Chapter 2

Entity Beans: Bean-Managed Persistence

OBJECTIVES

After completing "Entity Beans: Bean-Managed Persistence," you will be able to:

- **Describe the role of entity beans in the EJB architecture.**
- **Differentiate between bean-managed and container-managed persistence mechanisms.**
- **Describe the relationship between an entity bean and its container.**
- **Develop a simple bean-managed persistent entity bean, compile it, and deploy it to the J2EE reference server.**
- **Implement persistence code for a BMP entity bean.**
- **Declare target databases as external resource references.**
- **Map those declared references to JDNI names in deployment.**

© 2000-2006 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

Entity Beans and the Data Layer

- **Entity beans are objects that model persistent data.**
- **This data can take many forms but there is a clear orientation towards relational databases, first and foremost.**
- **Other persistence frameworks are possible:**
  - ODBMS
  - Files
- **Entity beans, once pooled by a container, effect exactly the pattern on which industry consensus has developed for scalable persistence architectures:**
  - A potentially very large number of **instances** of data exist in persistent state – in a DB or file, for instance.
  - A much smaller number of **incarnations** exist at any one time in memory.
  - The relationship between the two allows for a Java object to act as the incarnation of different instances from moment to moment.
  - The container effects the switching of identity.
  - The terms **instance** and **incarnation** as used here are actually not EJB terms, but very useful ones taken from the CORBA Persistent State Service specification, whose design is very similar to EJB's persistence mechanism.

Evaluation Only

© 2000-2006 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

Persistence Mechanisms

- **EJB allows for two means of making a bean class persistent.**
  - **Bean-managed persistence** dictates that the container will tell the bean when to effect what sort of persistence behavior – create, remove, load, store – and that the bean will do the necessary work itself.
  - With **container-managed persistence** the container still notifies the bean, but only as a convenience or hook for additional implementation; the container handles the actual persistence of state.
- **This is the first of many examples we'll see of a dichotomy between declarative and programmatic means of achieving the same thing.**
  - Bean-managed persistence relies on Java code to perform the storage and retrieval itself – this is the more traditional approach.
  - Container-managed persistence offers the opportunity to use a generic persistence engine, provided by the server vendor, by declaring a few important values for that engine's use – in this case, the names and types of persistent fields.
- **BMP and CMP illustrate this difference nicely.**
  - CMP is clearly easier, just based on this definition, and you'll see this confirmed in the next two chapters.
  - However, BMP offers more complete control. Some services are easier to manage with declarative language than others.

Evaluation Only

© 2000-2006 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

Persistence Methods

- **The orientation of EJB architecture towards objectifying relational data can be seen in the obvious mapping of entity bean methods to SQL statements:**

ejbCreate (...)	**INSERT**
ejbRemove	**DELETE**
ejbLoad	**SELECT one**
ejbStore	**UPDATE**
ejbFind* (...)	**SELECT one or several**

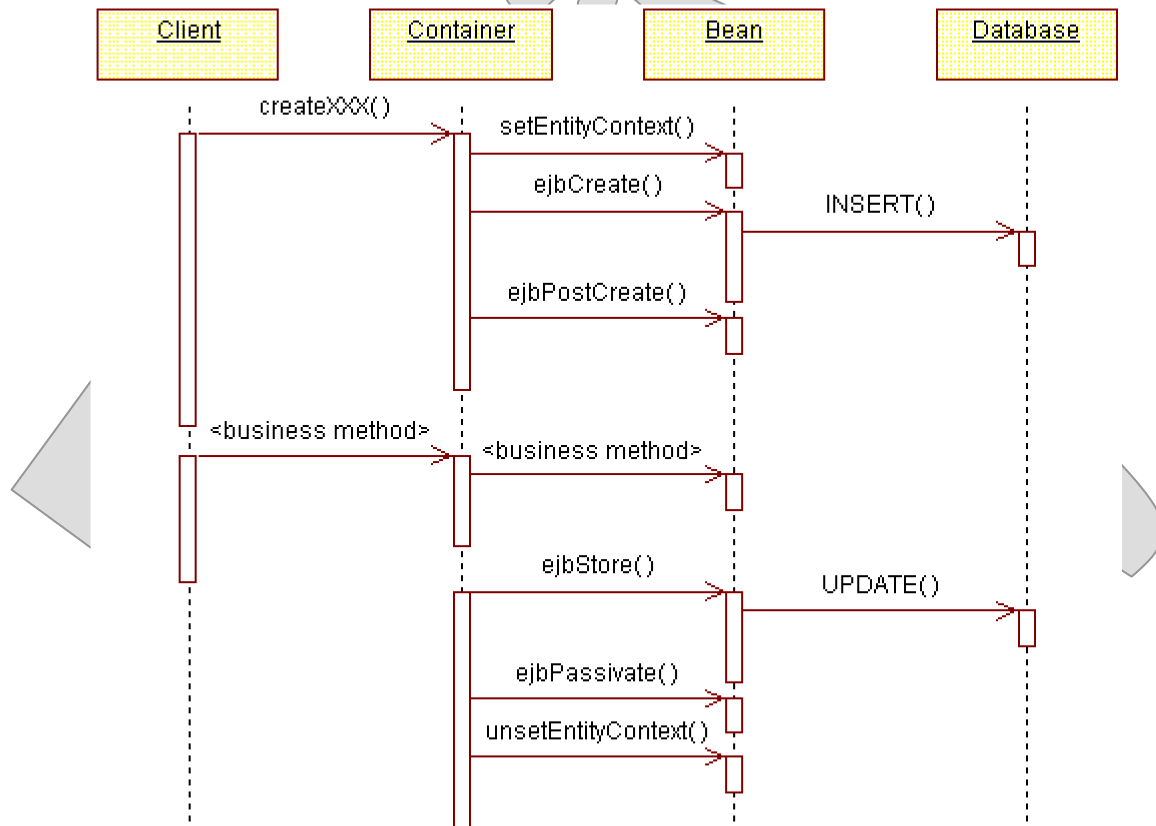
- **These methods are called at appropriate times by the container.**
  - For instance when the client looks up the home object and calls **create**, the home object implementation will eventually delegate to the corresponding **ejbCreate**, but clearly the Java object already exists at this point. That's the **incarnation ...**
  - **ejbCreate** is responsible for creating the **instance**.
  - **ejbLoad/Store** will be called at more mysterious times, as deemed appropriate by the container, especially when an active bean (representing a specific instance) is being returned to the pool for other use, and then re-activated.

Evaluation Only

© 2000-2006 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

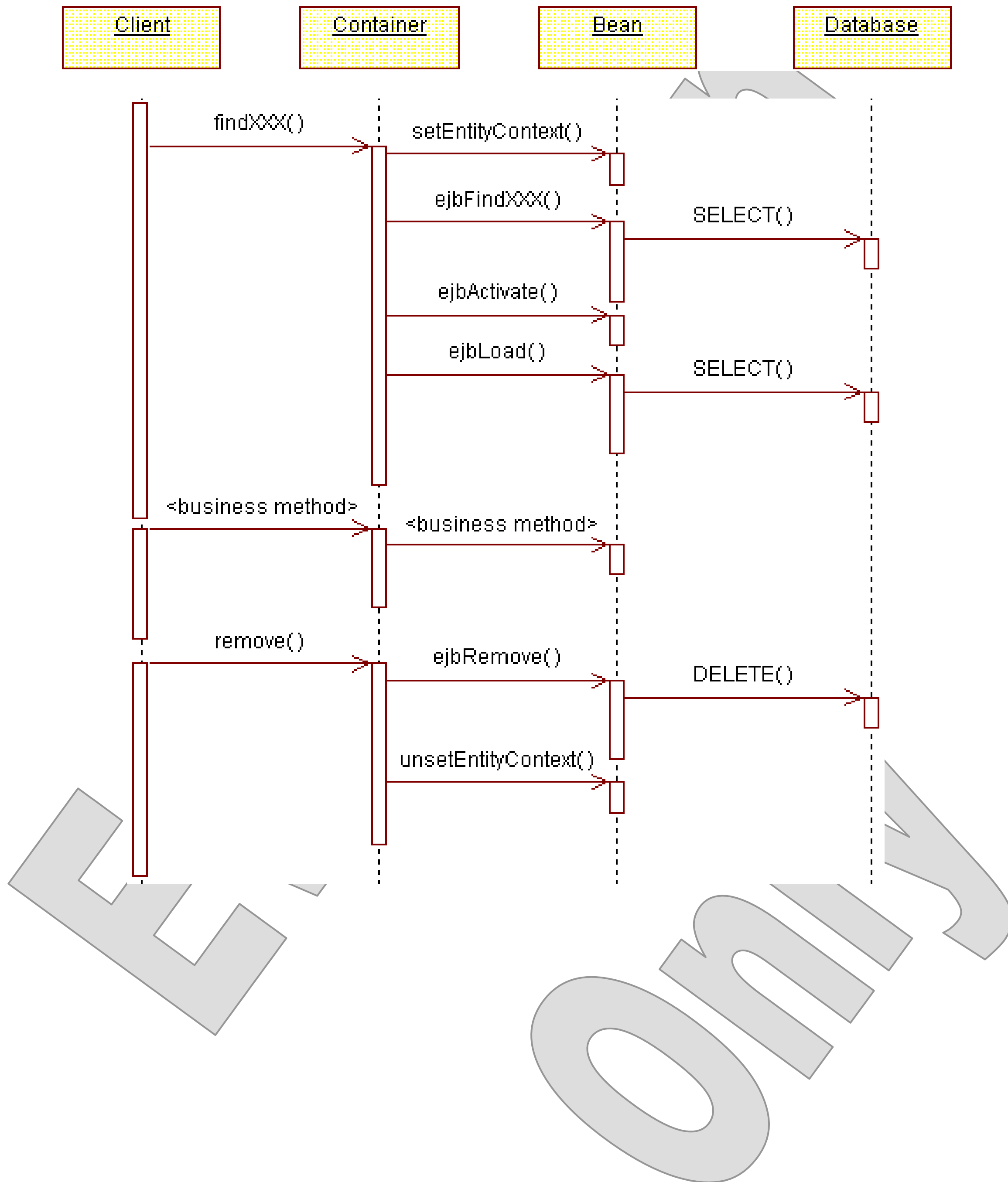
Interactions and Lifecycle

- **The life span of any EJB (not just entities) is under the complete control of the EJB container.**
- **Consider the following interaction diagrams.**
  - Note however that these are merely some of the possible sequences.
  - Especially, bean creation and destruction need not be synchronized to client requests, although it is convenient to consider this particular scenario.



© 2000-2006 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

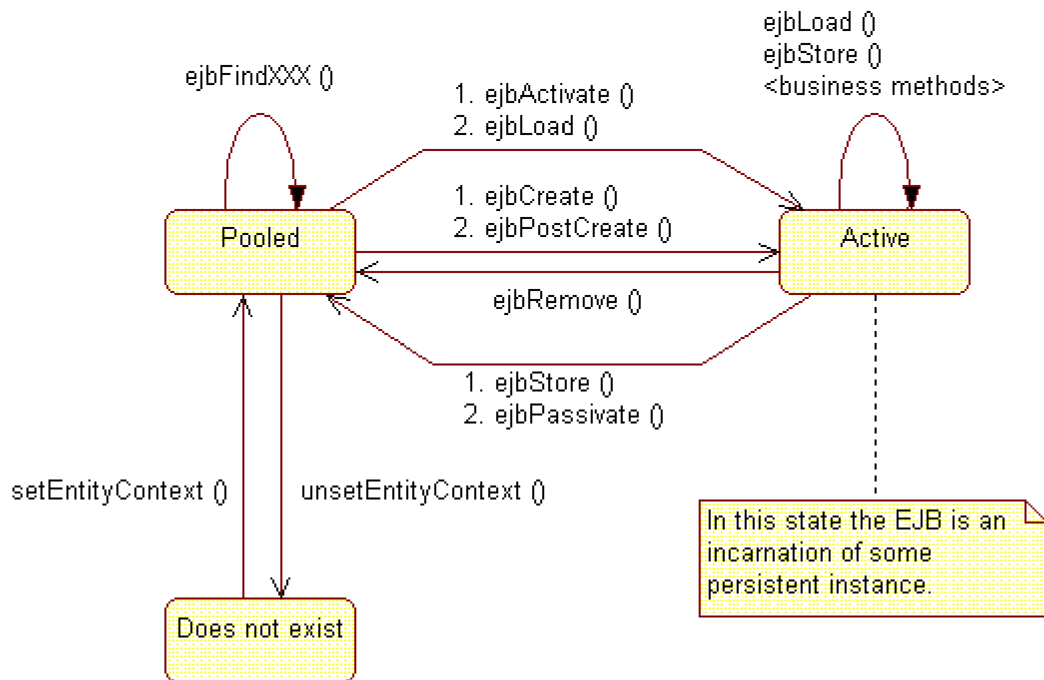
Interactions and Lifecycle



© 2000-2006 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

Entity Bean State Transitions

- An entity bean can exist in either of two states:
  - **Pooled**, in which it cannot represent a data instance, but can serve requests on the bean's finder methods
  - **Active**, in which the bean incarnates a specific data instance



- There are some operations that do not effect any state transition; these are excluded from the above diagram.

© 2000-2006 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

Bean-Managed Persistence

- **How are persistence methods implemented?**
- **The specification defines two possibilities.**
- **The bean code can implement these methods explicitly – this is Bean-Managed Persistence, or BMP.**
  - This is laborious, as we'll see, but gives complete control.
  - This is essentially **programmatic** persistence.
- **The container can read XML definitions of fields, types, primary key, etc., and implement persistence automatically – this is Container-Managed Persistence or CMP.**
  - This is much easier, but the mapping from object to data must be comprehensible to the container.
  - The container implementation may be primitive or very sophisticated, and XML allows for more detailed declarations if necessary.
  - This is **declarative** where BMP is programmatic.

Evaluation Only

© 2000-2006 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

Deployment Settings

- **A BMP entity declares that it will manage its own persistence via the deployment descriptor:**

```
<persistence-type>Bean</persistence-type>
```

  - In this way the container is informed that it should not perform persistence chores automatically.
  - It will assume that the bean will perform these in its code for the various methods on the **EntityBean** interface.

- **Also, whatever resource the bean will need to create, remove, load, store, and find must be declared as a resource reference.**
  - This is another example of the separation of roles: the bean provider knows what is needed and gives it a name ...

```
<resource-ref>
  <res-ref-name>MyData</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Container</res-auth>
</resource-ref>
```

  - ... and the deployer knows where that resource will be located on a given system (development environment, client site, ...) and maps the reference to a full JNDI name for that system.

```
<reference-descriptor>
  <res-ref-name>MyData</res-ref-name>
  <jndi-name>jdbc/ActualDatabase</jndi-name>
</reference-descriptor>
```

Evaluation Only

© 2000-2006 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

Bean Context

- All EJBs have access to a **context object** – a window on the outside world:

```
public interface javax.ejb.EJBContext
{
  java.security.Identity getCallerIdentity ();
  java.security.Principal getCallerPrincipal ();
  javax.ejb.EJBHome getEJBHome ();
  java.util.Properties getEnvironment ();
  boolean getRollbackOnly   ();
  javax.transaction.UserTransaction
    getUserTransaction ();
  boolean isCallerInRole (java.lang.String);
  boolean isCallerInRole (java.security.Identity);
  void setRollbackOnly ();
}
// Again, exception sigs removed ...
```

- The container passes a **context** reference to the bean when the bean is activated from the pool.
  - While in the pool, the bean has no context.
- The bean must store this if it will need it.
- Use this to "find yourself" in bean code:
  - An entity bean gets its primary key for implementations of persistence methods – we'll see such code in this chapter.
  - Session beans can monitor and control transactions.
  - Any bean can perform authorization of a security principal to perform a certain action or access a certain resource.

Evaluation Only

© 2000-2006 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

Entity Context

- **An entity bean will be passed an EntityContext.**

```
public interface javax.ejb.EntityContext
  extends javax.ejb.EJBContext
{
  javax.ejb.EJBObject getEJBObject ();
  java.lang.Object getPrimaryKey ();
}
```

- **Call getEJBObject on the cached context reference to get a reference to the proxy object for your bean.**
  - Callers can only see this proxy, never your bean itself.
  - Your bean doesn't need much from this object, but must be able to reference it explicitly – for instance when passing reference to itself in a call to another bean.
- **getPrimaryKey is critical to bean-managed persistence code, and is useful whenever an entity must check to see which persistent instance it is representing.**
  - Remember, it's not safe to save the PK value in a field from call to call. The bean may be used as an incarnation of a different persistent object the next time it's called!
- **Most of the hook methods on an entity bean class will begin by deriving their primary-key-of-the-moment:**

```
public void ejbLoad ()
{
  Integer key = (Integer) context.getPrimaryKey ();
  ...
```

Evaluation Only

© 2000-2006 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

Primary Keys

- **Also, an entity bean must work in prescribed ways with a defined primary key – obviously, an RDB concept.**
- **The container relies on primary keys for**
  - creation methods
  - finder methods
  - other hook methods – for instance **ejbStore** needs to know the primary key of the instance.
- **The primary key is encapsulated as a Java class, defined for each entity bean type. Why is this? Most database key types will be primitives in practice ...**
  - Abstracting the primary key as a class is the most flexible approach – Java hasn't templates, so we need a class to get an **equals** method, etc.
  - The PK can be an opaque type, which has its advantages.
  - The container and even the client can work with a PK object without knowing the implementation.
  - Most compellingly, primitive types could not express composite keys!
- **The PK type can be declared to be String, Integer, Long, etc. The PK class need not be dedicated to a specific bean type.**

Evaluation Only

© 2000-2006 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

Create Methods

- **An entity bean will typically expose at least one creation method.**
- **There may be more than one: as many overloads of the Java method as are useful.**
  - This is analogous to overloading an ordinary constructor.
- **There may be none!**
  - This allows the EJB to "hide" creation semantics.
- **Each creation method must be defined in two places:**
  - The home interface, as a method named **create**. This is the method the client will call to create a bean instance.
  - The bean class itself, as a method named **ejbCreate**. This will be invoked on a (possibly new) instance of the bean class after the container has assigned it to a persistent instance, but before that instance has been created in a database.
- **The two methods must have the same parameters and throw the same exceptions.**
  - One exception on exceptions: **create** will throw **java.rmi.RemoteException**; **ejbCreate** will not.
- **The bean's implementation must create the new persistent instance and return its primary key.**
- **This often means effecting a SQL INSERT.**

© 2000-2006 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

Evaluation Only

Bean Removal

- Implement **ejbRemove** to remove the persistent instance from the database – typically a SQL DELETE.
- This method will be called by the container and/or home object when the client calls **remove** on the home.
- A BMP bean will have to derive its primary key from its context before making the deletion.

Evaluation Only

© 2000-2006 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

ejbLoad and ejbStore

- Load and store methods are not mirrored in the home interface.
- This is because, unlike creation and destruction, these behaviors are not to be exposed to the client.
- The container alone has access to these method, and it calls them as part of managing the object pool.
- Implement **ejbLoad** to update the bean state from persistent data.
- Implement **ejbStore** to update the persistent data from the bean state.
- In both cases the primary key must be pulled from the bean context.

Evaluation Only

© 2000-2006 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

Finder Methods

- **An entity bean must expose at least one finder method, which takes a primary key as its parameter.**
  - On the home interface this is **findByPrimaryKey**.
  - On the bean it is **ejbFindByPrimaryKey**.
- **Other finders may be defined, but since parameter lists alone may not be enough to distinguish them, they are to be given distinct method names.**
- **The name of any finder must begin with "find" as defined on the home interface, and thus with "ejbFind" on the bean class.**
- **Each finder represents a possible query related to the bean class.**
  - If the finder returns zero or one objects, its return type must be a primary key type. (The method on the home interface returns an object reference, and the home object translates.)
  - If the finder can return many objects, its return type must be an **Enumeration**, a **Collection** or a **Set** – the first of these is an EJB 1.0 legacy, and its use is generally discouraged.
- **Note that a call to a finder – even ejbFindByPrimaryKey – does not effect a state transition on the called bean.**
  - Only on subsequent calls from the container might the bean be activated as the recently-found instance.

Evaluation Only

© 2000-2006 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

Using PointBase Databases

- **The J2EE reference implementation bundles the PointBase RDBMS to support its own sample code.**
- **Our exercises will also use PointBase databases.**
  - SQL scripts are submitted to PointBase through a JDBC driver (also bundled) to create databases and schema.
  - PointBase itself is "embedded" in various clients, the AS8 server, and administrative processes as loaded from a JAR.
  - Thus there is no separate RDBMS process to start and stop.
- **For each database that will be used by a J2EE application, the AS8 domain must be configured with:**
  - A **JDBC connection pool** that captures the details of connecting to the database: URL, driver or data-source class, logon credentials, and other mundane items.
  - A **JDBC resource** that adapts this connection pool for use by J2EE applications and components. Components find the JDBC resource using JNDI (typically through relative names in their local environments).

- **The asadmin tool provides commands for managing these objects.**
- **Note that in effecting these configurations we'll be playing the administrator role as defined in the EJB specification.**

Evaluation Only

© 2000-2006 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

Calling an EJB

- **Client code will usually implement a few common steps to make contact with an EJB:**
  - Look up a home object by name using JNDI.
  - Call a creation or finder method on the home object to derive a single bean instance or a collection of beans.
  - Make calls on the bean or beans.
  - In the case of a session bean, remove after having created it.
- **Clients of remote EJBs must take care when downcasting the type of remote references.**
  - Due to the extensive use of delegation in stubs and skeletons, the caller cannot be certain that a simple Java downcast will be safe, even if the referenced object does support the required interface.
  - Therefore it is important to use the safe typecasting offered as part of the RMI framework.
  - Call the static method **javax.rmi.PortableRemoteObject.narrow**, passing the target reference and the desired type as a **Class** instance.
  - The result must still be downcast to satisfy the compiler, but the reference returned by this method is guaranteed to downcast safely.

```
SomeHome home = (SomeHome)
  javax.rmi.PortableRemoteObject.narrow
    (context.lookup ("TheName"), SomeHome.class);
```

Evaluation Only

© 2000-2006 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

The Love Is Blind Dating Service

- **Introducing the Love Is Blind Dating Service!**
- **Our demonstrations will focus largely on the process of deploying to the J2EE reference implementation server.**
  - Developing EJBs without the aid of a code-generating tool is rather labor-intensive.
  - Thus our demos will not involve any actual coding, although we will examine completed code in some detail.
- **Demo source files can be found in Demos\Entity.**
  - Java source files for the EJBs are in the subdirectory **cc\dating**.
  - In the main directory for the demo are supporting XML files and scripts for compiling, deploying and testing.
- **JSPs are in the Website subdirectory of each step, and will be packed up into a web application resource (WAR) file as part of the complete application.**
  - These will not appear until later versions of the application.
  - For now, the only clients will be J2SE applications.

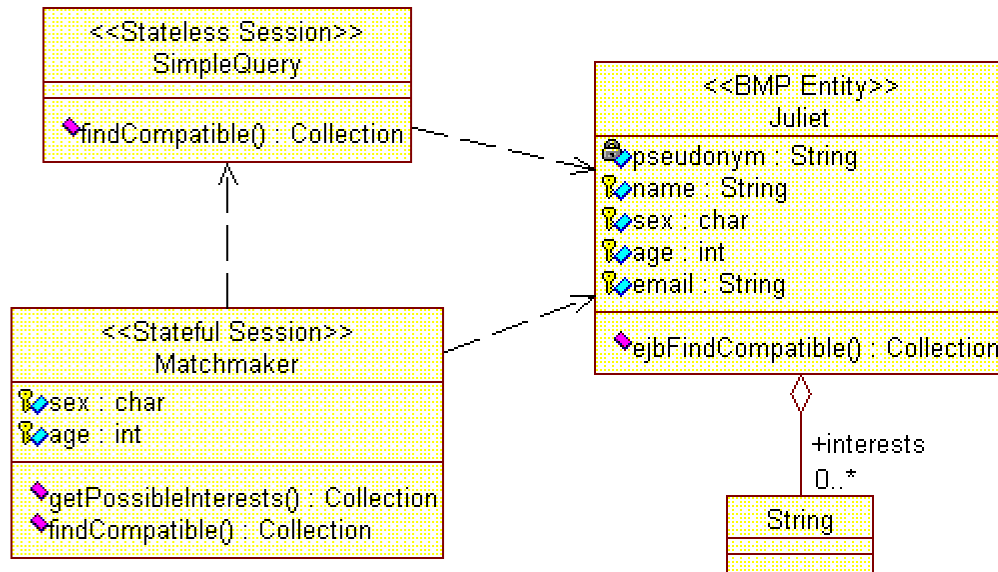
Evaluation Only

© 2000-2006 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

The Juliet Entity Bean

DEMO

- **Here is a summary of the Love Is Blind EJB design, as we'll see it implemented over the next several chapters:**



- **In this demo we'll look at a completed entity bean representing the Juliet class.**
  - In later chapters we'll look at the session beans that complete the design.

Evaluation Only

© 2000-2006 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

The Juliet Entity Bean

DEMO

1. We'll begin with a look at the bean's Java code – firstly, the remote interface in **cc\dating\Juliet.java** – this defines the **business logic** that motivates use of this bean:

```
package cc.dating;

import javax.ejb.EJBObject;
import java.rmi.RemoteException;

public interface Juliet
    extends EJBObject
{
    public void setInterests (String[] interests)
        throws RemoteException;

    public void setEmail (String email)
        throws RemoteException;

    public String[] getInterests ()
        throws RemoteException;

    public String getEmail ()
        throws RemoteException;

    public int getAge ()
        throws RemoteException;

    public char getSex ()
        throws RemoteException;

    public String getPseudonym ()
        throws RemoteException;

    public String getName   ()
        throws RemoteException;
}
```

Evaluation Only

© 2000-2006 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

The Juliet Entity Bean

DEMO

2. The home interface – again, this interface is exposed on a generated and container-managed object just once per bean type – in **cc\dating\JulietHome.java**:

```
package cc.dating;

import javax.ejb.EJBHome;
import javax.ejb.CreateException;
import java.rmi.RemoteException;
import java.util.Enumeration;
import javax.ejb.FinderException;

public interface JulietHome
    extends EJBHome
{
    public Juliet create (String name,
        String pseudonym, char sex, int age,
            String email)
        throws CreateException, RemoteException;

    public Juliet findByPrimaryKey (Integer key)
        throws RemoteException, FinderException;

    public java.util.Collection findCompatible
            (char sex, int age, String[] interests)
        throws FinderException, RemoteException;
}
```

Evaluation Only

© 2000-2006 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

The Juliet Entity Bean

DEMO

3. Finally, **cc\dating\JulietEJB.java** provides the actual implementation for the bean. We'll skip a detailed look at this file for now, as most of it is persistence implementation. Notice, though, that methods from both the remote and home interfaces are implemented here – the home-interface methods have slightly different names. Also note that there is no inheritance relationship between any of the three.

```
public class JulietEJB
    implements EntityBean
{
    ...
    public Integer ejbCreate (String name,
        String pseudonym, char sex, int age,
            String email)
        throws CreateException
    ...
    public void setInterests(String[] interests)
    ...
}
```

4. Remember that a primary-key class is required for all entity beans. We have defined (or more accurately we will define) our PK to be of type **Integer**, which simplifies our work since the built-in class satisfies all the requirements for a primary-key representation and will be usable by the container. Strictly speaking, though, always consider the primary-key class to be the fourth of four required .java files for EJB deployment.

Evaluation Only

© 2000-2006 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

The Juliet Entity Bean

DEMO

5. The remaining requirements for deployment are expressed at various levels of the deployment descriptor. Here is a fragment of the portable descriptor **ejb-jar.xml** that declares the key features of the **Juliet** bean:

```
<entity>
  <display-name>Juliet</display-name>
  <ejb-name>Juliet</ejb-name>
  <home>cc.dating.JulietHome</home>
  <remote>cc.dating.Juliet</remote>
  <ejb-class>cc.dating.JulietEJB</ejb-class>
  <persistence-type>Bean</persistence-type>
  <prim-key-class>java.lang.Integer
    </prim-key-class>
  <reentrant>false</reentrant>
  ...
  <resource-ref>
    <res-ref-name>Database</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <res-auth>Container</res-auth>
  </resource-ref>
</entity>
```

6. Notice that the bean manages persistence, and that it declares a **resource reference** on which it will rely for a database connection at runtime. This name is mapped in **sun-ejb-jar.xml** to a database we're about to create and configure for the AS8 domain.

```
<ejb>
  <ejb-name>Juliet</ejb-name>
  <jndi-name>Juliet</jndi-name>
  <resource-ref>
    <res-ref-name>Database</res-ref-name>
    <jndi-name>jdbc/LoveIsBlind</jndi-name>
  </resource-ref>
</ejb>
```

Evaluation Only

© 2000-2006 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

The Juliet Entity Bean

DEMO

7. Start the default domain, if necessary, and run **asant init-DB** to create and configure the database. This runs two targets: the first submits a SQL script to PointBase to build a three-table schema and the second runs **asadmin** to build a connection pool and JDBC resource for the domain:

```
asant init-DB
create-DB:
  [echo] Creating database ...
  [sql] Executing file:

C:\Capstone\EJBIntro\Demos\Entity\create_DB.sql
  [sql] Failed to execute: DROP TABLE INTERESTS
  [sql] java.sql.SQLException: Cannot find the
         table "PBPUBLIC.INTERESTS".
  [sql] Failed to execute: DROP TABLE JULIET
  [sql] java.sql.SQLException: Cannot find the
         table "PBPUBLIC.JULIET".
  [sql] Failed to execute: DROP TABLE JULIET_KEY
  [sql] java.sql.SQLException: Cannot find the
         table "PBPUBLIC.JULIET_KEY".
  [sql] 8 of 11 SQL statements executed
         successfully

configure-DB:
  [exec] create-jdbc-connection-pool ...
  [exec] Command create-jdbc-connection-pool
          executed successfully.
  [exec] create-jdbc-resource ...
  [exec] Command create-jdbc-resource executed
          successfully.
```

(Don't worry that the three SQL table deletes failed. The script is built to clear out any existing tables before building new ones, so that it can be run repeatedly to reset the database. Thus on the first run of the script these will always fail – harmlessly.)

Evaluation Only

© 2000-2006 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

The Juliet Entity Bean

DEMO

8. Run **asant** to build and deploy the application. This will do more or less the same work that the build for last chapter's **Hello, EJB** application did. Also, it will automatically run the **JulietClient** application, which automatically prime the schema with data for 16 dating-service members. Selected console output:

```
...
run-juliet:
  [java] Existing Juliets:

  [java] Resetting the database.

  [java] Populating the database.
  [java] Adding James Stevens
  [java]  sailing gourmet badminton
  [java] Adding Robert Doolittle
  [java]  sports checkers music
  [java] Adding Samuel Filten
    [java]  music politics gourmet
  ...
```

9. Now that it's run successfully, take a look at the client source code in **cc\dating\admin\JulietClient.java**. Note that the client looks up the **Juliet** home object to find all existing objects, report and then delete them, and then to populate the database with 16 members, simply by creating that many **Juliet** instances with prepared values. See code on the following page.

Evaluation Only

© 2000-2006 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

The Juliet Entity Bean

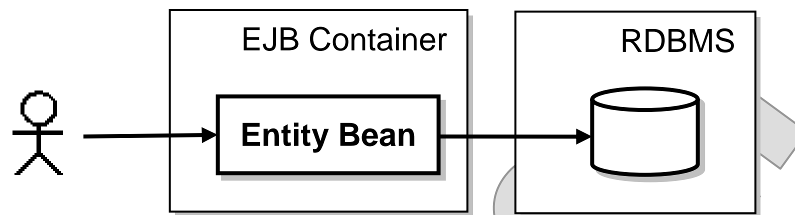
DEMO

```
Context context = new InitialContext ();
JulietHome home = (JulietHome)
  PortableRemoteObject.narrow
    (context.lookup ("Juliet"), JulietHome.class);

System.out.println ("Removing existing Juliets:");
Iterator existing = home.findAll ().iterator ();
while (existing.hasNext ())
{
  Juliet doomed = (Juliet) existing.next ();
  System.out.println (doomed.getName ());
  doomed.remove ();
}

System.out.println ();
System.out.println ("Populating the database:");
Iterator each = new TableData ().iterator ();
while (each.hasNext ())
{
  TableData.Record record =
    (TableData.Record) each.next ();
  System.out.println ("Adding " + record.name);

  Juliet member = home.create (record.name,
    record.pseudonym, record.sex, record.age,
      record.email);

  for (int i = 0; i < record.interests.length; ++i)
    System.out.print (" " + record.interests[i]);
    System.out.println ();
  member.setInterests (record.interests);
}

context.close ();
```

Evaluation Only

© 2000-2006 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

End-to-End Interactions



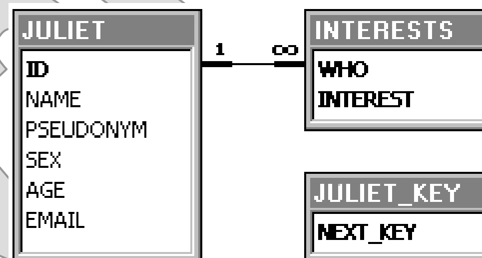
- **So, your test client – a Java application – is operating on application data as follows:**
  - The client uses JNDI to find the **Juliet** home object.
  - First the client runs the **findAll** query to enumerate all instances, and as it goes it **remove**s them from the database.
  - Then the client iterates over hardcoded data in the **TableData** class. for each record, it creates a **Juliet** instance, populates it, and moves on to create another.
  - The container will at some point deactivate each of the requested entity beans – incarnations of DB entities, remember – replacing them in the container's pool, but since the client never calls **remove**, the persistent data – the true instances – are never deleted.

Evaluation Only

© 2000-2006 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

Implementing Persistence

- **Consider the persistence code in the Juliet bean – look in Demos\Entity\cc\dating\JulietEJB.java.**
- **Note that this bean makes use of three tables:**
  - **JULIET** is the home of most of the bean's attributes.
  - **INTEREST**s are broken out into a separate table. This is sensible data design but would throw many container-managed persistence engines off their game. By using bean-managed persistence we have complete control over the data layout.
  - To generate unique primary keys for **JULIET** instances, we keep a single global value in **JULIET_KEY**. This too would confuse a CMP engine, but most commercial O/R tools are able to generate IDs themselves, so this wouldn't likely be necessary.



Evaluation Only

© 2000-2006 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

Implementing ejbCreate

- **Except for the use of these three tables instead of one, the persistence code in JulietEJB is pretty much by the book.**
  - Note that **ejbCreate** inserts new rows and returns the key value.
  - It does not populate **Interests**, but leaves that for a later call to **ejbStore**, which we can be sure will come.
  - All the other persistence methods need to "discover themselves" before interacting with the database: they each start with a call to **context.getPrimaryKey**.

```
public Integer ejbCreate (String name,
  String pseudonym, char sex, int age,
    String email)
  throws CreateException
{
  System.out.println ("ejbCreate () ...");

  // First set the passed values into the bean's
  // fields as a Java object:
  this.name = name;
  this.pseudonym = pseudonym;
  this.sex = sex;
  this.age = age;
  this.email = email;
  this.interests = null;

  ...
```

Evaluation Only

© 2000-2006 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

Implementing ejbCreate

```
  // Now create the instance in the database:
  try
  {
    Connection conn = getConnection    ();

    Statement getKey = conn.createStatement ();
    String SQL = "SELECT NEXT_KEY FROM JULIET_KEY";
    System.out.println (SQL);
    getKey.execute (SQL);
    ResultSet results = getKey.getResultSet ();
    results.next ();
    int key = results.getInt (1);
    results.close ();
    getKey.close ();

    Statement updateKey = conn.createStatement ();
    SQL = "UPDATE JULIET_KEY SET NEXT_KEY = " +
      (key + 1) + " WHERE NEXT_KEY = " + key;
    System.out.println (SQL);
    updateKey.execute (SQL);
    updateKey.close ();

    Statement createRow = conn.createStatement ();
    SQL = "INSERT INTO JULIET (ID, NAME, " +
      "PSEUDONYM, SEX, AGE, EMAIL) VALUES (" +
      key + ", '" + name + "', '" + pseudonym +
      "', '" + sex + "', " + age + ", '" + email +
      "')";
    System.out.println (SQL);
    createRow.execute (SQL);
    createRow.close ();

    conn.close ();

    return new Integer (key);
  }
}
```

Evaluation Only

© 2000-2006 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

Remaining Persistence Methods

- We will not reprint the code for all the **Juliet** persistence methods here.
- This decision itself speaks to the bulkiness of the code, and this is typical of BMP beans.
- Not only is there a lot of code to write, but much of it is at once tedious to write and highly error-prone, since the SQL passed through the JDBC interface cannot be validated until runtime.

Evaluation Only

© 2000-2006 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

Bean Passivation/Activation

- **Entity beans are passivated before disappearing into the pool, and then activated out of the pool.**
- **Methods must be implemented on EntityBean interface.**
- **Passivation and activation are most important for stateful session beans, and we'll come back to them in more detail in later chapters.**
  - The ability is less important for entities since there is always a store/load call right next to the passivate/activate call.
  - The specification does distinguish different roles for the methods, though: passivation is about releasing resources, while storage is about writing persistent state; similarly for activate/load.

Evaluation Only

© 2000-2006 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

Eager Persistence

- **The EJB container is responsible for assuring that an entity bean remains perfectly synchronized with the persistent data it represents.**
  - That is, the **incarnation** of the data must have the same state as the persistent **instance**.
- **To do this, it relies on the ejbLoad and ejbStore methods, which in the case of container-managed persistence it implements itself.**
- **The container must "flush" any changes to the bean as data incarnation down to the persistent instance.**
  - This must happen whenever the bean's data changes.
  - How can the container know when this is?
  - The best it can do, according to the EJB specification, is to treat any business method as a mutator.
  - Therefore an EJB container will call **ejbStore** on the bean after each business method invocation – even on a call to **getName**, for example.
  - This can result in a large number of unnecessary and expensive Java and SQL invocations. (Now don't you wish Java had some meaningful **const** semantics?)

Evaluation Only

Calming Things Down

- **A BMP entity bean can avoid the costs of repetitive and needless persistence operations by keeping track internally of its dirty state.**
- **The basic strategy is:**
  - Declare a private boolean field such as **dirty**, initially **false**.
  - Set the flag to **true** in business methods that change state.
  - Check the flag in **ejbStore** and skip SQL or other save operations if the flag is **false**.
  - Be sure to reset the flag to **false** after all persistence calls: **ejbStore**, **ejbLoad**, **ejbCreate**, and **ejbRemove**. This will assure that the object is considered "clean" when activated to incarnate a new persistent instance.
- **It is also possible to optimize over subsets of a bean's persistence operations.**
  - If a bean writes to multiple tables – for instance a coarse-grained bean that internally manages table relationships – it's a good idea to keep a flag for each table.
  - In this way a change to the parent but not the children will not instigate a riot of useless child updates.
- **EJB containers may also provide a means of declaring "const-ness" for business methods, and skip their usual calls to ejbStore accordingly.**
  - This would not be portable, however.

Evaluation Only

© 2000-2006 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

Optimizing Juliet

LAB 2

In this lab you will refine the JulietEJB class to eliminate unnecessary database calls that currently result from container calls to ejbStore when in fact the bean's state is clean.

Complete instructions for this lab can be found at the end of the chapter.

Suggested time: 45 minutes.

Evaluation Only

© 2000-2006 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

SUMMARY

- Entity beans offer a straightforward means of incarnating persistent data in memory **as Java objects**, and not just as some data structure or static information.
- Entity beans are full-fledged classes/components, and can certainly have useful business-specific behavior defined for them.
- The EJB spec suggests that behavior on an entity ought to be kept to a minimum.
- This is debatable to say the least, and one of the statements that motivated the discussion in Chapter 1, sounding as it does like a capitulation to a data-centric philosophy.
- Bear in mind, though, that entities will always be closely bound to data definitions, and so most business logic that is not directly concerned with a particular entity will usually reside in session beans.
- The power of entity beans in EJB is that they give us a true object mapping from relational data.
- How natural and effortless this mapping can be is an open question, and we will look at CMP in the next chapter.

© 2000-2006 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

Optimizing Juliet

LAB 2

Introduction

In this lab you will refine the **JulietEJB** class to eliminate unnecessary database calls that currently result from container calls to **ejbStore** when in fact the bean's state is clean.

Suggested Time: 45 minutes

Directories:	**Labs\Lab2**	(do your work here)
	Examples\LoveIsBlind\Step1	(backup of starter files)
	Examples\LoveIsBlind\Step2	(answer)
Files:	**cc\dating\JulietEJB.java**	
	cc\dating\admin\StandardizeEMail.java	

Instructions

1. Build and test the starter application – just run **asant**. Once it's deployed and the database has been populated by the **JulietClient**, run a second prepared client application, **StandardizeEMail**, by passing a specific target to Ant as shown below. This client updates all member e-mail addresses according to a standard scheme. In doing so, it makes a number of calls to accessor and mutator methods on the **Juliet** bean. It reports the total time consumed by the task when it's done:

```
asant -Dclient-class=cc.dating.admin.StandardizeEMail run-client
   [java] Standardizing new email addresses ...
   [java] Update took 1413 milliseconds.
```

2. Open **JulietEJB.java**. Add two new boolean fields to the class, called **julietDirty** and **interestsDirty**, and initialize them to **false**.

3. Create private methods **setJulietDirty** and **setInterestsDirty** to set the respective flags to **true**.

4. Create a private method **clean** to set both flags back to **false**.

5. Call **clean** from each of the methods **ejbCreate** and **ejbLoad**.

6. Call **setJulietDirty** in the mutator method **setEMail**.

7. Call **setInterestsDirty** in the mutator **setInterests**.

Evaluation Only

© 2000-2006 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.

8. In **ejbStore**, check both flags right at the beginning, and if they are both **false**, simply return from the method. Then, bracket the code paragraphs (as indicated by code comments) that update the JULIET and INTERESTS tables with tests of the appropriate dirty flags, and skip each code section if it is not necessary. At the bottom of **ejbStore**, call **clean**. Save the source file.

9. Rebuild and re-deploy the application. Try the **StandardizeEMail** client again, and you should see that your optimizations have dramatically reduced the time necessary to perform this batch script. (Average from some very informal testing is a 42% improvement in the total time.)

```
asant -Dclient-class=cc.dating.admin.StandardizeEMail run-client
  [java] Standardizing new email addresses ...
  [java] Update took 610 milliseconds.
```

Evaluation Only

© 2000-2006 Will Provost.
All Rights Reserved by Capstone Courseware, LLC.