all the database access**

**Disadvantage: more work**

**Advantage: more flexibility, ability to aggregate database information into a single entity**

- **Life cycle is same as CMP bean, except developer must implement**

**ejbCreate, ejbFindByPrimary, ejbRemove, ejbLoad, ejbStore**

- **Consider following table**

<b>CNO</b>	<b>NOT NULL CHAR(3)</b>
<b>CNAME</b>	<b>NOT NULL VARCHAR2(22)</b>
<b>CDESCP</b>	<b>NOT NULL VARCHAR2(25)</b>



```

description = desc;
Connection conn = null;
Statement stmt = null;
try {
    int rows;
    conn = getConnection();
    stmt = conn.createStatement();
    start("insert into ejbcourse (cno,cname,cdesc) values (");
    quote(courseNumber);comma();
    quote(name);comma();
    quote(description); add(")");
    rows = stmt.executeUpdate(getSQL());
    if (rows != 1) {
        throw new CreateException("CourseBean::ejbCreate insert returned "
            + rows + " rows, 1 expected");
    }
    return courseNumber;
} catch (SQLException e) {
    throw new CreateException(e.toString());
}

```

- ```

finally {
    closeIt(stmt);
    closeIt(conn);
}
}

```
- **Note there's no storage for the primary key (cno). Since it's available from the EntityContext, it would be redundant to store it.**
  - **findByPrimary key. Verifies entity in database. Note in bean name is *ejbFindByPrimaryKey***

```

public String ejbFindByPrimaryKey(String courseNumber)
    throws FinderException {
    Connection conn = null;
    Statement stmt = null;
    ResultSet rs = null;
    try {
        int rows;
        conn = getConnection();

```

```

stmt = conn.createStatement( );
start("select cno from ejbcourse where cno =");
quote(courseNumber);
rs = stmt.executeQuery(getSQL( ));
if (!rs.next()) {
    throw new ObjectNotFoundException("No course with course Number "
        + courseNumber);
}
} catch (SQLException e) {
    throw new FinderException(e.toString( ));
}
}
finally {
    closeIt(rs);
    closeIt(stmt);
    closeIt(conn);
}
return courseNumber;

```

- **ejbRemove method. Invoked by a client call to EJBObject.remove()**

```

public void.ejbRemove( ) {
    Connection conn = null;
    Statement stmt = null;
    try {
        int rows;
        String cno = (String)context.getPrimaryKey( );
        conn = getConnection( );
        stmt = conn.createStatement( );
        start("delete from ejbcourse where cno=");
        quote(cno);
        rows = stmt.executeUpdate(getSQL( ));
        if (rows != 1) {
            throw ...

```

- **Example set method. Note use of flag to record object modification**

```
public void setName(String n) {
    name = n;
    dirty = true; //private class boolean
}
```

- **ejbLoad method. Called by container to synchronize object to database**

```
public void ejbLoad() {
    Connection conn = null;
    Statement stmt = null;
    ResultSet rs = null;
    try {
        String cno = (String)context.getPrimaryKey();
        conn = getConnection();
        stmt = conn.createStatement();
        start("select cname,cdescp,cred,clabfee from ejbcourse where cno =");
        quote(cno);
```

```
rs = stmt.executeQuery(getSQL());
if (!rs.next()) {
    throw new EJBException("No course with course Number "
        + cno);
}
name = rs.getString(1); //indexes more efficient than column names
description = rs.getString(2);
credit = rs.getInt(3);
fee = rs.getInt(4);
dirty = false;
....
```

- **ejbStore method. Called by container to sync database with object**

```
public void ejbStore() {
    if (!dirty) { //optimization; avoid unnecessary DB update
        return;
    }
    Connection conn = null;
    Statement stmt = null;
```

```

try {
    int rows;
    // "context" protected member of EntityAdapter
    String cno = (String)context.getPrimaryKey();
    conn = getConnection();
    stmt = conn.createStatement();
    start("update ejbcourse set cname="); quote(name);
    add(",cdescp="); quote(description);
    add(",cred="); add(Integer.toString(credit));
    add(",clabfee="); add(Integer.toString(fee));
    add (" where cno="); quote(cno);
    rows = stmt.executeUpdate(getSQL());
    if (rows != 1) {
        throw new EJBException("CourseBean::ejbStore update returned "
            + rows + " rows, 1 expected");
    }
    ...
}

```

- **getting connected. Obtain database handle from DataSource, which must be configured in server.**

```

private Connection getConnection() throws EJBException {
    try {
        InitialContext c = new InitialContext();
        if (databaseAccount==null) {
            databaseAccount =
                (String)c.lookup("java:comp/env/ejb/databaseAccount");
            databasePassword =
                (String)c.lookup("java:comp/env/ejb/databasePassword");
        }
        DataSource ds = (DataSource)c.lookup("jdbc/Oracle");
        return ds.getConnection(databaseAccount,databasePassword); ...
    }
}

```

**Note use of <env-entry> in deployment descriptor to avoid hard-coding account name/password**

## Base interface services, client side

- **EJBObject** is base interface for remote interfaces. Includes:
  - getEJBHome* - get home interface for this EJB
  - getHandle* - get a serializable handle to the object. #
  - getPrimaryKey* - get the key if it's an entity bean. Get an exception if it's a session bean.
  - remove* - make this go away. Removes image from database if entity bean
  - isIdentical* - tests two interfaces to see if they refer to the same bean. Object.equals can't be used for this, as two stub classes may be connected to / represent the same EJB.
- **EJBLocalObject** is for local interfaces. No *getHandle*  
get home method is *getEJBLocalHome*

- **EJBHome** is base interface for home interfaces. Contains
  - getEJBMetaData*. Returns interface which has methods to
    - getEJBHome*
    - getHomeInterfaceClass*
    - getPrimaryKeyClass*
    - getRemoteInterfaceClass*
    - isSession*
    - isStatelessSession*
  - getHomeHandle* - return serializable object for interface
  - remove* - remove an EJB. Overloaded to accept primary key or Handle

## Using handle example

```
Context initial = new InitialContext();
Object objref = initial.lookup("java:comp/env/ejb/CourseBean");
CourseHome home = (CourseHome)PortableRemoteObject.narrow(objref,
    CourseHome.class);
```

- **to store**

```
CourseRemote course = home.findByPrimaryKey("ejb");
Handle h = course.getHandle( );
ObjectOutputStream oos = new ObjectOutputStream(
    new FileOutputStream("course.ser"));
oos.writeObject(h);
ios.close( );
```

- **to reload later from file**

```
ObjectInputStream ois = new ObjectInputStream(
    new FileInputStream("course.ser"));
Handle h = (Handle)ois.readObject( );
ois.close( );
objref = h.getEJBObjct( );
CourseRemote course = (CourseRemote)PortableRemoteObject.narrow(
    objref,CourseRemote.class);
```

**Note that no naming context lookup or home interface access was required.**

**\*.ser valid across server restarts.**

**Becomes invalid if entity deleted (database row removed)**

## Base interface services, EJB side

- **EntityContext** provides *getEJBObject*, *getEJBLocalObject*, *getPrimaryKey*, *getEJBHome*, *getEJBLocalHome*

**Only valid if in the ready state (or transitioning to/from).**

**getPrimaryKey fails if object not an entity bean.**

**Local methods fail if EJB doesn't have local interfaces.**

**Remote methods fail if EJB doesn't have remote interfaces.**

**Security / transactions methods to be covered with session beans**

- **Note that passing *this* to clients, including other EJBs, is forbidden.**

**Passing the return value of *getEJBObject* et. al. is valid and is used in lieu of *this*.**

## Session beans

- **Purpose of session beans**
- **Types of beans**
  - Stateless**
  - Stateful<sup>1</sup>**
- **Life Cycles**
- **Deployment**
- **Security API**
- **Transaction management**

<sup>1</sup>Spelled with one L, like *zestful*

## Session bean functionality

- **Implements stand alone business functions - operations which are completed in a single call. May or may not have database interaction.**

*Stateless* session beans.

**If acting upon more than one bean or bean method, more efficient**

**Less network traffic.**

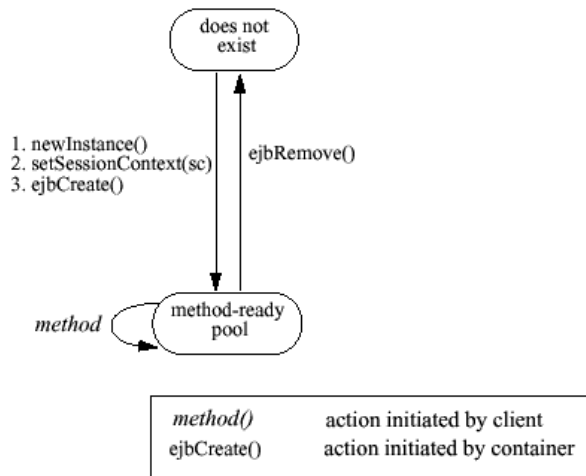
- **Implements a multiple step process -- subsequent calls to the same interface go to the same bean**

*Stateful* session beans

**Maintains conversational state**

# Stateless Session Bean LifeCycle

- **Note subsequent calls to the same interface may not go to same EJB**
- **Stateless bean stores no information specific to a client**
- **Container can manage very efficiently -- doesn't have to match bean instance to a particular client**



# Transaction management

- **Session beans may use either container managed transactions or bean managed transactions**

*Entity Beans* always use container managed transactions

- **Container managed -- use deployment descriptor specification**

```
<container-transaction>
  <method>
    <ejb-name>StudentBean</ejb-name>
    <method-name>getFirstName</method-name>
  </method>
  <trans-attribute>Required</trans-attribute> </container-transaction>
```

- **Bean managed -- use `javax.transaction.UserTransaction` interface. Returned by `getUserTransaction ()` on `SessionContext` or `EntityContext`**

## UserTransaction API

- *begin()* Create a new transaction
- *void commit()* Complete the current transaction
- *int getStatus()* Obtain the status of the current transaction  
Constant from interface `javax.transaction.Status`
- *void rollback()* Roll back the current transaction
- *void setRollbackOnly()* Modify the current transaction such that the only possible outcome of the transaction is to roll back the transaction.
- *void setTransactionTimeout(int seconds)* Modify the value of the timeout value of the current transaction

*Note:* package `javax.transaction` is in both the JDK Standard Edition and the JDK Enterprise Edition

## Accounting example

```
ACCOUNTING_TRANSACTION
ID          NOT NULL NUMBER(5)    --generated by database
TYPE                VARCHAR2(6)
AMOUNT             NUMBER(10,2)
CUSTOMER           VARCHAR2(30)
```

- **Entity bean (local interface)**

```
public interface AccountingTransactionLocal extends javax.ejb.EJBLocalObject {
    public abstract Integer getId() ;
```

```
    /* AccountingTransaction.CREDIT or AccountingTransaction.DEBIT */
```

```
    public abstract int getTransactionType() ;
    public abstract void setTransactionType(int type) ;
    public abstract String getType() ;
    public abstract double getAmount() ;
    public abstract void setAmount(double amount) ;
    public abstract String getCustomer() ;
    public abstract void setCustomer(String customer) ; }
```

- **In a double entry accounting system, each entry is made twice**

**One account gets a credit**

**One account gets a debit**

**Sum of ledger should always be zero**

**Each operation requires actions one two entity beans**

- **Stateless session bean (remote interface)**

```
public interface AccountingFunctionsRemote extends javax.ejb.EJBObject {
```

```
    public void bookEntry(String to, String from, double amount)
        throws RemoteException;
}
```

- **Home interface (remote)**

```
public interface AccountingFunctionsHome extends javax.ejb.EJBHome {
    public AccountingFunctionsRemote create()
        throws CreateException, RemoteException;
}
```

**Stateless bean may only have the single, no argument create call.**

**Passing arguments would be pointless**

**No *findBy* methods -- they only make sense for Entity Beans**

**No remove method -- no state is preserved, so client doesn't need to dispose of instance**

- **To the client, a stateless session bean is essentially a set of standalone methods, not an object in the classical sense.**

## Session bean deployment descriptor

```
<session>
  <display-name>AccountingFunctionsBean</display-name>
  <ejb-name>AccountingFunctionsBean</ejb-name>
  <home>edu.rh.ejb.AccountingFunctionsHome</home>
  <remote>edu.rh.ejb.AccountingFunctionsRemote</remote>
  <ejb-class>edu.rh.ejb.AccountingFunctionsBean</ejb-class>
  <session-type>Stateless</session-type>
  <transaction-type>Bean</transaction-type> ...
```

- **Only the session-type and transaction-type elements are unique to session beans**

*session-type* - “Stateful” or “Stateless”

*transaction-type* - “Bean” or “Container”

*container-transaction* elements will be present for bean only if type is Container

## Bean managed transaction code

- **Note that everything between the *begin* and *commit* either happens or not -- atomic operation**

```
public void bookEntry(String to, String from, double amount) {
  try {
    Context initial = new InitialContext();
    Object objref = initial.lookup("java:comp/env/ejb/AccountingTransaction");
    AccountingTransactionHome home = (AccountingTransactionHome)
      PortableRemoteObject.narrow(objref,
        AccountingTransactionHome.class);
    UserTransaction txn = context.getUserTransaction();
    txn.begin();
    AccountingTransactionRemote trans = home.create();
    trans.setTransactionType(AccountingTransaction.CREDIT);
    trans.setAmount(amount);
    trans.setCustomer(to);
```

```

    AccountingTransactionRemote trans2 = home.create();
    trans2.setTransactionType(AccountingTransaction.DEBIT);
    trans2.setAmount(amount);
    trans2.setCustomer(from);
    txn.commit();
    }
    catch (Exception e) {
        throw new EJBException("bookEntry",e);
    } }

```

- **If *begin* not called, behavior depends on the deployment settings of the AccountingTransaction interface:**

**If *Required*, a transaction will be generated automatically.**

**If *Mandatory*, exception is generated:**

```

javax.ejb.TransactionRequiredLocalException
<stack trace snipped...>
    at edu.rh.ejb.AccountingFunctionsBean.exerciseTransaction(AccountingFunc-
tionsBean.java:163)
163:     AccountingTransactionLocal trans = home.create();

```

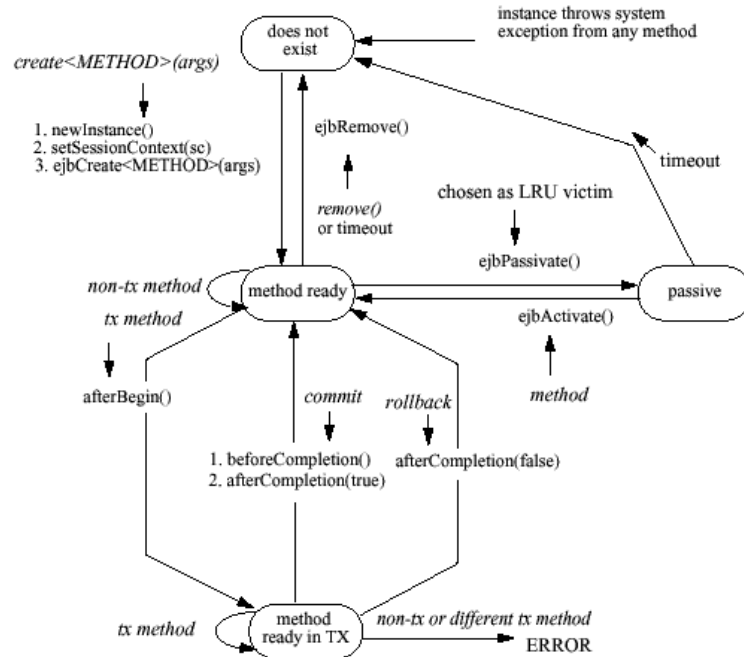
## Client side transactions

- **Client may initiate transaction.**
  - ***UserTransaction* obtained via JNDI Context.**
- ```

UserTransaction txn = (UserTransaction)
    initial.lookup("java:comp/UserTransaction");

```
- **Note the name does not use the typical *env* element.**
  - **J2EE RI does not support client side transactions.**

# Stateful Session Bean LifeCycle



- `ejbCreate`, `ejbActivate`, `ejbPassivate`, `ejbRemove` equivalent to Entity Bean lifecycle.

Home interface may contain multiple *create* methods, and *remove* is available on home interface and business interface.

- Object data must be serializable or explicitly managed via `ejbCreate/ejbActivate` and `ejbPassivate`
- Method Ready in Transaction state notification only available if stateful session bean implements *SessionSynchronization* interface

*afterBegin* - called after transaction started. Specification suggests database information could be cached at this point.

Cache -- copy of data stored in a faster medium.

*beforeCompletion* - called just before transaction ends.

***afterCompletion(boolean committed)*** - called after transaction ends. “committed” is true if transaction was committed, false if rolled back

- **Stateful EJBs may continue transactions across multiple method calls**

**Will time-out if too much time passes**

**Setting is vendor-specific**

**Transaction rolled back**

## Stateful Bean Example

- **Accounting query object which remembers name of customer**
- **Home interface (remote)**

```
public interface AccountingQueryHome extends javax.ejb.EJBHome {
    public AccountingQueryRemote create()
        throws CreateException,RemoteException;
}
```

- **Remote interface**

```
public interface AccountingQueryRemote extends javax.ejb.EJBObject {

    public void setCustomer(String customer) throws RemoteException;
    public String getCustomer() throws RemoteException;
    public double getAmount() throws RemoteException;
    public double getCredits() throws RemoteException;
    public double getDebits() throws RemoteException;
    public java.util.Collection getAccountIds() throws RemoteException;
}
```

- **Bean (excerpts)**

```
//SessionAdapter implements javax.ejb.SessionBean
public class AccountingQueryBean extends SessionAdapter {
    private String customer;
    private Connection conn;
    ...
    public void ejbCreate() {
        acquireResources();
    }
    public void ejbActivate(){
        acquireResources( );
    }
    public void setCustomer(String customer) {
        this.customer = customer;
    }
    public double getCredits() {
        return getSum("CREDIT");
    }
}
```

```
public void ejbPassivate(){
    try { conn.close();} catch (Exception e) {}
    conn = null;
    ...
}
private void acquireResources(){
    try {
        InitialContext c = new InitialContext();
        DataSource ds = (DataSource)c.lookup("jdbc/Oracle");
        conn = ds.getConnection(databaseAccount,databasePassword);
        ...
    } catch (Exception e) {
        throw new EJBException(e);
    } }

private double getSum(String type) {
    checkCustomer( );
    ... sql query
}
```

```
private void checkCustomer() {
    if (customer == null) {
        throw new IllegalStateException("Customer not set");
    } ...
}
```

- **Note *customer* is maintained on behalf of client.**

**String is serializable, so container can passivate.**

**Most fundamental Java types are serializable**

- **Database connection is acquired and released as object activated / passivated.**
- **Note there are no transaction operations. Container managed transactions are specified in deployment descriptor:**

```
<session>
  <ejb-name>AccountingQueryBean</ejb-name>
  <home>edu.rh.ejb.AccountingQueryHome</home>
  <remote>edu.rh.ejb.AccountingQueryRemote</remote>
  <ejb-class>edu.rh.ejb.AccountingQueryBean</ejb-class>
```

```
<session-type>Stateful</session-type>
<transaction-type>Container</transaction-type>
...
<container-transaction>
  <method>
    <ejb-name>AccountingQueryBean</ejb-name>
    <method-name>getDebits</method-name>
  </method>
  <trans-attribute>Required</trans-attribute>
</container-transaction>
<container-transaction>
  <method>
    <ejb-name>AccountingQueryBean</ejb-name>
    <method-name>setCustomer</method-name>
  </method>
  <trans-attribute>Supports</trans-attribute> ...
```

- **Note that *getDebit* requires a transaction, since it goes to the database. *setCustomer* only accesses the bean, so a transaction is not required, but it's not forbidden either.**

## Transaction requirements and exceptions

- **Stateless session bean must complete transaction within method call. Failure to do so throws exception:**

Caught an exception: `java.rmi.ServerException: RemoteException occurred in server thread; nested exception is:`

`java.rmi.RemoteException: Stateless SessionBean method returned without completing transaction`

This can occur if *begin* is not followed by *rollback* or *commit*

- **If *setRollbackOnly* is called, must follow with *rollback*.**
- **Attempting to perform action requiring transaction after *setRollbackOnly* throws *TransactionRolledBackLocalException* or *TransactionRolledBackException***
- **If *setRollbackOnly* called, *commit* will throw *RollbackException*.**

- **Calling *commit* or *rollback* without *begin* throws *IllegalStateException***
- **A `SystemException` (subclass of *RuntimeException*, including *EJBException*) leaving an EJB method causes container to:**
  - Rollback any transaction in progress**
  - Log error**
  - Trash the bean**
    - If stateful, subsequent client calls throw *NoSuchObjectException*.**
- **An application exception (not a subclass of *RuntimeException*) does not automatically rollback transaction.**

## Security API

- *isCallerInRole* can be used to see if caller is in a particular security role. Mapping of name is done in deployment descriptor:

```
<security-role-ref>
  <role-name>bigDebit</role-name> <!-- name used in code -->
  <role-link>viewLargeDebit</role-link><!-- role defined in container -->
</security-role-ref>
```

- *getCallerPrincipal* returns *java.security.Principal* interface.

**Principal just has method to return name.**

```
private double debitLimit; //set in acquireResources from <env-entry>
private static final String debitRole = "bigDebit";

public double getDebits() {
    double debits = getSum("DEBIT");
```

```
if ((debits > debitLimit) && (!context.isCallerInRole(debitRole)) ) {
    throw new EJBException("Caller " + context.getCallerPrincipal() +
        " must be in role " + debitRole +
        " to retrieve information about total debits exceeding " +
        debitLimit);
}
return debits;
}
```

**For caller ejb, not assigned “viewLargeDebt” in container, attempting query of large debtor returns:**

Caught an exception:java.rmi.ServerException: RemoteException occurred in server thread; nested exception is:

java.rmi.RemoteException: Caller ejb must be in role bigDebit to retrieve information about total debits exceeding 500.37

## Database isolation and locking

- **Three specific issues**

***Dirty reads*** - occur when one client reads uncommitted changes entered by another client

***Repeatable reads*** - occur when a client's view of data doesn't change during the scope of a transaction

Even ***committed*** changes by another user are not seen

Acting on "snapshot" of database

May be out of date

***Phantom reads*** - occur when client can see uncommitted new records added to database by another client

## Database locks

- ***Read lock*** - others can read but can't change. Ensures repeatable reads.

LOCK TABLE SHARE

- ***Write lock*** - others can read but can't change. Others will have dirty reads.

SELECT ... FOR UPDATE

LOCK TABLE ROW EXCLUSIVE

- ***Write lock exclusive*** - exclusively locked. Prevents dirty reads (if you can't read the data, it can't be dirty)
- ***Snapshot*** - database maintains an image of the data as of the time transaction started

SET TRANSACTION READ\_ONLY

## Isolation levels

- **Obtained via *java.sql.Connection.getTransactionIsolation()*;**  
*TRANSACTION\_NONE* - **no transactions on this connection**  
*TRANSACTION\_READ\_COMMITTED* - **Dirty reads are prevented; non-repeatable reads and phantom reads can occur.**  
 SET TRANSACTION READ COMMITTED  
 Oracle default  
*TRANSACTION\_READ\_UNCOMMITTED* - **Dirty reads, non-repeatable reads and phantom reads can occur.**  
*TRANSACTION\_REPEATABLE\_READ* - **Dirty reads and non-repeatable reads are prevented; phantom reads can occur.**  
 SET TRANSACTION READ ONLY

Not technically repeatable read per Jdbc due to phantom reads

- **Can query database via *DatabaseMetaData*:**  

```
public void test() throws SQLException{
    DatabaseMetaData md = conn.getMetaData( );
    test(md,Connection.TRANSACTION_NONE ,"TRANSACTION_NONE ");
    ... }
public void test(DatabaseMetaData md,int level, String desc)
    throws SQLException{
    System.out.println("Database support for " + desc + " is "
    + md.supportsTransactionIsolationLevel(level));}
```
- **Oracle supports read committed & serialiazable**
- **Can set via *Connection.setTransactionIsolation***

# Message Beans

## Java Message Service

### Interface containment hierarchy

### Publishing a message

### Receiving message

### Types of messages

## Message Driven Beans

### Deployment descriptor

### MDB Transactions, Exceptions

# Java Message Service

- **API to encapsulate multiple messaging systems.**
- **Two types of messages delivery models**
  - Queue - end to end delivery to a single receiver**
    - Point to point model**
    - Requires acknowledgement**
  - Topic - delivery to multiple recipients**
    - Publish - subscriber model**
- **JMS provider provides services**
  - Reliability - will resend unacknowledged messages. Persistent messages are back in secondary storage to survive server crashes.**

**Durability - client may register as durable subscriber. JMS Provider will store messages received while client not running.**

- **Messaging is asynchronous.**

**Sender and receiver operate independently of each other**

**Receiver may consume messages synchronously:**

*receive* blocks until message retrieved

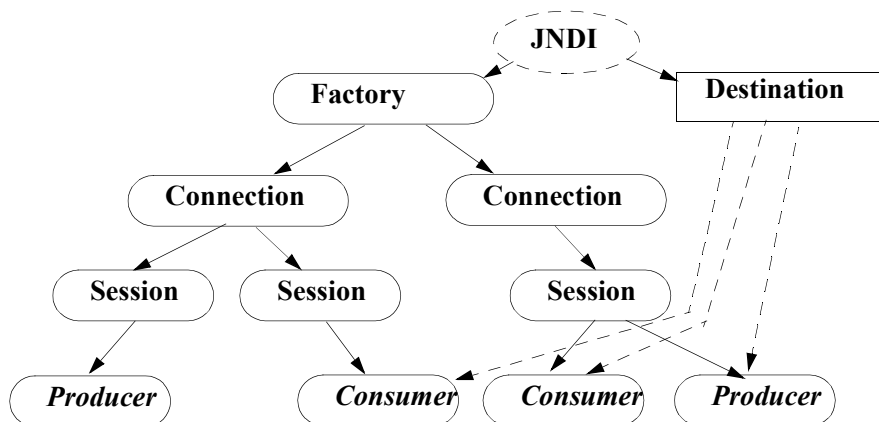
*receive* with timeout waits specified period

*receiveNoWait* polls for message available

**Receiver may receive messages asynchronously**

*setMessageListener* specifies object to receive messages as they arrive.

## Interface containment hierarchy



- **Factory - *TopicConnectionFactory, QueueConnectionFactory***
- **Destination - *Queue, Topic***
- **Connection - *TopicConnection, QueueConnection***

- **Session** - *TopicSession, QueueSession*
- **MessageProducer** - *TopicPublisher, QueueSender*
- **MessageConsumer** - *TopicSubscriber, QueueReceiver*

## Example - send primary key info

```
public class UniversityClassPK implements java.io.Serializable {
    private String courseNumber;
    private String section;
    public UniversityClassPK(String courseNumber, String section) {
        this.courseNumber = courseNumber;
        this.section = section;
    }
    public String getCourseNumber() {return courseNumber;}
    public String getSection() {return section;}
    ...
}
```

## Publishing a message

```
InitialContext c = new InitialContext();
TopicConnectionFactory topicConnectionFactory = null;
TopicConnection topicConnection = null;
Topic topic = null;
topicConnectionFactory = (TopicConnectionFactory)
    c.lookup("java:comp/env/jms/ClassTopicConnFactory");
topic = (Topic)c.lookup("java:comp/env/jms/ClassTopic");
topicConnection = topicConnectionFactory.createTopicConnection();
topicSession = topicConnection.createTopicSession(true,0);
topicPublisher = topicSession.createPublisher(topic);
UniversityClassPK pk = getPK();
MapMessage message = topicSession.createMapMessage();
message.setStringProperty("action", "create");//set header property
//can only set primitive types and wrappers, so break key down into strings
message.setString("course",pk.getCourseNumber());
message.setString("section",pk.getSection());
topicPublisher.publish(message);
topicConnection.close();
```

- **Example from an Entity Bean**  
 Okay to send message, bad practice to try to receive one  
 Interferes with server efficiency by making bean  
 method long running
- **Topic and TopicConnection configured in server and deployment descriptor**
- ***createTopicSession* parameters specify transaction and acknowledge mode when JMS used outside of EJB container ignored in EJB, (true, 0) recommended values**
- ***setStringProperty* defines part of message header available on all messages can be used to select message**

## Receiving message

```
//Client application -- running outside J2EE container
TopicConnectionFactory topicConnectionFactory = null;
TopicConnection topicConnection = null;
TopicSession topicSession = null;
Topic topic = null;
TopicSubscriber topicSubscriber = null;
topicConnectionFactory = (TopicConnectionFactory)
    initial.lookup("java:comp/env/jms/TopicConnectionFactory");
topic = (Topic) initial.lookup("java:comp/env/jms/Topic");
topicConnection = topicConnectionFactory.createTopicConnection();
topicSession = topicConnection.createTopicSession(false,
    Session.AUTO_ACKNOWLEDGE);
topicSubscriber = topicSession.createSubscriber(topic,"action <> 'updated'",false);
topicSubscriber.setMessageListener(
    new MessageListener() {
        public void onMessage(Message m) {
            try {
                if (!(m instanceof MapMessage)) { return; }
            }
        }
    }
);
```

```

    MapMessage mm = (MapMessage)m;
    System.out.println("action " + mm.getStringProperty("action"));
    System.out.println("courseNumber " + mm.getString("course"));
    System.out.println("section " + mm.getString("section"));
} catch(Exception e) {
    System.out.println("onMessage exception " + e);
}
} //onMessage
} /*anonymous inner class */);
topicConnection.start();
...
topicConnection.close();

```

- **Session.AUTO\_ACKNOWLEDGE specifies server should automatically send acknowledgement.**

**DUPS\_OK\_ACKNOWLEDGE specifies server should acknowledge, but that duplicate messages are okay. Allows server to delay acknowledgement for efficiency.**

**CLIENT\_ACKNOWLEDGE specifies client will call**

***acknowledge* method of the *Message* interface.**

- **Second argument to *createSubscriber*, "action <> 'updated' ", is a *message selector*. Only messages with a header property of action not equal to *updated* will be received.**

**Valid syntax is similar to SQL syntax. A full description is in the [javadoc for the Message interface](#).**

- **Third argument, "noLocal" to *createSubscriber* specifies whether server should *not* deliver messages sent on this particular *TopicConnection*.**

**If true, local messages not delivered.**

- ***setMessageListener* specifies an object which implements the *MessageListener* interface, which has a single method**

```
public void onMessage(Message m)
```

**Example uses an anonymous inner class to implement.**

- **Messages not received until TopicConnection *start* method is called.**

**Messages are received in a separate thread.**

- ***close* should be called on subscribers (actually all consumers and producers), sessions, and connections when they are not longer needed.**

**Close calls “cascade” -- closing a connection automatically closes all sessions, closing a session closes all producers and consumers.**

- **MessageListener will receive messages in a separate thread.**
- **Each Session essentially represents a thread and transaction  
Sessions should used by only one thread.**

**Only method safe to call from another thread is *close*.**

**Multi threaded programs should create a separate session for each thread.**

- **If the first argument to *createTopicSession* / *createQueueSession* is true, messages are transacted.**

**Acknowledge occurs when transaction committed -- second argument to createXxxSession is ignored**

**Session has *commit* and *rollback* methods to commit (send and/or acknowledge receipt of messages)**

**Rollback trashes messages “sent” and does not acknowledge any messages received -- JMS provider should redeliver**

**Different mechanism used inside EJB container.**

## Types of messages

- ***Message*** - base interface for other types. Has no body, just header information.

***messageId*** - unique, provider assigned ID. Clients may call *MessageProducer.setDisableMessageID* to disable and reduce overhead.

***JMSTimestamp*** - time provider received message for transmission. Clients may call *MessageProducer.setDisableMessageTimestamp* to disable and reduce overhead.

***JMSCorrelationID*** - used to link message with some other. Not all providers required to support.

***JMSReplyTo*** - used to specify a *Destination* reply should be set to. May be null.

***JMSDestination*** - normally set by *MessageProducer*.

***JMSDeliveryMode*** - may be *DeliveryMode.PERSISTENT* or *DeliveryMode.NON\_PERSISTENT*. If persistent, provider must store message in stable storage to allow delivery in event of transient failure. Non-persistent reduces overhead, message may be dropped. Persistent is default. Set by *MessageProducer.setDeliveryMode*.

***JMSRedelivered*** - flag indicated message is being redelivered. Implies a previous delivery was not acknowledged.

***JMSType*** - indicates type of message. Values and usage vendor specific -- some may require type be set. Recommended to use symbolic values (e.g. properties file to configure).

***JMSExpiration*** - when the message should be discarded by provider, even if undelivered. Value is same long value used by *java.util.Date*: number of milliseconds since 1/1/1970.

Provider sets by when message sent by adding *MessageProducer.getTimeToLive()* to current time. Clients call *MessageProducer.setTimeToLive* to configure. Default value of 0 means message does not expire -- JMSExpiration will also be zero in this case.

*JMSPriority* - value between 0 (low) and 9 (high). Providers not required to strictly deliver messages in order of priority but should “do their best” to send to deliver “expedited” (5-9) messages before normal (0-4). Set by *MessageProducer.setPriority*.

**Note:** values set by MessageProducer may be changed on a per message basis by using overloaded forms of *send* or *publish*.

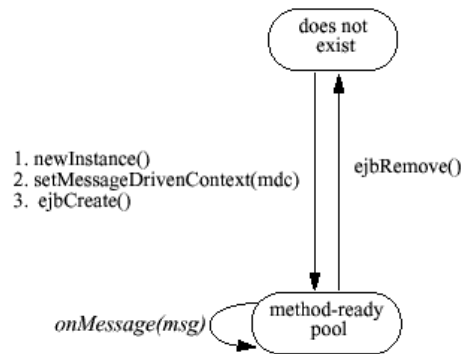
- Arbitrary additional named properties may be set on header. Values must be primitive (e.g. int, double) or java.lang wrap-

pers for primitive values (Integer, Double) or Strings.

- *TextMessage* is a message with a text payload.
- *MapMessage* is a message with a java.util.Map containing String keys and primitive type wrappers and Strings as values.
- *ObjectMessage* contains a *Serializable* object. This can be a *java.util.Collection* of Serializable objects.
- *StreamMessage* contains a writable and readable sequence of primitive data types. Similar to *java.io.DataInputStream*.
- *BytesMessage* contains a set of uninterpreted bytes. Designed to support a preexisting message format unknown to the JMS API.

# Message Driven Beans

- **Session and Entity EJB may send messages, but receiving would muck up efficient use of beans -- would hang in methods.**
- **Message Driven Bean (MDB) introduced in ejb 2.0 to support message receiving**
- **Very simple lifecycle.**
- ***MessageBean* interface has three methods -- *ejbCreate*, *setMessageDrivenContext*, and *ejbRemove*.**
- **No concept of a “current caller” -- MDB just receive mes-**



**sages. Security identity is determined by deployment descriptor, not sender of message. (Can't specify *use-caller-identity* in deployment descriptor.)**

- **Essentially stateless -- container may create multiple instances of MDB. Same instance may not receive all messages of specified type.**

```

public class MessageAdapter implements javax.ejb.MessageDrivenBean,
    javax.jms.MessageListener {
    protected javax.ejb.MessageDrivenContext context;

    public void setMessageDrivenContext(javax.ejb.MessageDrivenContext context) {
        this.context = context;
    }
    public void ejbCreate() {}

    public void ejbRemove() {
        context = null; }
    ...
  
```

- **MessageDrivenContext extends EJBContext but adds no methods.**
- **MDB must separately extend the *MessageListener* interface.**

**Specification designed this way because future MDBs may be able to implement some other message system besides JMS.**

```
public void onMessage(Message m) {
    try {
        ObjectMessage om = (ObjectMessage)m;
        UniversityClassCommand cmd =
            (UniversityClassCommand)om.getObject();
        if (cmd.isCancel()) {
            cancel(cmd); //additional private method on MDB class
        }else if (cmd.isSchedule()) {
            schedule(cmd); //additional private method on MDB class
        } catch(Exception e) {
            throw new EJBException("UniversityClassBean",e);}
    }
```

## Deployment descriptor

- **Note there's no setting up of connection in *onMessage* -- everything is specified in deployment descriptor.**

```
<message-driven>
  <display-name>UniversityClassMessageBean</display-name>
  <ejb-name>UniversityClassMessageBean</ejb-name>
  <ejb-class>edu.rh.ejb.UniversityClassMessageBean</ejb-class>
  <transaction-type>Container</transaction-type>
  <message-driven-destination>
    <destination-type>javax.jms.Queue</destination-type>
  </message-driven-destination>
```

...

- ***transaction-type* may be Container or Bean. If container, will have entry in assembly-descriptor part:**

```
<container-transaction>
  <method>
    <ejb-name>UniversityClassMessageBean</ejb-name>
```

```

    <method-name>onMessage</method-name>    </method>
    <trans-attribute>Required</trans-attribute>
</container-transaction>

```

**Allowed values are Required and NotSupported.**

- ***destination-type* may be javax.jms.Queue or javax.jms.Topic.**

```

<message-driven>
  <ejb-name>UniversityClassMessageBean</ejb-name>
  <ejb-class>edu.rh.ejb.UniversityClassMessageBean</ejb-class>
  <transaction-type>Bean</transaction-type>
  <message-selector>action = 'updated'</message-selector>
  <acknowledge-mode>Auto-acknowledge</acknowledge-mode>
  <message-driven-destination> ...

```

- **If *transaction-type* is Bean, *acknowledge-mode* should be set.**

**Valid values are:**

**Auto-acknowledge**

**Dups-ok-acknowledge**

- **Optional *message-selector* can be used to filter messages based on header information.**
- **Note that selector expressions such as action <> 'updated' are not valid XML. Must used entity references or CDATA sections to specify**

```

<message-selector>action &lt;> 'updated'

```

```

</message-selector> or

```

```

<message-selector><![CDATA[action<>'updated']]>

```

```

</message-selector>

```

- **Complete list of message-bean elements. *Italicized are optional.* Those not specific to MDB have same meaning as on Session/Entity Beans.**

*description, display-name*

*small-icon, large-icon*

**ejb-name**  
**ejb-class**  
**transaction-type**  
*message-selector*  
*acknowledge-mode*  
*message-driven-destination*  
*env-entry*  
*ejb-ref, ejb-local-ref*  
*security-identity*  
*resource-ref*  
*resource-env-ref*

- **Note specific Queue or Topic is not specified in EJB deployment descriptor**

**Setting is vendor specific**

## **MDB Transactions, Exceptions**

- **Bean managed transaction message driven beans use same transaction functions as bean managed transaction session beans.**
- **Commit and rollback of message receipt, sending and database transactions are all grouped together.**
- **MDB must not throw application exceptions.**
- **System exceptions result in container managed and uncommitted bean managed transactions being rolled back**

**Bean instance is discarded.**

**Message redelivered to a new instance.**

**Vendor specific limit on redelivery attempts.**

**Error must be logged.**

# Enterprise Java Bean Strategies

Compound Primary Key

Common interfaces

Collections

Aggregate setters

J2EE Example

## Compound Primary Key

- **If a database column has a multiple column primary key, can't use a primitive for the EJB Primary Key.**

```
CREATE TABLE CLASS (  
CNO      CHAR(3)  NOT NULL,  
SEC      CHAR(2)  NOT NULL,  
...  
CONSTRAINT PK_CLASS PRIMARY KEY (CNO, SEC),
```

- **Provide our own class for the primary key.**

```
public class UniversityClassPK implements java.io.Serializable {  
  
    private String courseNumber;  
    private String section;  
  
    public UniversityClassPK(String courseNumber, String section) {  
        this.courseNumber = courseNumber;  
        this.section = section; }  
}
```

```
public String getCourseNumber() {
    return courseNumber; }
```

```
public String getSection() {
    return section;}
```

- **The class, by design, should be immutable.**

**The values of the object cannot be changed.**

**This is important when passing values remotely.**

**Unlike typical inprocess Java calls, rmi calls pass objects by value.**

**Modifying the returned copy *will not* change the original.**

**Immutable objects prevent a user from inadvertently changing the fields of a copy and thinking they're setting the**

**remote values.**

```
public boolean equals(Object o) {
    if (o instanceof UniversityClassPK) {
        UniversityClassPK other = (UniversityClassPK)o;
        return (courseNumber.equals(other.courseNumber)
            && section.equals(other.section) );
    }
    return false;
}
/** required to implement hashCode to be PK */
public int hashCode() {
    return (courseNumber.hashCode()^section.hashCode());
}
```

- **Must implement the *Object.equals* and *Object.hashCode* methods explicitly to be EJB primary key**

**Default implementation of equals checks address of reference; not appropriate for J2EE systems.**

***hashCode* must return the same value for two objects *equals* says are the same.**

**For efficiency in storing in hashed collections, return value should be well distributed.**

**Different objects likely to return different values.**

**Monson-Haefel suggests building a string. Example above uses exclusive-or to create new value.**

```
/** implement toString to have nice output */
public String toString() {
    StringBuffer sb = new StringBuffer("Course ");
    sb.append(courseNumber);
    sb.append(",section ");
    sb.append(section);
    return sb.toString();
}
}
```

- **Last method not required but helpful -- makes printing**

**instance of object *System.out.println(pk)* produce meaningful, readable input.**

**Default implementation prints out name of class and hexadecimal address.**

**Note that building compound Strings with String + operator is very inefficient**

```
return "Course " + courseNumber + ",section " + section
```

**Each + creates a new String object in memory**

**StringBuffer preferred for frequently used calls**

## Common interfaces

- **Beans are required to implement the same methods as their local and remote interfaces.**

**Failure is not detected at compile time, but deployment time.**

**Deployment may be time consuming**

**Beans should not implement remote interfaces ‘cause they would then have to implement EJBObject methods.**

- **Solution for session and bean managed persistence entity beans is to define a business interface:**

```
public interface UniversityClass {
    public String getCourseNumber() throws RemoteException;
    public String getSection() throws RemoteException;
    public String getDay() throws RemoteException;
```

```
    public void setDay(String day) throws RemoteException;
```

...

- **Remote and Local interfaces and bean implementation then extend / implement interface:**

```
public interface UniversityClassRemote
    extends javax.ejb.EJBObject, UniversityClass {
    public abstract UniversityClassPK getClassPK() throws RemoteException;
}
```

```
public interface UniversityClassLocal
    extends javax.ejb.EJBLocalObject, UniversityClass {
    public abstract UniversityClassPK getClassPK();
}
```

```
public class UniversityClassBean extends EntityAdapter
    implements UniversityClass{
```

- **Does not guarantee correct implementation of home, select and find methods.**

- **Since a subclass or interface cannot add a new Exception to *throws* and the remote interface must specify RemoteException, the common interface must also specify RemoteException.**

**Causes local interface methods to now have RemoteException specified, so clients using local interface now have to, unnecessarily, declare or catch RemoteException.**

### **Trade-off**

- **Allows addition of method to remote and local interfaces in one step -- only edit interface file.**
- **Does not work particularly well for CMP 2.0 beans. Why not?**

## **Collections**

- **Possible to use find method to return a collection of entities**  
**Bean returns a collection of primary keys -- container returns a collection of local or remote interfaces to client.**  
**Each reference generates another Entity Bean -- may be expensive**
- **Alternative is to return Collection of lightweight proxy objects or Strings.**

**Example - ejbHomeClassInfo maps to classInfo on home interface**

**Define lightweight proxy class for class information:**

```
public class UniversityClassInfo implements java.io.Serializable {
    private String courseNumber;
    private String section;
```

```

private String day;
private String time;
private String building;
private String room;
public UniversityClassInfo(String courseNumber, String section) {
    this.courseNumber = courseNumber;
    this.section = section; }

public UniversityClassInfo(UniversityClassPK pk) {
    this.courseNumber = pk.getCourseNumber();
    this.section = pk.getSection(); }

public UniversityClassPK getPK() {
    return new UniversityClassPK(courseNumber,section); }

public String getCourseNumber() {
    return courseNumber; }

public String getSection() {
    return section; }

```

```

public void setDay(String day) {
    this.day = day; }

public String getDay() {
    return day; }
...
Implement classInfo on Bean:
public java.util.Collection.ejbHomeClassInfo() {
    Connection conn = null;
    Statement stmt = null;
    ResultSet rs = null;
    try {
        int rows;
        conn = getConnection();
        stmt = conn.createStatement();
        rs = stmt.executeQuery("select cno,sec,cday,ctime,cbld,croom from class");
        LinkedList list = new LinkedList();
        while (rs.next()) {

```

```

UniversityClassInfo info =
    new UniversityClassInfo(rs.getString(1),rs.getString(2));
info.setDay(rs.getString(3));
info.setTime(rs.getString(4));
info.setBuilding(rs.getString(5));
info.setRoom(rs.getString(6));
list.add(info);
}
return list;

```

- **Disadvantage - data returned is not kept in synchronization with what's in database. May go "stale."**

## Aggregate setters

- **While using an entity bean with a remote interface, each set of an attribute is a network round trip.**

**Appropriate if just one or two fields are being edited, but poor if many are.**

- **Can use previous defined lightweight class.**

```

public UniversityClassInfo getInfo() {
    UniversityClassInfo i = new UniversityClassInfo(getPK());
    i.setDay(getDay());
    i.setTime(getTime());
    i.setBuilding(getBuilding());
    i.setRoom(getRoom());
    return i;
}

```

- **Corresponding set function updates the class with single network call.**

**Note check to ensure the *UniversityClassInfo* object corresponds to this bean.**

```
public void setInfo(UniversityClassInfo i) throws EJBException {
    if (!getPK().equals(i.getPK())) {
        throw new EJBException("invalid UniversityClassInfo object passed "
            + i.getPK() + " does not match " + getPK());
    }
    setDay(i.getDay());
    setTime(i.getTime());
    setBuilding(i.getBuilding());
    setRoom(i.getRoom());
}
```

## J2EE Example

- **Sample application consists of three JSP pages, an Entity Bean, a Message Driven Bean, and three web page beans.**
- **JspUnivClass is bean to support listing all classes:**

```
package edu.rh.ejb;
...
public class JspUnivClass {
    public java.util.Collection getInfo() {
        //should use local interfaces for running within J2EE server, but this
        //allows testing as standalone client
        try {
            Context initial = new InitialContext();
            Object objref = initial.lookup("java:comp/env/ejb/Class");
            UniversityClassHome home = (UniversityClassHome)
                PortableRemoteObject.narrow(objref, UniversityClassHome.class);
            return home.classInfo();
        } catch ...
    }
}
```

## Bean encapsulates JNDI lookup and adapts *classInfo* to *get-Info* -- following bean convention.

- JSP page:

```
<%@ page errorPage="classErrorPage.jsp" %>
<%@ taglib uri="/WEB-INF/struts-logic.tld" prefix="logic" %>
<!doctype html public "-//w3c//dtd html 4.0 transitional//en">
<html><head>
  <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
  <meta name="Author" content="Gerard Weatherby">
  <title>List of classes</title>
</head>
<body><center> <h1>Class List</h1>
<jsp:useBean id="juc" class="edu.rh.ejb.JspUnivClass" scope="request"/>
List of classes<br>
<table>
<tr><td>Number</td>
<td>Section</td>
<td>Day</td>
<td>Time</td>
```

```
<td>Building</td>
<td>Room</td></tr>
<logic:iterate id="info" name="juc" property="info">
<tr><td><jsp:getProperty name="info" property="courseNumber"/></td>
<td><jsp:getProperty name="info" property="section"/></td>
<td><jsp:getProperty name="info" property="day"/></td>
<td><jsp:getProperty name="info" property="time"/></td>
<td><jsp:getProperty name="info" property="building"/></td>
<td><jsp:getProperty name="info" property="room"/></td></tr>
</logic:iterate>
</table><p><a href="class.jsp">Edit university classes</a>
</body>
</html>
```

- *logic:iterate* is custom tag available from Apache Jakarta Struts project.

- \* Iterates over collection obtained from *info* property of *JspUnivClass* bean *juc*

- \* assigns each value to the bean named *info*

\* standard *jsp:getProperty* then used to retrieve values from each *UniversityClassInfo* object.

## Class List

List of classes

Number	Section	Day	Time	Building	Room
C33	01	WE	09:00-10:30A.M.	SC	305
C55	01	TH	11:00-12:00A.M.	SC	306
C11	01	MO	08:00-09:00A.M.	SC	305
C11	02	TU	08:00-09:00A.M.	SC	306
P11	01	TH	09:00-10:00A.M.	HU	102
P33	01	FR	11:00-12:00A.M.	HU	201
T11	01	MO	10:00-11:00A.M.	HU	101
T11	02	MO	10:00-11:00A.M.	HU	102

[Edit university classes](#)

### • Editing form source

```
<center> <h1>Class Edit Form</h1>
<html:form action="doClass.jsp" name="classCommand" scope="session"
  type="edu.rh.ejb.UniversityClassCommand" >
  Course number and section are required.
  <table>
  <tr><td>Course Number:</td><td> <html:text maxlength="3" property="course-
  Number"/></td></tr>
  <tr><td>Section:</td><td> <html:text maxlength="3" property="section"/></td>
  </tr>
  <tr><td>Day:</td><td> <html:text maxlength="2" property="day"/></td></tr>
  <tr><td>Time:</td><td> <html:text maxlength="15" property="time"/></td></tr>
  <tr><td>Room:</td><td> <html:text maxlength="3" property="room"/></td></
  tr>
  <tr><td>Building:</td><td> <html:text maxlength="2" property="building"/></
  td></tr> </table>
  <html:submit property="cancelCommand" value="Cancel"/>
  <html:submit property="scheduleCommand" value="Schedule"/>
  <p>
  <a href="list.jsp">List classes</a>
```

## Class Edit Form

Course number and section are required.

Course Number:	<input type="text" value="EJB"/>
Section:	<input type="text" value="03"/>
Day:	<input type="text" value="TR"/>
Time:	<input type="text" value="5 : 30"/>
Room:	<input type="text" value="465"/>
Building:	<input type="text" value="CS"/>

[List classes](#)

- **html: taglib also from Struts**
- **There's no servlet API to assign request parameters to corre-**

**sponding bean properties, but JSP pages can do that**

**Send form page to JSP page to set values, then invoked bean method to perform action**

```

<html>
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
  <meta name="Author" content="Gerard Weatherby">
  <title>Class Edit Form</title>
</head>
<body>
<jsp:useBean id="classCommand" class="edu.rh.ejb.UniversityClassCommand"
scope="session"/>
<jsp:setProperty name="classCommand" property="*" />
<jsp:useBean id="proc" class="edu.rh.ejb.UnivClassProcessor" scope="session"/>
<jsp:setProperty name="proc" property="send" value="classCommand"/>

<center>
<h1>Class Edit Form</h1>
</center>

```

```

<!--<%= request.getParameterMap() %> for debugging -->
Course <jsp:getProperty name="classCommand" property="courseNumber"/>,
section <jsp:getProperty name="classCommand" property="section"/>
<logic:equal name="classCommand" property="cancel" value="true">
  cancelled.
</logic:equal>
<logic:equal name="classCommand" property="schedule" value="true">
Course <jsp:getProperty name="classCommand" property="courseNumber"/>,
section <jsp:getProperty name="classCommand" property="schedule"/>
  scheduled.
</logic:equal>
<p> <a href="class.jsp">Back to class edit form</a> <a href="list.jsp">List
classes</a> </body> </html>

```

- **logic: taglib also from Struts**
- **The *send* property of the *UnivClassProcessor* class is given the name of a session bean with the appropriate command.**
- ***UnivClassProcessor* is an *HttpSessionBindingListener* - it is notified when bound to a session.**

```

public class UnivClassProcessor implements HttpSessionBindingListener {
  private HttpSession session;
  public void valueBound(HttpSessionBindingEvent event) {
    session = event.getSession( );
  }
  public void valueUnbound(HttpSessionBindingEvent event) {
    session = null;
  }
  public void setSend(String beanName) throws ServletException {
    try {
      UniversityClassCommand cmd =
        (UniversityClassCommand)session.getAttribute(beanName);
      Context c = new InitialContext( );
      UniversityClassHome h =
        (UniversityClassHome)c.lookup("java:comp/env/ejb/Class");
      if (cmd.isCancel( )) {
        h.remove(cmd.getPK( ));
      } else if (cmd.isSchedule( ))
      {
        UniversityClassRemote uclass = h.create(cmd.getPK( ));

```

```

        UniversityClassInfo i = uclass.getInfo();
        i.setDay(cmd.getDay());
        i.setTime(cmd.getTime());
        i.setBuilding(cmd.getBuilding());
        i.setRoom(cmd.getRoom());
        uclass.setInfo(i);
    }
}
catch ...

```

- **UniversityClass beans sends messages when modified:**

```

private void createSender() {
    try {
        InitialContext c = new InitialContext();
        TopicConnectionFactory topicConnectionFactory = null;
        TopicConnection topicConnection = null;
        Topic topic = null;
        topicConnectionFactory = (TopicConnectionFactory)
            c.lookup("java:comp/env/jms/ClassTopicConnFactory");
        topic = (Topic)c.lookup("java:comp/env/jms/ClassTopic");
    }
}

```

```

        topicConnection = topicConnectionFactory.createTopicConnection();
        topicSession = topicConnection.createTopicSession(true, 0);
        topicPublisher = topicSession.createPublisher(topic);
    }
    catch (Exception e) {
        throw new EJBException("UniversityClassBean.createSender",e);
    }
}

/** send JMS message notifying clients of update */
private void notify(String action) {
    try {
        if (topicSession == null) {
            createSender();
        }
        UniversityClassPK pk = getPK();
        MapMessage message = topicSession.createMapMessage();
        //set header property
        message.setStringProperty("action",action);
        //can only set primitive types and wrappers, so break key down into strings
    }
}

```

```

        message.setString("course",pk.getCourseNumber());
        message.setString("section",pk.getSection());
        topicPublisher.publish(message);
        System.out.println("sent " + action + " " + pk);
    }
    catch (Exception e) {
        throw new EJBException("UniversityClassBean.notify",e);
    }
}

```

- **Note factory and connection names specified in deployment descriptor**

```

<resource-ref>
  <res-ref-name>jms/ClassTopicConnFactory</res-ref-name>
  <res-type>javax.jms.TopicConnectionFactory</res-type>
  <res-auth>Container</res-auth>
  <res-sharing-scope>Shareable</res-sharing-scope>
</resource-ref>
<resource-env-ref>
  <resource-env-ref-name>jms/ClassTopic</resource-env-ref-name>

```

```

  <resource-env-ref-type>javax.jms.Topic</resource-env-ref-type>
</resource-env-ref>

```

**Mapping of names to server topics / queues is vendor specific**

**In J2EE RI, found in sun-j2ee-ri.xml file.**

- **Message Driven Bean**

**If receives object message, process UniversityClassCommand**

**If receives map message, logs activity**

```

public class UniversityClassMessageBean extends MessageAdapter {
    public void onMessage(Message m) {
        try {
            if (m instanceof ObjectMessage) {
                ObjectMessage om = (ObjectMessage)m;
                UniversityClassCommand cmd =
                    (UniversityClassCommand)om.getObject();
            }
        }
    }
}

```

```

    if (cmd.isCancel()) {
        cancel(cmd);
    } else if (cmd.isSchedule()) {
        schedule(cmd);
    }
} else if (m instanceof MapMessage) {
    MapMessage mm = (MapMessage)m;
    System.out.println("MB: action " + mm.getStringProperty("action"));
    System.out.println("MB: courseNumber " + mm.getString("course"));
    System.out.println("MB: section " + mm.getString("section"));
}
} catch (Exception e) {
    throw new EJBException("UniversityClassBean",e);
}
}
}
private UniversityClassLocalHome getHome()
    throws NamingException {
    Context c = new InitialContext();
    UniversityClassLocalHome h =
        (UniversityClassLocalHome)c.lookup("java:comp/env/ejb/UnivClassLocal");

```

```

    return h; }

private void cancel(UniversityClassCommand cmd)
    throws NamingException, RemoveException {
    getHome().remove(cmd.getPK()); }

private void schedule(UniversityClassCommand cmd)
    throws NamingException, CreateException, RemoteException {
    UniversityClassLocal univ = getHome().create(cmd.getPK());
    univ.setDay(cmd.getDay());
    univ.setTime(cmd.getTime());
    univ.setRoom(cmd.getRoom());
    univ.setBuilding(cmd.getBuilding()); }
}

```

- **StudentClient can also monitor for changes.**

**See “Receiving message” on Message Driven Beans page 8**

