Developing an Entity
Enterprise Java Bean
Entity Bean Tutorial Agenda:

How to develop and deploy an entity bean by example:
- **What** is an entity bean
- **When** to use entity beans
- **Steps for implementing** an entity bean
  - Necessary interfaces and implementation classes
  - Overview of some APIs used in EJB: RMI, JNDI
  - EJB™ APIs
- **Steps for assembling and deploying** an enterprise bean
- **Steps for writing and running the client** of an enterprise bean
What Is an Entity Bean?

- **model business concepts** that can be expressed as nouns (i.e. Customer inventory item).
- **Model permanent data**: entity bean instance variables read from & stored in DB, gives in memory view and manipulation of data.
- Entity beans support **shared access** from multiple users.
- Always **transactional**.
- Entity beans can be **re-instantiated from attributes stored in database**:
  - Survives EJB™ server crashes.
  - Lives as long as the data in the database.

Entity Beans

An entity bean represents an object view of business data stored in persistent storage or an existing application. The bean provides an object wrapper around the data to simplify the task of accessing and manipulating it. This object interface lends itself to software reuse. For example, an entity bean representing user account information can be used by order management, user personalization, and marketing in a uniform way.

An entity bean allows shared access from multiple clients and can live past the duration of client’s session with the server. If the state of an entity bean is being updated by a transaction at the time of server crash, the entity bean’s state is automatically reset to the state of the last committed transaction.

The most common application for entity beans is for the **representation of data in a relational database**.

- A simple entity bean can be defined to represent a **row** in a database table, where each instance of the bean represents a specific row.
- **Complex** entity beans could represent **views of joined tables** in a database. For example an entity bean could represent a customer’s order and order items.

For example, the online bookstore application maintains information about customers, products, and orders using the entity beans **Customers, Products, and Orders**. The Customer bean provides customer details, such as name, address, credit card# of the customer. The Orders bean provides order details.
When to Use Entity Beans?

A Bean Provider can use the following entity bean characteristics as guidelines when deciding whether to model a business object as an entity bean:

- Representing persistent data
- Providing concurrent access by multiple clients
- Representing a single logical record (row) of data
- Providing robust, long-lived persistent data management

Guidelines for Using Entity Beans

A Bean Provider can use the following entity bean characteristics as guidelines when deciding whether to model a business object as an entity bean:

• Representing persistent data
  If the state of a business object needs to be stored in a persistent storage and its behavior primarily represents manipulation of data represented in its state, then it should be modeled as entity bean. However, it should be noted that every method call to an entity object via the remote and home interface is potentially a remote call. Even if the calling and called enterprise beans are located in the same Java virtual machine, the call must go through the container, which must create copies of all parameters that are passed through the interface by value. The container also checks security and applies declarative transaction attributes on the inter-component calls. Therefore modeling every object representing a row in the database as an entity bean is not recommended. An entity bean is better suited to represent a coarse-grained business object that provides more complex behavior than only get and set methods for its fields.

• Providing concurrent access by multiple clients
  When the state and behavior of a business object needs to be shared among multiple clients, they should be modeled as entity beans. This kind of business object needs to maintain state between method calls. However, this state is not specific to a particular client but is representative of persistent state of the business object, typically stored in a database. By modeling such business objects as entity beans, a Bean Provider can rely on an EJB server to ensure appropriate synchronization for entity beans as they are accessed concurrently from multiple transactions.

• Representing a single logical record (row) of data
  The business objects that typically operate on one logical record in the database are excellent candidates to model as entity beans. In fact, entity beans are designed to represent an individual (logical) record in the database. Entity beans provide methods to locate, create, and manipulate one row at a time.

• Providing robust, long-lived persistent data management
  A business object that needs to live after a client’s session with the server is over or that needs to be present when the server comes back after a crash, should be modeled as an entity bean. Entity beans live even after a client’s session with the server is over and can even survive
An entity bean usually represents a business object in a persistent storage mechanism.

EJB™ specifies two types of persistence:

• **container-managed persistence**: EJB™ container generates code to transfer data from the entity bean instance to the data store. The code is generated at deployment time, making CMP beans portable. The bean provider specifies the fields that need to be retained. Then, the container automatically performs all data retrieval and storage operations on behalf of the bean.

• **Bean-managed persistence**: bean providers write their own database access code. This does not mean that the developer should also do transaction management, the container can still take care of transactions.
How Do You Develop and Deploy an EJB?

A bean provider is concerned with the following 3 components:

1. **EJBHome interface**: plays role of "factory" object design pattern. Specifies methods to create, remove and find enterprise beans.

2. **EJBOBJECT interface**: plays role of "proxy" object design pattern. Specifies the business methods which are implemented in the bean.

3. **Bean implementation class**: implements the business logic.

The container uses the **EJBHome** interface to manage the life cycle of the enterprise bean. The EJB™ Home interface lists the methods for locating, creating, and removing instances of the EJB™ classes of the enterprise beans residing on the EJB™ container. The implementation of the EJB™ Home interface is the EJBHome object. The EJBHome object defines the methods that the client can use to create, find, and remove the enterprise bean from its container.

The **EJB™ object** performs the following tasks:

• It provides access to the business methods within the enterprise bean.

• It represents a client view of the enterprise bean.

• It exposes all the application-related interfaces for the object. However, it does not expose the interfaces that allow the EJB™ container to manage and control the object.

• It allows the EJB™ container to intercept all operations made on the enterprise bean.

When developing an entity bean, the bean developer needs to define the home and the remote interfaces that represent the client view of the bean. The bean developer also needs to create a class that implements the Entity Bean interface and the methods corresponding to the bean's home and the remote interfaces.
**EJB™ Home & EJB™ Object**

**EJBHome**: “factory” object design pattern. Provides methods to create, remove and find enterprise beans

**EJBObject**: “proxy” object design pattern. Provides the remote interface for the business methods which are implemented in the Enterprise Bean.

**Bean Implementation Class**: Implements the business method functionality.

A bean runs in a container. The container, which is part of the EJB™ server, provides transaction, synchronization, and security services to the bean.

To provide these services, the bean container/EJBObject intercepts all calls to bean methods.

For this reason, a client application does not call the remote bean methods directly. Instead, the client invokes the bean method through a two-step process, mediated by the Object Broker and by the container.

First, **the client calls a local proxy stub for the remote method**. The stub marshalls any parameter data, and then calls a remote skeleton on the server. The skeleton unmarshalls the data, and calls the bean container. This is transparent both to the client application developer and to the bean developer.

In the second step, the bean container gets the skeleton call, then **performs whatever services** are required by the context. These can include:

- authenticating the client
- performing transaction logic
- calling call back methods in the bean
- Bean instance activation/passivation

The container then delegates the method call to the bean. The bean method executes. When it returns, the thread of control returns to the bean container, which performs whatever services are required by the context. For example, if the method is running in a transaction context, the bean container performs a commit operation, depending on the transaction attributes in the bean descriptor.

Then the bean container calls the skeleton, which marshalls return data, and returns it to the client stub. These steps are invisible to client-side and server-side application developers.
In RMI, RPC, CORBA, DCOM the central connecting tissue between two distributed programs are the **client-side stub** and the **server-side skeleton**.

The stub is a piece of code that implements an interface to a remote object or service in the address space of a client of that service. The job of the stub is to open up a communication channel to the server, convert all the arguments to be sent to the server into a form that can be transmitted across the wire, and dispatch those converted arguments. The stub code then waits for the response from the server, converting any return values from their wire representation to the internal form used in the process, and handing those results back to the program or thread that made the call.

The skeleton code provides similar functionality on the side of the server. The skeleton code receives the information transmitted by a stub, converts the information that has been transmitted over the network into a form that can be understood by the server program, and makes the appropriate up-call to that server program. The server program will return any result values to the stub code, which will translate those results into a form that can be transmitted over the wire, and send them back to the calling client (where the stub code receives them, as outlined above).

The code for the stub and the skeleton is produced by a compiler that takes as input, a definition of the interface.
A Client makes a request on a remote object using a client-side stub. The stub marshalls the method arguments into serialized form and forwards the method request to the Object Request Broker. An Object Request Broker is an intermediary between distributed objects enabling them to communicate. ORBs are responsible for finding objects to service methods calls, handling parameter passing, and returning results. The ORB converts the client request into low-level transport requests (IIOP), then the request is sent over the network to the remote object.

On the Server side the ORB receives the transport level request and converts it into a request for the server skeleton for the remote object. The skeleton converts the remote request into the appropriate method call un-marshalling the method arguments and passing them to the remote object implementation.
RMI-IIOP combines RMI's ease of use with CORBA's interoperability and mission-critical infrastructure. Its RMI-centric programming model enables interoperability with, and migration from, existing RMI code. The ability to use IIOP and generate IDL allows seamless integration into a CORBA infrastructure. Its close association with EJB™ and its inclusion in the core Java 2 Platform will establish it as a foundation technology for enterprise middleware. By bringing Java and CORBA together as a strong and well-integrated combination, RMI-IIOP enhances the value of both technologies. This protocol offers developers access to the best of both worlds, and makes both worlds better.
Creating Distributed Applications Using RMI

Design and Implement the Remote Application Components

• Defining the remote interfaces: A remote interface specifies the methods that can be invoked remotely by a client. Clients program to remote interfaces, not to the implementation classes of those interfaces. Part of the design of such interfaces is the determination of any local objects that will be used as parameters and return values for these methods; if any of these interfaces or classes do not yet exist, you need to define them as well.

• Implementing the remote objects: Remote objects must implement one or more remote interfaces. The remote object class may include implementations of other interfaces (either local or remote) and other methods (which are available only locally). If any local classes are to be used as parameters or return values to any of these methods, they must be implemented as well.

Compile Sources and Generate Stubs

This is a two-step process. In the first step you use the javac compiler to compile the source files, which contain the implementation of the remote interfaces and implementations, the server classes, and the client classes. In the second step you use the rmic compiler to create stubs for the remote objects. RMI uses a remote object's stub class as a proxy in clients so that clients can communicate with a particular remote object.
This shows the development procedures for RMI-IIOP servers and clients. You will notice that they are almost the same as those of RMI (JRMP). Just as in RMI (JRMP), a distributed object's definition is its RMI Java interface. A difference is the -iiop parameter of the rmic compiler. This option is used to make rmic generate the stubs and tie that support the IIOP protocol. Without this -iiop option, rmic generates a stub and a skeleton for the JRMP protocol. Although the development procedure for RMI-IIOP is close to that for RMI (JRMP), the runtime environment is different in that communication is made through a CORBA 2.3-compliant ORB, using IIOP for communication between servers and clients.

There are some implementation differences when running over IIOP. Distributed garbage collection is not supported by CORBA, which uses explicit destruction and persistent object references with transparent passivation and activation. The RMI registry is replaced by JNDI with the CosNaming or LDAP service provider, and RMI activation is replaced by the portable object adapter. Remote object references must be downcast using a programmatic narrow() method instead of a direct Java language cast. Other RMI semantics, such as object serialization, are fully supported over IIOP.
Creating Distributed Applications Using RMI

Design and Implement the Remote Application Components

• **Defining the remote interfaces:** A remote interface specifies the methods that can be invoked remotely by a client. For EJB™ these are the EJBHome and EJBOBJECT interfaces

• **Implement the EJB.**

• **Compile Sources and Generate Stubs:**

This is a two-step process. In the first step you use the javac compiler to compile the source files, which contain the implementation of the remote interfaces and implementations, the server classes, and the client classes. In the second step you use the deployment tools to generate the EJBHome & EJBOBJECT implementations and the EJBHome & EJBOBJECT stubs and skeletons.
Demonstrates the purpose of the EJBHome “factory” and EJBOBJECT “proxy” remote interfaces and classes to provide remote access to the Enterprise bean.
Entity Bean Implementation

1. Create the **remote interface** for the bean.
2. Create the bean’s **home interface**.
3. Create the bean’s **implementation class**.
4. **Compile** the remote interface, home interface, and implementation class.
5. Create a **deployment descriptor**.
7. **Deploy** the enterprise application.
8. Create the **client**.
9. **Compile and run** the client.

What the **entity Bean Provider** is responsible for:

- Define the entity bean’s **remote interface** (Account). The remote interface defines the business methods callable by a client. The remote interface must extend the javax.ejb.EJBObject interface, and follow the standard rules for a RMI-IIOP remote interface. The remote interface must be defined as public.

- Define a **home interface** (AccountHome) for the entity bean. The home interface defines the entity bean’s specific create and finder methods. The home interface must be defined as public, extend the javax.ejb.EJBHome interface, and follow the standard rules for RMI-IIOP remote interfaces.

- Write the **business logic** in the entity bean class (AccountBean). The entity bean class may, but is not required to, implement the entity bean’s remote interface (Account). The entity bean must implement the methods of the javax.ejb.EntityBean interface, the ejbCreate(...) and ejbPost-Create(...) methods invoked at an entity object creation, and the finder methods (the finders should not have to be implemented if the entity bean uses container-managed persistence).

- Define a **deployment descriptor** that specifies any **declarative information** that the entity bean provider wishes to pass with the entity bean to the next stage of the development/deployment workflow.
Implement the Account Entity Bean

ATM

transfer()

Account

accountID

balance

withdraw()

deposit()

Checking

Account

Savings

Account

0…*

accesses

0…*
Implement the Account Entity Bean

```sql
create table account
(accountID varchar(3) constraint pk_account primary key,
balance decimal(10,2));
```

<table>
<thead>
<tr>
<th>accountID</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>41476</td>
<td>10000</td>
</tr>
<tr>
<td>333</td>
<td>20000</td>
</tr>
</tbody>
</table>
What the **container provider** is responsible for:

The tools provided by xxx Corporation are responsible for the following:

- **Generate** the entity EJBOBject class (**xxxRemoteAccount**) that **implements the entity bean’s remote interface**. The tools also generate the classes that implement the communication protocol specific artifacts for the remote interface.

- **Generate** the entity EJBHome class (**xxxAccountHome**) for the entity bean, that **implements the entity bean’s home interface** (**AccountHome**). The tools also generate the classes that implement the communication protocol specific artifacts for the home interface.

- Generate the implementation of the entity bean class suitable for the xxx container (**xxxAccountBean**). **xxxAccountBean** includes the business logic from the AccountBean class mixed with the services defined in the xxxBean class. xxx tools can use inheritance, delegation, and code generation to achieve mix-in of the two classes.

- **Generate** a class (**xxxAccountMetaData**) that implements the **javax.ejb.EJBMetaData** interface

The above classes and tools are container-specific (i.e., they reflect the way xxx Corp implemented them). Other container providers may use different mechanisms to produce their runtime classes, and the generated classes most likely will be different from those generated by xxx’s tools.
Enterprise JavaBeans specifies late binding of the EJBs to the underlying server. When the application is deployed, the EJB™ container generates container/server specific implementations of the Home and EJB™ object interfaces. Once these are generated, the deployment tool generates and compiles RMI or IIOP stubs and TIE skeletons for the Home and EJBOBJECT implementations using an rmic or rmi2iiop compiler.

The Server Object Broker runtime delivers the invocations to the skeleton. The skeleton unmarshals the parameters and delegates the invocation to the container generated implementation, which enforces the transaction, security, and state management rules of EJB™ and finally delivers the request to the bean provider written Enterprise JavaBean.
The **Remote** interface serves to identify interfaces whose methods may be invoked from a non-local virtual machine. Any object that is a remote object must directly or indirectly implement this interface. Only those methods specified in a "remote interface", an interface that extends `java.rmi.Remote` are available remotely.

The **EJBHome** interface is extended by all enterprise Bean's home interfaces. An enterprise Bean's home interface defines the methods that allow a client to create, find, and remove EJB™ objects.

Each enterprise Bean has a home interface. The home interface must extend the `javax.ejb.EJBHome` interface, and define the enterprise Bean type specific create and finder methods (session Beans do not have finders). The home interface is defined by the enterprise Bean provider and implemented by the enterprise Bean container.

**Method Summary:**

**EJBMetaData getEJBMetaData()**
- Obtain the EJBMetaData interface for the enterprise Bean. Used to access information about the bean. Mainly used by development tools

**HomeHandle getHomeHandle()**
- Obtain a handle for the home object.

Void **remove(HomeHandle handle)**
- Remove an EJB™ object identified by its handle.

Void **remove(java.lang.Object primaryKey)**
- Remove an EJB™ object identified by its primary key.

The **EJBObject** interface is extended by all enterprise Bean's remote interface. An enterprise Bean's remote interface provides the client's view of an EJB™ object. An enterprise Bean's remote interface defines the business methods callable by a client. Each enterprise Bean has a remote interface. The remote interface must extend the `javax.ejb.EJBObject` interface, and define the enterprise Bean specific business methods. The enterprise Bean's remote interface is defined by the enterprise Bean provider and implemented by the enterprise Bean container.

**Method Summary:**

**EJBHome getEJBHome()**
- Obtain the enterprise Bean's home interface.

**Handle getHandle()**
- Obtain a handle for the EJB™ object. The handle is a persistent reference to the the EJB™ Object and can be stored for later use.

**java.lang.Object getPrimaryKey()**
- Obtain the primary key of the EJB™ object.

**Boolean isIdentical(EJBObject obj)**
- Test if a given FIB™ object is identical to the invoked FIB™ object.
The **EnterpriseBean** interface must be implemented by every enterprise Bean class. It is a common super interface for the SessionBean and EntityBean interfaces.

The **EntityBean** interface is implemented by every entity enterprise Bean class. The container uses the EntityBean methods to notify the enterprise Bean instances of the instance’s life cycle events.

Method Summary:

- **void ejbActivate()**
  • A container invokes this method when the instance is taken out of the pool of available instances to become associated with a specific EJB™ object.

- **void ejbLoad()**
  • A container invokes this method to instruct the instance to synchronize its state by loading its state from the underlying database.

- **void ejbPassivate()**
  • A container invokes this method on an instance before the instance becomes disassociated with a specific EJB™ object.

- **void ejbRemove()**
  • A container invokes this method before it removes the EJB™ object that is currently associated with the instance.

- **void ejbStore()**
  • A container invokes this method to instruct the instance to synchronize its state by storing it to the underlying database.

**Void setEntityContext(EntityContext ctx)**
• Set the associated entity context.

**Void unsetEntityContext()**
• Unset the associated entity context.
public abstract interface **EJBContext**

The EJBContext interface provides an instance with access to the container-provided runtime context of an enterprise Bean instance. This interface is extended by the SessionContext and EntityContext interface to provide additional methods specific to the enterprise Bean type.

Method Summary

java.security.Principal getCallerPrincipal()
• Obtain the java.security.Principal that identifies the caller.

EJBHome getEJBHome()
• Obtain the enterprise bean’s home interface.

Boolean getRollbackOnly()
• Test if the transaction has been marked for rollback only.

UserTransaction getUserTransaction()
• Obtain the transaction demarcation interface.

Boolean isCallerInRole(java.lang.String roleName)
• Test if the caller has a given security role.

Void setRollbackOnly()
• Mark the current transaction for rollback.

public abstract interface **EntityContext** extends EJBContext

The EntityContext interface provides an instance with access to the container-provided runtime context of an entity enterprise Bean instance. The container passes the EntityContext interface to an entity enterprise Bean instance after the instance has been created. The EntityContext interface remains associated with the instance for the lifetime of the instance.

Method Summary

EJBObject getEJBObject()
• Obtain a reference to the EJB™ object that is currently associated with the instance.

java.lang.Object getPrimaryKey()
• Obtain the primary key of the EJB™ object that is currently associated with this instance.
<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Create the remote interface for the bean.</td>
</tr>
<tr>
<td>2.</td>
<td>Create the bean’s home interface.</td>
</tr>
<tr>
<td>3.</td>
<td>Create the bean’s implementation class.</td>
</tr>
<tr>
<td>4.</td>
<td>Compile the remote interface, home interface, and implementation class.</td>
</tr>
<tr>
<td>5.</td>
<td>Create a deployment descriptor.</td>
</tr>
<tr>
<td>7.</td>
<td>Deploy the enterprise application.</td>
</tr>
<tr>
<td>8.</td>
<td>Create the client.</td>
</tr>
<tr>
<td>9.</td>
<td>Compile and run the client.</td>
</tr>
</tbody>
</table>

What the **entity Bean Provider** is responsible for:

- Define the entity bean’s **remote interface** (Account). The remote interface defines the business methods callable by a client. The remote interface must extend the `javax.ejb.EJBObject` interface, and follow the standard rules for a RMI-IIOP remote interface. The remote interface must be defined as public.
Remote interface: “Proxy” design pattern.
An enterprise Bean's remote interface provides the client's view of an EJB™ object. An enterprise Bean's remote interface defines the business methods callable by a client.

- Business method declarations
- Exposes business methods to clients
- Extends javax.ejb.EJBObject interface
- All methods follow Java RMI rules

**Entity bean’s remote interface**
The following are the requirements for the entity bean’s remote interface:
- The interface must extend the javax.ejb.EJBObject interface.
- The methods defined in the remote interface must follow the rules for RMI-IIOP. This means that their arguments and return value types must be valid types for RMI-IIOP, and their throws clause must include the java.rmi.RemoteException.
- The remote interface is allowed to have superinterfaces. Use of interface inheritance is subject to the RMI-IIOP rules for the definition of remote interfaces.

For each method defined in the remote interface, there must be a matching method in the entity bean’s class. The matching method must have:
- The same name.
- The same number and types of its arguments, and the same return type.
- All the exceptions defined in the throws clause of the matching method of the enterprise Bean class must be defined in the throws clause of the method of the remote interface.
Entity Bean Implementation

1. Create the remote interface for the bean.
2. **Create the bean’s home interface.**
3. Create the bean’s implementation class.
4. Compile the remote interface, home interface, and implementation class.
5. Create a deployment descriptor.
7. Deploy the enterprise application.
8. Create the client.
9. Compile and run the client.

What the **entity Bean Provider** is responsible for:

- Define a **home interface** (AccountHome) for the entity bean. The home interface defines the entity bean’s specific create and finder methods. The home interface must be defined as public, extend the javax.ejb.EJBHome interface, and follow the standard rules for RMI-IIOP remote interfaces.
2) Create the Home Interface

```java
public interface AccountHome extends javax.ejb.EJBHome {
    Account create(int accountNumber, double startingBalance) throws java.rmi.RemoteException, javax.ejb.CreateException;
    Account findByPrimaryKey(Integer accountNumber) throws java.rmi.RemoteException, javax.ejb.FinderException;
}
```

The Home Interface “Factory” interface:

- Clients invoke this interface to create, or locate instances of the Bean.
- The Home Interface serves as the Factory interface for the Bean.
- The EJB™ server-specific implementation classes for the Bean Home Interface are generated at deployment time.

Requirements:
- Extends javax.ejb.EJBHome interface
- All methods follow Java RMI rules
- Create methods
- Finder methods
- Must provide findByPrimaryKey

The entity bean’s home interface allows a client to do the following:

- Create new entity objects within the home.
- Find existing entity objects within the home.
- Remove an entity object from the home.
- Get the javax.ejb.EJBMetaData interface for the entity bean. The javax.ejb.EJBMetaData interface is intended to allow application assembly tools to discover the meta-data information about the entity bean. The meta-data information allows loose client/server binding and scripting.
- Obtain a handle for the home interface. The home handle can be serialized and written to stable storage; later, possibly in a different JVM, the handle can be deserialized from stable storage and used to obtain a reference to the home interface. An entity bean’s home interface must extend the javax.ejb.EJBHome interface and follow the standard rules for Java programming language remote interfaces.
What the entity Bean Provider is responsible for:

- Write the business logic in the entity bean class (AccountBean). The entity bean must implement the methods of the javax.ejb.EntityBean interface, the ejbCreate(...) and ejbPost-Create(...) methods invoked at an entity object creation, and the finder methods (the finders should not have to be implemented if the entity bean uses container-managed persistence).
3) AccountBean Implementation

Entity Bean:
- Must implement javax.ejb.EntityBean interface
- Must provide implementation of
  - Remote interface business methods
  - Home ejbCreate, ejbPostCreate, ejbFindXXX methods

Entity Bean Persistence Programming Model:
- Enterprise JavaBeans provides a simple programming model for managing Entity Bean persistence.
- Persistence functions must be performed whenever Entity Beans are created or destroyed or whenever Entity Beans are loaded or evicted from memory.
- The Enterprise JavaBeans model isolates these functions in the ejbCreate(...), ejbRemove(), ejbLoad(), ejbStore(), ejbActivate(), and ejbPassivate() methods in the entity bean class. The container calls these call back methods to alert the bean about state change events.
Life Cycle

Stage 1 - The container instantiates the enterprise bean.
Stage 2 - The container moves the bean between a pool of available instances and the ready state.
Stage 3 - The container removes the bean.

Although the client calls methods to create and remove an enterprise bean, it is actually the container that performs these tasks and makes these operations transparent to the client.
Entity Bean Business Method OID

client → EJBHome → EJBOBJECT → container → bean instance → database

business method

business method
3) AccountBean: Implement Account Interface Business Methods

```java
public class AccountBean implements EntityBean {
    // entity state
    private double balance;
    private int accountNumber;
    // resources
    private EntityContext entityContext;

    // implement account interface business methods
    public void deposit(double amount) {
        balance = balance + amount;
    }
    public void withdraw(double amount) throws InsufficientFundsException {
        if (balance >= amount) {
            balance = balance - amount;
        } else {
            throw new InsufficientFundsException(balance);
        }
    }
    public double getBalance() {
        return balance;
    }
}
```

Bean business-logic methods:

These methods must match the methods defined in this EJB's remote interface, Account. Although AccountBean does not implement the Account interface, generated stub classes on the client side will. Calls on those classes will be forwarded to the AccountBean.

Business methods

The entity bean class may define zero or more business methods whose signatures must follow these rules:

- The method names can be arbitrary, but they must not start with 'ejb' to avoid conflicts with the callback methods used by the EJB™ architecture.
- The business method must be declared as public.
- The method must not be declared as final or static.
- The method’s arguments and return value types must be legal types for RMI-IIOP.
- The throws clause may define arbitrary application specific exceptions.

Compatibility Note: EJB™ 1.0 allowed the business methods to throw the java.rmi.RemoteException to indicate a non-application exception. This practice is deprecated in EJB™ 1.1—an EJB™ 1.1 compliant enterprise bean should throw the javax.ejb.EJBException or another java.lang.RuntimeException to indicate non-application exceptions to the Container.
**ejbCreate, ejbPostCreate**

The container invokes these two methods during the creation of an entity object as a result of a client invoking a create(...) method on the entity bean’s home interface. The container first invokes the ejbCreate(...) method whose signature matches the create(...) method invoked by the client. **With bean-managed persistence, the entity Bean Provider is responsible for writing the code that inserts a record into the database in the ejbCreate(...) methods.** The ejbCreate(...) method returns a primary key for the created entity object. The container creates an entity EJBObject reference for the primary key. The container then invokes a matching ejbPostCreate(...) method to allow the instance to fully initialize itself. Finally, the container returns the entity object’s remote interface (i.e. a reference to the entity EJBObject) to the client.
import java.sql.*

  public void exampleJDBC()
  {
    InitialContext context = new InitialContext();
    DataSource dataSource = (DataSource) context.lookup(dataSourceName);
    connection = dataSource.getConnection(user, password);
    Statement stmt = conn.createStatement();
    ResultSet rs = stmt.executeQuery("select ename, empno, sal from emp");
    while(rs.next())
    {
      String name = rs.getString(1);
      int number = rs.getInt(2);
      double salary = rs.getDouble(3);
      conn.close();
    }
  }
Get the Resource Factory for a Database Connection:

Example EJB™ Code to get the resource factory for a Database connection:

```java
// obtain the initial JNDI context
Context initCtx = new InitialContext();

// perform JNDI lookup to obtain resource factory
javax.sql.DataSource ds = (javax.sql.DataSource) initCtx.lookup("java:comp/env/jdbc/AccountDB");

// Invoke factory to obtain a connection.
java.sql.Connection con = ds.getConnection();
```

The deployer maps the coded name jdbc/AccountDB to JNDI names like jdbc/Cloudscape which the J2EE™ server automatically enters in the JNDI name space.
3) AccountBean: Create Method with BMP

```java
public Integer ejbCreate (int accountNumber, double startingBalance) throws CreateException {
    try {
        Statement stmt = con.createStatement();
        // insert a record representing the entity in DB
        stmt.executeUpdate("INSERT INTO ACCOUNT …");
        stmt.close();
    } catch (SQLException ex) {
        throw new CreateException();
    }

    // initialize instance variables
    this.balance = startingBalance;
    this.accountId = accountNumber;

    // return primary key
    return new Integer(accountNumber);
}
```

ejbCreate

- returns primary key type
- matches signatures of create methods in Home interface
- inserts a record representing the entity into the database
- initializes instance variables

This method must match the create method defined in this EJB’s home interface, AccountHome. Container tools generate the implementation of the AccountHome class and the client stub and server skeleton classes. Calls on those classes will be forwarded to the AccountBean.

When the client calls AccountHome.create() the container allocates an instance of the AccountBean and calls AccountBean.ejbCreate().

```java
public PrimaryKeyClass ejbCreate(...);```

There are 0 or more ejbCreate(...) methods, whose signatures match the signatures of the create(...) methods of the entity bean home interface. The container invokes an ejbCreate(...) method on an entity bean instance when a client invokes a matching create(...) method to create an entity object.

The implementation of the ejbCreate(...) method typically validates the client-supplied arguments, and inserts a record representing the entity object into the database. The method also initializes the instance’s variables. The ejbCreate(...) method must return the primary key for the created entity object.
ejbCreate, ejbPostCreate

With container-managed persistence, the **container performs the database insert after the ejbCreate(...) method completes.**

The entity Bean Provider’s responsibility is to **initialize the container-managed fields in the ejbCreate(...) methods from the input arguments such that when an ejbCreate(...) method returns,**

**the container can extract the container-managed fields from the instance and insert them into the database.**

The ejbCreate(...) methods must be defined to **return the primary key class type.** The implementation of the ejbCreate(...) methods should be coded to return a null. The returned value is ignored by the Container.

The container is responsible for creating the entity object’s representation in the underlying database, extracting the primary key fields of the newly created entity object representation in the database, and for creating an entity EJBOBJECT reference for the newly created entity object. The Container must establish the primary key before it invokes the ejbPostCreate(...) method.
3) AccountBean: Create Method with CMP

```java
// must match signatures of create methods in Home interface

public Integer ejbCreate (int accountNumber, double startingBalance) throws CreateException {
    if (accountNumber == null) {
        throw new CreateException("The accountNumber is required.");
    }
    this.balance = startingBalance;
    this.accountId = accountNumber;
    return null;
}
```

ejbCreate
- matches signatures of create methods in Home interface
- Container inserts a record representing the entity into the database
- initializes instance variables

With container-managed persistence, the **container performs the database insert after the ejbCreate(...) method completes.**

The entity Bean Provider’s responsibility is to **initialize the container-managed fields in the ejbCreate(...) methods from the input arguments such that when an ejbCreate(...) method returns,**

the **container can extract the container-managed fields from the instance and insert them into the database.**

The ejbCreate(...) methods must be **defined to return the primary key class type.** The implementation of the ejbCreate(...) methods should be **coded to return a null.** The returned value is ignored by the Container.

The container is responsible for creating the entity object’s representation in the underlying database, extracting the primary key fields of the newly created entity object representation in the database, and for creating an entity EJBObject reference for the newly created entity object. The Container must establish the primary key before it invokes the ejbPostCreate(...) method.
public primary key type or collection ejbFind<METHOD>(...);
The container invokes this method on the instance when the container selects
the instance to execute a matching client-invoked find<METHOD>(...) method. The instance is in the pooled state (i.e. it is not assigned to any
particular entity object identity) when the container selects the instance to
execute the ejbFind<METHOD> method on it, and it is returned to the pooled
state when the execution of the ejbFind<METHOD> method completes.
The implementation of an ejbFind<METHOD> method typically uses the
method’s arguments to locate the requested entity object or a collection of
entity objects in the database. The method must return a primary key or a
collection of primary keys to the container.
3) AccountBean: Find Method for BMP

```java
//returns void and must match each ejbCreate method
public void ejbPostCreate (int myAccount, double balance){ }
//must match each find method in Home interface
public Integer ejbFindByPrimaryKey (Integer key) throws FinderException {
    try {
        Statement stmt = con.createStatement();
        // find entity in database
        ResultSet rs = stmt.executeQuery("SELECT ACCNTNUM ...");
        if (rs.next()) { // record exists?
            stmt.close();
            // return primary key
            return new Integer(accountNumber);
        } else {
            stmt.close();
            return null;
        }
    } catch (SQLException ex) {
        throw new FinderException();
    }
}
```

- **ejbPostCreate**
  - called after ejbCreate
  - EJB™ object identity is available (via EntityContext.getPrimaryKey)
- **ejbFindXXX**
  - matches signature of finder methods in Home interface
  - returns primary key type or collection
  - locates one or more entity objects in the database
  - Must define ejbFindByPrimaryKey method

```java
public primary key type or collection ejbFind<METHOD>(...);
```

For each ejbCreate(...) method, there is a matching ejbPostCreate(...) method that has the same input parameters but the return value is void. The container invokes the matching ejbPostCreate(...) method on an instance after it invokes the ejbCreate(...) method with the same arguments. The entity object identity is available during the ejbPostCreate(...) method. The instance may, for example, obtain the remote interface of the associated entity object and pass it to another enterprise bean as a method argument. An ejbPostCreate(...) method executes in the same transaction context as the previous ejbCreate(...) method.

```java
public primary key type or collection ejbFind< METHOD > ( ... );
```

The signatures of the finder methods must follow the following rules:
- A finder method name must start with the prefix “ejbFind” (A finder method must be declared as public.
- The method must not be declared as final or static.
- The methods’ arguments’ types must be legal types for RMI-IIOP.
- The return type of a finder method must be the entity bean’s primary key type, or a collection of primary keys
- The throws clause may define arbitrary application specific exceptions, including the javax.ejb.FinderException.
- Every entity bean must define the ejbFindByPrimaryKey method. The result type for this method must be the primary key type.
CMP finder methods

The entity Bean Provider does not write the finder (ejbFind<METHOD>(...)) methods. The finder methods are generated at the entity bean deployment time using the container provider’s tools. The tools can, for example, create a subclass of the entity bean class that implements the ejbFind<METHOD>() methods, or the tools can generate the implementation of the finder methods directly in the class that implements the entity bean’s home interface. Note that the ejbFind<METHOD> names and parameter signatures do not provide the container tools with sufficient information for automatically generating the implementation of the finder methods for methods other than ejbFindByPrimaryKey. Therefore, the bean provider is responsible for providing a description of each finder method. The entity bean Deployer uses container tools to generate the implementation of the finder methods based in the description supplied by the bean provider. The Enterprise JavaBeans architecture does not specify the format of the finder method description.
3) **AccountBean: Find Methods for CMP**

- Entity Beans with Container Managed Persistence do not implement the Find Methods.
- These methods are generated by the Container at deployment time.

- **ejbPostCreate**
  - called after ejbCreate
  - EJB™ object identity is available (via EntityContext.getPrimaryKey)

- **ejbFindXXX**
  - matches signature of finder methods in Home interface
  - returns primary key type or collection
  - locates one or more entity objects in the database
  - Must define ejbFindByPrimaryKey method

```java
public void ejbPostCreate (...);
```

For each ejbCreate(...) method, there is a matching ejbPostCreate(...) method that has the same input parameters but the return value is void. The container invokes the matching ejbPostCreate(...) method on an instance after it invokes the ejbCreate(...) method with the same arguments. The entity object identity is available during the ejbPostCreate(...) method. The instance may, for example, obtain the remote interface of the associated entity object and pass it to another enterprise bean as a method argument. An ejbPostCreate(...) method executes in the same transaction context as the previous ejbCreate(...) method.

```java
public primary key type or collection ejbFind<METHOD>(...);
```

The signatures of the finder methods must follow the following rules:

- A finder method name must start with the prefix “ejbFind” (A finder method must be declared as public.
- The method must not be declared as final or static.
- The methods’ arguments’ types must be legal types for RMI-IIOP.
- The return type of a finder method must be the entity bean’s primary key type, or a collection of primary keys
- The throws clause may define arbitrary application specific exceptions, including the javax.ejb.FinderException.
- Every entity bean must define the ejbFindByPrimaryKey method. The result type for this method must be the primary key type.
• When the container invokes the `ejbStore` method on the instance, the instance must **push all cached updates of the entity object’s state to the underlying database**. The container invokes the `ejbStore` method at the end of a transaction, and may also invoke it at other times when the instance is in the ready state. (For example the container may invoke `ejbStore` when passivating an instance in the middle of a transaction, or when transferring the instance’s state to another instance to support distributed transactions in a multi-process server.)

• When the container invokes the `ejbLoad` method on the instance, the instance must discard any cached entity object’s state. The instance may, but is not required to, **refresh the cached state by reloading it from the underlying database**. The following examples, which are illustrative but not prescriptive, show how an instance may cache the entity object’s state:

• An instance loads the entire entity object’s state in the `ejbLoad` method and caches it until the container invokes the `ejbStore` method. The business methods read and write the cached entity state. The `ejbStore` method writes the updated parts of the entity object’s state to the database.

• An instance loads the most frequently used part of the entity object’s state in the `ejbLoad` method and caches it until the container invokes the `ejbStore` method. Additional parts of the entity object’s state are loaded as needed by the business methods. The `ejbStore` method writes the updated parts of the entity object’s state to the database.

• An instance does not cache any entity object’s state between business methods. The business methods access and modify the entity object’s state directly in the database. The `ejbLoad` and `ejbStore` methods have an empty implementation.
Bean callback methods:
The container uses these to alert the bean of runtime events. Each method is called at a specific time during the life cycle of a bean.

**setEntityContext**
- Container passes reference to EntityContext
- Bean allocates resources held for the bean's lifetime

**unsetEntityContext**
- Bean releases resources allocated in setEntityContext

```java
public void setEntityContext(EntityContext ec) {
    // Information about the Entity instance is in EntityContext
    this.context = ec;
    try {
        makeConnection();
    } catch (Exception ex) {
        throw new EJBException("Unable to connect to database. " +
                                ex.getMessage());
    }
}
```

```java
public void unsetEntityContext() {
    try {
        con.close();
    } catch (SQLException ex) {
        throw new EJBException("unsetEntityContext: " + ex.getMessage());
    }
}
```

A container uses this method to pass a reference to the EntityContext interface to the entity bean instance. If the entity bean instance needs to use the EntityContext interface during its lifetime, it must **remember the EntityContext interface in an instance variable**. The instance can take advantage of the setEntityContext() method to **allocate any resources that are to be held by the instance for its lifetime**.

```java
public void setEntityContext(EntityContext ec);
```

A container invokes this method before terminating the life of the instance. This method executes with an unspecified transaction context. An identity of an entity object is not available during this method. The instance can take advantage of the unsetEntityContext() method to **free any resources that are held by the instance**. (These resources typically had been allocated by the setEntityContext() method.)
3) AccountBean: Implement Entity Interface
Container Callback Methods

```
//container associates EJB™ object identity with an instance
//bean allocates additional resources needed in ready state
public void ejbActivate() {
    id = (String)context.getPrimaryKey();
}

//container dissociates identity from instance bean
//frees resources allocated in ejbActivate
public void ejbPassivate() {
    id = null;
}
```

The container uses these methods to alert the bean of swapping the bean to and from the “ready state” to the “pooled state”.

**public void ejbActivate();**

The container invokes this method on the instance when the container picks the instance from the pool and assigns it to a specific entity object identity. The ejbActivate() method gives the entity bean instance the chance to acquire additional resources that it needs while it is in the ready state.

This method executes with an unspecified transaction context. The instance can obtain the identity of the entity object via the getPrimaryKey() or getEJBObject() method on the entity context. The instance can rely on the fact that the primary key and entity object identity will remain associated with the instance until the completion of ejbPassivate() or ejbRemove().

**public void ejbPassivate();**

The container invokes this method on an instance when the container decides to disassociate the instance from an entity object identity, and to put the instance back into the pool of available instances. The ejbPassivate() method gives the instance the chance to release any resources that should not be held while the instance is in the pool. (These resources typically had been allocated during the ejbActivate() method.)
3) AccountBean: Implement Entity Interface
Container Callback Methods

//synchronizes entity state in data store with instance
variables

public void ejbStore() {
    //update the data this instance represents in the
    //database, keyed by this EntityBean's primary key
    try {
        Statement stmt = con.createStatement();
        stmt.executeUpdate("UPDATE ACCOUNT SET balance=... WHERE ID=");
    } catch (SQLException ex) {
        throw new RemoteException();
    }
}

• ejbStore
  • synchronizes entity state in data store with instance variables
  • writes states to database

public void ejbStore();

The container invokes this method on an instance to inform the instance that
the instance must synchronize the entity state in the database with the entity
state cached in its instance variables. The instance must be prepared for the
container to invoke this method at any time that the instance is in the ready
state. An instance must write any updates cached in the instance variables
to the database in the ejbStore() method.
public void ejbLoad() {
    //retrieve the data this instance represents from the
    //database, keyed by this EntityBean’s primary key

    // retrieve state from DB
    try {
        Statement stmt = con.createStatement();
        ResultSet rs = stmt.executeQuery
            ("SELECT … ACCOUNT WHERE ACCNTNUM = …");
        if (rs.next()) {
            balance = rs.getDouble(1);
        } else {
            throw new RemoteException();
        }
        } catch (SQLException ex) {
            throw new RemoteException();
        }
    }
}
**Passivation of an Entity Object With Container Managed Persistence**

When the container needs to synchronize the state of an enterprise bean instance with the entity object’s state in the database, the `container reads the entity object’s state from the database` into the container managed fields and then it invokes the `ejbLoad()` method on the instance. The entity Bean Provider can rely on the container’s having loaded the container-managed fields from the database just before the container invokes the `ejbLoad()` method. The entity bean can use the `ejbLoad()` method, for instance, to perform some computation on the values of the fields that were read by the container (for example, uncompressing text fields).

**ejbStore**

When the container needs to synchronize the state of the entity object in the database with the state of the enterprise bean instance, the container first calls the `ejbStore()` method on the instance, and then it extracts the container-managed fields and writes them to the database.
• ejbLoad
  • synchronizes instance variables from entity state in database
  • refreshes states from database

public void ejbLoad();

The container invokes this method on an instance in the ready state to inform
the instance that it must synchronize the entity state cached in its instance
variables from the entity state in the database. The instance must be prepared
for the container to invoke this method at any time that the instance is in the
ready state.

If the instance is caching the entity state (or parts of the entity state), the
instance must not use the previously cached state in the subsequent business
method. The instance may take advantage of the ejbLoad method, for example,
to refresh the cached state by reading it from the database.

This method executes in the transaction context determined by the transaction
attribute of the business method that triggered the ejbLoad method.
Removal of an Entity Bean With BMP

- client
- EJBHome
- EJBOBJECT
- container
- bean instance
- database

remove

remove

ejbRemove()

Remove in DB

© Copyright 2000 Sun Microsystems, Inc. All rights reserved.
ejbRemove

The container invokes the `ejbRemove()` method on an entity bean instance with container-managed persistence in response to a **client-invoked remove operation on the entity bean’s home or remote interface.** The entity Bean Provider can use the `ejbRemove` method to implement any actions that must be done before the entity object’s representation is removed from the database. The container synchronizes the instance’s state before it invokes the `ejbRemove` method. This means that the state of the instance variables at the beginning of the `ejbRemove` method is the same as it would be at the beginning of a business method. After `ejbRemove` returns, the **container removes the entity object’s representation from the database.** The container must perform `ejbRemove` and the database delete operation in the transaction context determined by the transaction attribute of the invoked remove method, as described in subsection.
3) AccountBean: Implement Entity Interface
Container Callback Methods

//removes entity representation in the database
//releases resources allocated in ejbActivate

public void ejbRemove() throws RemoveException {
    try {
        Statement stmt = con.createStatement();
        stmt.executeUpdate
            ("DELETE FROM ACCOUNT WHERE ACCNTNUM = ...");
        stmt.close();
        con.close();  // free JDBC connection
    } catch (SQLException ex) {
        throw new RemoteException();
    }
}

• ejbRemove
  • removes entity representation in the database
  • releases resources allocated in ejbActivate

public void ejbRemove();

The container invokes this method on an instance as a result of a client’s invoking a remove method. The instance is in the ready state when ejbRemove() is invoked and it will be entered into the pool when the method completes.

This method executes in the transaction context determined by the transaction attribute of the remove method that triggered the ejbRemove method. The instance can still obtain the identity of the entity object via the getPrimaryKey() or getEJBObject() method of the EntityContext interface. The container synchronizes the instance’s state before it invokes the ejbRemove method. This means that the state of the instance variables at the beginning of the ejbRemove method is the same as it would be at the beginning of a business method. **An entity bean instance should use this method to remove the entity object’s representation in the database.** Since the instance will be entered into the pool, the state of the instance at the end of this method must be equivalent to the state of a passivated instance. This means that the instance must release any resource that it would normally release in the ejbPassivate() method.
Lifecycle of an Entity Bean

- does not exist
- newInstance()
- setEntityContext()
- unsetEntityContext()
- ejbFind()
- ejbCreate() eejbPostCreate()
- ejbActivate() ejbPassivate()
- ejbRemove()
- ejbLoad()
- ejbStore()

EJB Instance

EJB Object
Example of how the Netscape Application Server handles Concurrent Entity Access
EJBs: NAS Entity Beans w/BMP

- Isolation levels are set on a DB connection level
- Isolation level available depends on DB backend.
- Pessimistic vs. optimistic is developer's choice.
- Commit Option: datastore-style locking
- ejbLoad() called at start of txn
- ejbStore() called when txn commits

How the Netscape Application Server handles transactions with Entity Beans with BMP
Entity Bean Implementation

1. Create the remote interface for the bean.
2. Create the bean's home interface.
3. Create the bean's implementation class.
4. **Compile the remote interface, home interface, and implementation class.**
5. Create a deployment descriptor.
7. Deploy the enterprise application.
8. Create the client.
9. Compile and run the client.
4) Compile the Remote & Home Interfaces and Implementation Class.

javac -classpath $J2EE_HOME/lib/j2ee.jar
Account.java AccountHome.java
AccountEJB.java
Entity Bean Implementation

1. Create the remote interface for the bean.
2. Create the bean’s home interface.
3. Create the bean’s implementation class.
4. Compile the remote interface, home interface, and implementation class.
5. **Create a deployment descriptor.**
7. Deploy the enterprise application.
8. Create the client.
9. Compile and run the client.

What the **entity Bean Provider** is responsible for:

- Define a **deployment descriptor** that specifies any **declarative information** that the entity bean provider wishes to pass with the entity bean to the next stage of the development/deployment workflow.
The Deployment Descriptor is used to establish the runtime service settings for an enterprise bean. These settings guide the EJB™ container for managing and controlling the enterprise bean.

Declare how the container should control persistence, state management, transaction coordination, and security.
The Deployment Descriptor allows the user to specify the Bean’s transactional and security attributes declaratively, simplifying the process of building transactional applications. In addition to these attributes being specified at the bean level, the user also has the ability to specify finer grained transaction and security at the method level.
Deployment Descriptor

Puts together all the elements of the bean:
• Enterprise bean’s class.
• Home and remote interfaces.

Lists the bean properties:
• Bean type: session or entity.
• Session bean’s state management type: stateful or stateless.
• Session bean’s transaction demarcation type: bean or container managed.
• Entity bean’s persistence management type: enterprise bean or container.

The Bean Provider must provide the following information for each enterprise bean:
• Enterprise bean’s name. The Bean Provider must assign a logical name to each enterprise bean in the ejb-jar file. There is no architected relationship between this name, and the JNDI name that the Deployer will assign to the enterprise bean. The Bean Provider specifies the enterprise bean’s name in the ejb-name element.
• Enterprise bean’s class. The Bean Provider must specify the fully-qualified name of the Java class that implements the enterprise bean’s business methods. The Bean Provider specifies the enterprise bean’s class name in the ejb-class element.
• Enterprise bean’s home interfaces. The Bean Provider must specify the fully-qualified name of the enterprise bean’s home interface in the home element.
• Enterprise bean’s remote interfaces. The Bean Provider must specify the fully-qualified name of the enterprise bean’s remote interface in the remote element.
• Enterprise bean’s type. The enterprise beans types are session and entity. The Bean Provider must use the appropriate session or entity element to declare the enterprise bean’s structural information.
• Session bean’s state management type. If the enterprise bean is a Session bean, the Bean Provider must use the session-type element to declare whether the session bean is stateful or stateless.
• Session bean’s transaction demarcation type. If the enterprise bean is a Session bean, the Bean Provider must use the transaction-type element to declare whether transaction demarcation is performed by the enterprise bean or by the Container.
Lists the bean properties continued:

- Entity bean’s **primary key** class.
- **Container-managed fields**.
- **Environment** entries.
- **Resource factory** references.
- **EJB™ references**.
- **Security role references**.
- **Transaction attributes**.

Continued:

- **Entity bean’s persistence management**. If the enterprise bean is an Entity bean, the Bean Provider must use the persistence-type element to declare whether persistence management is performed by the enterprise **bean** or by the **Container**.

- **Entity bean’s primary key class**. If the enterprise bean is an Entity bean, the Bean Provider specifies the fully-qualified name of the Entity bean’s primary key class in the prim-key-class element. The Bean Provider must specify the primary key class for an Entity with bean-managed persistence, and may (but is not required to) specify the primary key class for an Entity with container-managed persistence.

- **Container-managed fields**. If the enterprise bean is an Entity bean with container-managed persistence, the Bean Provider must specify the container-managed fields using the cmp-fields elements.

- **Environment entries**. The Bean Provider must declare all the enterprise bean’s environment entries.

- **Resource factory** references. The Bean Provider must declare all the enterprise bean’s resource factory references.

- **EJB™ references**. The Bean Provider must declare all the enterprise bean’s references to the homes of other enterprise beans.

- **Security role references**. The Bean Provider must declare all the enterprise bean’s references to security roles.
**Account EJB™ Jar File contents:**

- **Enterprise bean’s class.** The Bean Provider must specify the fully-qualified name of the Java class that implements the enterprise bean’s business methods. The Bean Provider specifies the enterprise bean’s class name in the ejb-class element.

- **Enterprise bean’s home interfaces.** The Bean Provider must specify the fully-qualified name of the enterprise bean’s home interface in the home element.

- **Enterprise bean’s remote interfaces.** The Bean Provider must specify the fully-qualified name of the enterprise bean’s remote interface in the remote element.

- **Enterprise bean’s type.** The enterprise beans types are **session** and **entity.** The Bean Provider must use the appropriate session or entity element to declare the enterprise bean’s structural information.
Enterprise bean's class, Home and remote Interfaces, Bean type
Entity bean’s **persistence management type**, Entity bean’s **primary key class**: BMP

- **Entity bean’s persistence management.** If the enterprise bean is an Entity bean, the Bean Provider must use the persistence-type element to declare whether persistence management is performed by the enterprise **bean** or by the **Container**.
• Entity bean’s **persistence management**. If the enterprise bean is an Entity bean, the Bean Provider must use the persistence-type element to declare whether persistence management is performed by the enterprise **bean** or by the **Container**.
Environment entries. The Bean Provider must declare all the enterprise bean’s environment entries.

Example EJB™ Code to get the environment entry values:
Context initial = new InitialContext();
Context environment = (Context) initial.lookup("java:comp/env");
String username = (String) environment.lookup("user");
String password = (String) environment.lookup("password");
Resource factory references

The Bean Provider must declare all the enterprise bean’s resource factory references.

Example EJB™ Code to get the resource factory for a Database connection:

```java
// obtain the initial JNDI context
Context initCtx = new InitialContext();
// perform JNDI lookup to obtain resource factory
javax.sql.DataSource ds = (javax.sql.DataSource)initCtx.lookup("java:comp/env/jdbc/AccountDB");
// Invoke factory to obtain a connection.
java.sql.Connection con = ds.getConnection(user,password);
...
```

Resource factory references

A resource factory is an object that is used to create connections to a resource manager. For example, an object that implements the javax.sql.DataSource interface is a resource factory for java.sql.Connection objects which implement connections to a database management system. The deployment descriptor allow the enterprise bean code to refer to resource factories using logical names called resource factory references. The resource factory references are special entries in the enterprise bean’s environment. The Deployer binds the resource factory references to the actual resource factories that are configured in the Container. Because these resource factories allow the Container to affect resource management, the connections acquired through the resource factory references are called managed resources (e.g., these resource factories allow the Container to implement connection pooling and automatic enlistment of the connection with a transaction).
Get the Resource Factory for a Database Connection:

Example EJB™ Code to get the resource factory for a Database connection:

```java
//perform JNDI lookup to obtain resource factory
javax.sql.DataSource ds = (javax.sql.DataSource)initCtx.lookup("java:comp/env/jdbc/AccountDB");

// Invoke factory to obtain a connection.
java.sql.Connection con = ds.getConnection();
```

```xml
<resource-ref>
  <description>no description</description>
  <res-ref-name>jdbc/AccountDB</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Container</res-auth>
</resource-ref>
```
Transaction attributes: The Bean Provider may define the value of the transaction attributes for the methods of the remote and home interfaces of the enterprise beans that require container-managed transaction demarcation. All Entity beans and the Session beans declared by the Bean Provider as transaction-type Container require container-managed transaction demarcation. The Application Assembler uses the container-transaction elements to declare the transaction attributes.

The trans-attribute element specifies how the container must manage the transaction boundaries when delegating a method invocation to an enterprise bean’s business method.

The value of trans-attribute must be one of the following:

- `<trans-attribute>NotSupported</trans-attribute>`
- `<trans-attribute>Supports</trans-attribute>`
- `<trans-attribute>Required</trans-attribute>`
- `<trans-attribute>RequiresNew</trans-attribute>`
- `<trans-attribute>Mandatory</trans-attribute>`
- `<trans-attribute>Never</trans-attribute>`
Deployment Descriptor

- Container tools generate the deployment descriptor XML.
- GUI Container tools aid the bean provider to set transaction attributes, security, environment...references.
- Container tools aid the application assembler to specify application assembly information.
- Container tools allow the deployer to set or modify/configure values for the deployed environment.

The Deployment Descriptor is generated and set/modified using container tools.
5) Deployment Descriptor XML

```xml
<ejb-jar>
  <description>no description</description>
  <display-name>AccountEJB</display-name>
  <enterprise-beans>
    <entity>
      <description>no description</description>
      <display-name>AccountEJB</display-name>
      <home>AccountHome</home>
      <remote>Account</remote>
      <ejb-name>AccountEJB</ejb-name>
      <home>AccountHome</home>
      <remote>Account</remote>
      <ejb-class>AccountEJB</ejb-class>
      <persistence-type>Bean</persistence-type>
      <primary-class>java.lang.String</primary-class>
      <reentrant>False</reentrant>
      <resource-ref>
        <description>no description</description>
        <res-ref-name>jdbc/AccountDB</res-ref-name>
        <res-type>javax.sql.DataSource</res-type>
        <res-auth>Container</res-auth>
      </resource-ref>
    </entity>
  </enterprise-beans>
</ejb-jar>
```
5) DD XML Cont.

```
<assembly-descriptor>
  <container-transaction>
    <method>
      <ejb-name>AccountEJB</ejb-name>
      <method-intf>Remote</method-intf>
      <method-name>debit</method-name>
      <method-param>double</method-param>
    </method>
    <trans-attribute>Required</trans-attribute>
  </container-transaction>
  <container-transaction>
    <method>
      <ejb-name>AccountEJB</ejb-name>
      <method-intf>Remote</method-intf>
      <method-name>credit</method-name>
      <method-param>double</method-param>
    </method>
    <trans-attribute>Required</trans-attribute>
  </container-transaction>
</assembly-descriptor>
```
Entity Bean Implementation

1. Create the remote interface for the bean.
2. Create the bean’s home interface.
3. Create the bean’s implementation class.
4. Compile the remote interface, home interface, and implementation class.
5. Create a deployment descriptor.
6. **Package in an ejb-jar file.**
7. Deploy the enterprise application.
8. Create the client.
9. Compile and run the client.
Example of using the J2EE™ reference implementation command line packager to package in an EJB™ jar file.
Entity Bean Implementation

1. Create the remote interface for the bean.
2. Create the bean’s home interface.
3. Create the bean’s implementation class.
4. Compile the remote interface, home interface, and implementation class.
5. Create a deployment descriptor.
7. Deploy the enterprise application.
8. Create the client.
9. Compile and run the client.
How it is **deployment** is done in the J2EE™ RI:

1. The J2EE™ process opens the application JAR file, reads the deployment descriptors, and generates the home interface and EJBOBJECT implementation for each Bean.

2. The J2EE™ process compiles the home interface and the EJBOBJECT implementations and then runs the rmic command on the class files. This step creates the stubs and skeletons for the home and remote objects.

3. The server packages the generated classes into a server JAR file and stores the JAR file in the repository.

4. The server creates a client JAR file that contains the home and remote interfaces and the stubs for the home and remote objects.

5. The server sends the client JAR file to the deployer and saves the file according to the name chosen at the start of the deployment process.

6. The location of the client JAR file is added to the CLASSPATH environment variable on any client that calls the application. Then, at runtime, the appropriate stub classes can be loaded so that the client can successfully locate the home object for an enterprise bean in the application.

Example of using the J2EETM reference implementation deploytool to package in an EJB™ jar file.
Deploy Application: Specify JNDI names
Deploy Application: Specify location and Name for Client Jar (home & object stubs)
Deploy Application: Verify or Specify JNDI names
Deploy Application: EJBHome, EJBOBJECT classes and stubs generated, Server & Client Jars generated & copied to Server & Client
Container Deployment Tools:

1. **Parses** the deployment descriptor and **generates** code for declarative **security** and **transactions**.

2. **Generates** server side bean **EJBHome** and **EJBOBJECT implementations**.

3. **Generates** the client-side stubs required to access the **EJB™ server**.

4. **Binds JNDI name** from deployment descriptor to bean **home object** in naming service.

5. **Container makes deployment descriptor environment variables, EJB™ refs, resource refs**, available to bean thru **JNDI context**.
Entity Bean Implementation

1. Create the remote interface for the bean.
2. Create the bean’s home interface.
3. Create the bean’s implementation class.
4. Compile the remote interface, home interface, and implementation class.
5. Create a deployment descriptor.
7. Deploy the enterprise application.
8. **Create the client.**
9. Compile and run the client.
8) Create a Client

1. Use JNDI to lookup EJB’s home interface.
2. Call home’s create or find methods to get the EJB™ remote object interface.
3. Call bean’s business methods thru remote interface.

1) look up the bean home interface, which is published as part of the bean deployment process. Use the Java Naming and Directory Interface (JNDI) to look up the home interface.

2) Use the home interface to create instances of the bean in the server. The bean home interface acts like a factory object, producing bean objects on demand.
When a client wants to create a server-side bean, it uses the Java Naming and Directory Interface (JNDI) to locate the home interface for the class of bean it wants.

What is JNDI?
- A naming and directory interface for Java applications
- Enables access to existing and emerging naming and directory services
- Allows Java programs to locate and use resources by name, to find out information about those resources, and to traverse structures of resources.

What is a Naming and Directory Service?
- Naming Services map people friendly names to addresses or objects: www.sun.com -> 192.9.48.5
- Directory Services add attributes and attribute based searching

Example Naming and Directory services:
- DNS, X.500, NIS, LDAP, NDS, Active Directory, RMI registry, CORBA COS

Usage:
- Locating network resources: Printers, databases, EJB™ components
- Enterprise-wide namespace: Share file systems, network services
- Security: Manage certificates and keys, single sign-on
- Look up attributes of people: Email, white pages
- Support for Distributed Computing: RMI, JINI, CORBA

Initial Context is the starting point for name resolution, “root” for naming context tree:
- Context ctx = new InitialContext (EnvironmentProperties);
  EnvironmentProperties are a hash table of type, value parameters.
  Example: PROVIDER_URL,” ldap::host:port” specifies naming or directory server

Client Looks up object using name:
- Object o = (Object) ctx.lookup("objectname");
  Called by client, returns object corresponding or “bound” to name.

Server Binds name to object in name service:
- Object o;
  ctx.bind("name", o);
  Called by server, binds name to object in naming or directory server. In EJB, JNDI name is set by the deployer in deployment descriptor, and the ApplicationServer/Container “binds” the ejbHome object.
Naming Concepts

The primary function of a naming system is to map names to objects. It maps people-friendly names to addresses, identifiers, or objects typically used by computer programs. For example, the Internet Domain Name System (DNS) maps machine names (such as www.sun.com) to IP addresses (like 192.9.48.5). A file system maps a file name (c:\bin\autoexec.bat for example) to a file handle that a program can use to access the contents of the file. These two examples also illustrate the wide range of scale at which naming services exist—from an entity on the Internet to a file on the local file system. Names

To look up an object from a naming system, you supply it the name of the object. The naming system determines the syntax that the name must follow. This syntax is sometimes called the naming system’s naming convention.

For example, the Unix™ file system’s naming convention is that a file is named from its path relative to the root of the file system, with each component in the path separated left-to-right using the slash character (/). The Unix path name, /usr/hello, for example, names a file hello in the file directory/usr, which is located in the root of the file system.

In the DNS, the naming convention is that components in the DNS name are ordered from right to left, and are delimited by dot characters (‘.’). Thus, the DNS name sales.Wiz.COM names a DNS entry with the name sales, relative to the DNS entry Wiz.COM. The DNS entry Wiz.COM, in turn, names an entry with the name Wiz in the COM entry.

In the Lightweight Directory Access Protocol (LDAP), the naming convention is that components are ordered right to left, and are delimited by comma characters (‘,’). Thus, the LDAP name cn=Rosanna Lee, o=Sun, c=US names an LDAP entry cn=Rosanna Lee, relative to the entry o=Sun, which in turn, is relative to c=us. The LDAP has the further rule that each component of the name must be a name/value pair with the name and value separated by an equal character (‘=’).
The Naming Service provides a layer of abstraction for the identification of objects. It provides readable object identifiers for the human user. Users can assign names that look like structured file names - a persistent identification mechanism. Objects can bind themselves under the same name regardless of their object reference.

The typical use of the Naming Service involves object implementations binding to the Naming Service when they come into existence and unbinding before they terminate. Clients resolve names to objects, on which they subsequently invoke operations.

**Bindings**

The association of a name with an object is called a *binding*. For example, a file name is *bound* to a file. The DNS contains bindings that map machine names to IP addresses. An LDAP name is bound to an LDAP entry.
Each branch in the directory tree is called a naming context, and leaf objects have bindings to specific names. Each NamingContext can be asked to find an object within its branch of the tree by giving its name relative to that naming context. You can get a reference to the root context of the naming directory by creating an Initial Context.

A naming system is a connected set of contexts of the same type (having the same naming convention) and providing a common set of operations.

For example, the DNS is a naming system.

A naming system provides a naming service to its customers for performing naming-related operations. For example, the DNS offers a naming service that maps machine names to IP addresses.

A namespace is the set of names in a naming system. For example, the Unix file system has a namespace consisting of all the names of files and directories in that file system.

In the Lightweight Directory Access Protocol (LDAP), the naming convention is that components are ordered right to left, and are delimited by comma characters (','). Thus, the LDAP name cn=Rosanna Lee, o=Sun, c=US names an LDAP entry cn=Rosanna Lee, relative to the entry o=Sun, which in turn, is relative to c=us. The LDAP has the further rule that each component of the name must be a name/value pair with the name and value separated by an equal character (=).

A context is a set of name-to-object bindings.

Every context has an associated naming convention. A context provides a lookup (resolution) operation that returns the object, and may provide operations such as those for binding names, unbinding names, and listing bound names. A name in one context object can be bound to another context object with the same naming convention, called a subcontext.

For example, a file directory, such as /usr, in the Unix file system is a context. A file directory named relative to another file directory is a subcontext (some Unix users refer to this as a subdirectory). For example, given a file directory /usr/bin, the directory bin is a subcontext of usr.

A DNS domain, such as COM, is a context. A DNS domain named relative to another DNS domain is a subcontext. For example, given a DNS domain Sun.COM, the DNS domain Sun is a subcontext of COM.

An LDAP directory, such as c=us, is a context. An LDAP directory named relative to another LDAP directory is a subcontext. For example, given an LDAP entry o=sun,c=us, the entry o=sun is a subcontext of c=us.
Naming Systems and Namespaces

A **naming system** is a connected set of contexts of the same type (having the same naming convention) and providing a common set of operations.

For example, the DNS is a naming system. A system that communicates using the LDAP is a naming system.

A naming system provides a **naming service** to its customers for performing naming-related operations. A naming service is accessed through its interface. For example, the DNS offers a naming service that maps **machine names to IP addresses**. The LDAP offers a naming service that maps LDAP names to LDAP entries. The file system offers a naming service that maps file names to files and directories.

A **namespace** is the set of names in a naming system. For example, the Unix file system has a namespace consisting of all the names of files and directories in that file system. The DNS namespace contains names of DNS domains and entries. The LDAP namespace contains names of LDAP entries.
A possible view of a *federated* naming system using JNDI.
JNDI Naming concepts:

- **Naming Service** - provides service for associating people friendly names with addresses or objects (i.e., `www.sun.com` -> `192.9.96.82`) and for finding objects given their names.
- **To Lookup** an object (or get an object reference) from a naming system you supply a **name**.
- The **association of a name with an object** is called a **Binding**.
- A **Context** represents a set of name-to-object bindings.
- **Initial Context**: Starting point for name resolution

A naming system provides a *naming service* to its customers for performing naming-related operations. A naming service is accessed through its interface. For example, the **DNS** offers a naming service that maps *machine names to IP addresses*. The **LDAP** offers a naming service that maps LDAP names to LDAP entries. The file system offers a naming service that maps file names to files and directories.

To *look up* an object from a naming system, you supply it the **name** of the object. The naming system determines the syntax that the name must follow. This **syntax** is sometimes called the naming system's **naming convention**.

**References and Addresses**

**Bindings**

The **association of a name with an object** is called a **binding**. For example, a file name is *bound* to a file. The **DNS** contains **bindings** that map *machine names to IP addresses*. An LDAP name is bound to an LDAP entry.

A **context** is a set of *name-to-object bindings*.

Every context has an associated naming convention. A **context** provides a lookup (resolution) operation that **returns the object**, and may provide operations such as those for **binding names**, unbinding names, and listing bound names. For example, a file directory, such as `/usr`, in the Unix file system is a **context**.

A file directory named relative to another file directory is a subcontext (some Unix users refer to this as a **subdirectory**). For example, given a file directory `/usr/bin`, the directory `bin` is a subcontext of `usr`.

A DNS domain, such as **COM**, is a **context**.
A Context object provides the methods for binding names to objects and for looking up objects

**EJB™ Server Binds name to object in name service:**

// Create Remote object to be bound
AccountHome aHome = new AccountHomeImpl();

// Bind object to name in directory
ctx.bind("bankApp/account", aHome);

**EJB™ Client Looks up object using name:**

```java
java.util.Properties p = new Properties();
p.put(Context.INITIAL_CONTEXT_FACTORY, "my.ejb.server.context.factory");
p.put(Context.PROVIDER_URL, "my.server.jndi.url");

// create JNDI InitialContext:
javax.naming.Context ctx = new javax.naming.InitialContext(p);
AccountHome a = (AccountHome) ctx.lookup("bankApp/account");
```

The container automatically registers the EJB™ Home interface for each enterprise bean in a directory. The container uses the JNDI API to register the EJB™ Home interface. The client uses the JNDI service to locate the EJB™ Home object and create a new bean instance or find and remove an existing bean instance.

**Context**

A context is a set of name-to-object bindings.

Before performing any operation on a naming or directory service, you need to acquire an initial context - the starting point into the namespace. This is because all methods on naming and directory services are performed relative to some context.

**Select the Service Provider for the Initial Context**

You can specify the service provider to use for the initial context by creating a set of environment properties (a Hashtable) and adding the name of the service provider class to it. For example, if you are using the LDAP service provider from Sun Microsystems, your code would look like this:

```java
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,
"com.sun.jndi.ldap.LdapCtxFactory");
Context ctx = new InitialContext(env);
```

**Look Up an Object**

Then, use `Context.lookup()` to look up an object.

Object obj = ctx.lookup(name);
8) Client Code

```java
// create JNDI InitialContext:
javax.naming.Context ic = new javax.naming.InitialContext();

// jndi name is set by deployer in deployment descriptor
String jndiName = "Account";  // url can also be in name

// call context lookup to get a reference to the Home interface
java.lang.Object objref = ic.lookup(jndiName);
AccountHome home = (AccountHome)PortableRemoteObject.narrow(objref, AccountHome.class);

// call home create to get a reference to the Account interface (EJBOBJECT stub)
Account acct1 = home.create(414766666, 40000000);
// OR for an existing account use find
Account acct2 = home.findByPrimaryKey(new Integer(414766666));

// call business methods
double amount = acct2.withdraw(325);
```

The JNDI application programming interface (API) makes it easy to plug lookup services from various providers into a program written in the Java language. The client can easily look up information registered with the server and establish communication.

1) get the root naming context by creating a new JNDI InitialContext:
   Context initialContext = new InitialContext(environment);
   The environment parameter is a Java hashtable containing properties that are passed to the java.naming.Context.

   Once created, the JNDI context is used to look up Enterprise Bean home interfaces.

2) lookup the Home interface using the JNDI name given in deployment descriptor:
   Call the context lookup method to get a reference to the home using either the full pathname of the object (if it is local or the server was specified in the environment properties) or use a URL to specify the server where the object is located:
   `<service name>: //<hostname>:<port>/<published name>`
   lookup returns a stub reference to the object implementation of AccountHome

   **Note:** the JNDI name is set by the deployer in deployment descriptor, and the Container "binds" the ejbHome object.
   ctx.bind("account", accountHomeImpl);

3) call home.create or find to get a reference to the Account EJBOBJECT stub.
4) call Account business methods
Entity Bean Implementation

1. Create the remote interface for the bean.
2. Create the bean’s home interface.
3. Create the bean’s implementation class.
4. Compile the remote interface, home interface, and implementation class.
5. Create a deployment descriptor.
7. Deploy the enterprise application.
8. Create the client.
9. **Compile and run the client.**
9) Compile and Run the Client

Compile the Client:

javac  -classpath $J2EE_HOME/lib/j2ee.jar
    AccountClient.java

Run the Client:

java  -classpath
    $J2EE_HOME/lib/j2ee.jar:AccountAppClient.jar
    AccountClient
When a client wants to create a server-side bean, it uses the Java Naming and Directory Interface (JNDI) to locate the home interface for the class of bean it wants.

Once the client has the home interface for the EJB™ class it wants to create, it calls one of the create() methods on the home interface to create a server-side object. The client-side home interface object does a remote method call to the EJB™ container on the server, which then creates the EJB™ component and returns an EJB™ object to the client. The client may then call the EJB™ object’s methods, which are forwarded to the container. The container defers the implementation of the method to the EJB™ component.

A bean runs in a container. The container, which is part of the EJB™ server, provides transaction, synchronization, and security services to the bean. To provide these services, the bean container must be able to intercept calls to bean methods.

For this reason, a client application does not call the remote bean methods directly. Instead, the client invokes the bean method through a two-step process, mediated by the Object Broker and by the container. First, the client calls a local proxy stub for the remote method. The stub marshalls any parameter data, and then calls a remote skeleton on the server. The skeleton unmarshalls the data, and calls the bean container. This is transparent both to the client application developer and to the bean developer. In the second step, the bean container gets the skeleton call, then intercepts whatever services are required by the context. These can include:

• authenticating the client, on the first method call
• performing transaction management
• calling synchronization methods in the bean

• Activation of the bean

The container then delegates the method call to the bean. The bean method executes. When it returns, the thread of control returns to the bean container, which intercepts whatever services are required by the context. For example, if the method is running in a transaction context, the bean container performs a commit operation, depending on the transaction attributes in the bean descriptor.

Then the bean container calls the skeleton, which marshalls return data, and returns it to the client stub. These steps are invisible to client-side and server-side application developers.
Lab Exercise: Design and Implement Customer Entity Bean

Use Case Scenarios

1) New customer “registers” for an online shopping account. A form is filled out with information and a new account is created in the database.

2) Existing customer login to an existing account. Userid (email address) and password are checked against existing account in the database.
The customer table maintains customer information, one record per customer, with information such as customer name, password, and customer address.

In the sample application, a customer object is modeled as entity bean. The concept of customer account is central to all the clients. Multiple clients need to share the concept of creating a customer account, verifying an existing account, and updating account information. Updates to the state of a customer account object need to be written to persistent storage and an account object lives even when the client’s session with the server is over. The bean developer writes the code of the account object as if it is going to be accessed only by one client at a time, with necessary synchronization and concurrency provided by the EJB™ server.

To avoid expensive get methods on remote object account, the example uses a value object representing customer details.

The client’s request to query the state of an account object can then be satisfied via local get methods on this details object. Similarly, to avoid fine-grained set methods, a coarse-grained method to update all account information via one remote call is provided.
Lab Exercise: **CustomerEJB**

tracks customer account information

- **Customer**
  - setCustomerDetails()
  - getCustomerDetails()
  - login()

- **CustomerHome**
  - ejbCreate()
  - ejbFindByPrimaryKey()
  - ejbFindByName()
  - ejbRemove()
  - ejbActivate()
  - ejbPassivate()
  - ejbLoad()
  - ejbStore()

- **CustomerEJB**
  - email
  - address
  - name
  - password
  - creditCardNumber
  - ejbCreate()
  - ejbPostCreate()
  - ejbFindByName()
  - ejbRemove()
  - ejbActivate()
  - ejbPassivate()
  - ejbLoad()
  - ejbStore()

- **EntityBean**
  - setEntityContext()
  - unsetEntityContext()
  - ejbRemove()
  - ejbActivate()
  - ejbPassivate()
  - ejbLoad()
  - ejbStore()

- **EJBObject**
  - getEJBHome()
  - getPrimaryKey()
  - remove()
  - getHandle()
  - isIdentical()

- **EJBHome**
  - remove()
  - remove()
  - getEJBMetaData()

- **EntityBean**
  - ejbCreate()
  - ejbRemove()
  - ejbActivate()
  - ejbPassivate()
  - ejbLoad()
  - ejbStore()
Value Objects

Value objects are used to encapsulate a serializable read-only version of an entire remote object. This allows to retrieve the value of all the details of a remote object.

To avoid expensive get methods on remote object account, the example uses a value object representing customer details.

The client’s request to query the state of an account object can then be satisfied via local get methods on this details object. Similarly, to avoid fine-grained set methods, a coarse-grained method to update all account information via one remote call is provided.
• Working EJB/Servlet/JSP server
• Not a commercial product
• Proof of specification
• Correctness over performance
• Operational definition
• Compliance and Branding
• Baseline functionality
• Write once, Run anywhere (WORA)

Standards:
  - IIOP, HTTP(S), XML

Java APIs:
  - RMI, Java IDL, JDBC, JNDI, JTA ...

Technologies:
  - CosNaming name server
  - Web integration
  - RMI/IOP
  - Distributed transactions
Why A Reference Implementation?

• Operational definition
• Compatibility and branding
• Promote Write Once, Run Anywhere™
• Help developers prototype J2EE™ applications
• Help server platform and tool vendors build J2EE™ support in their products