This session is about Entity beans. Understanding entity beans is very important. Yet, entity bean is a lot more complex beast than a session bean since it involves backend database. So we have a very large number of slides here (almost 230 slides) and it will take sometime for you to digest it.
Disclaimer & Acknowledgments

Even though Sang Shin is a full-time employee of Sun Microsystems, the contents here are created as his own personal endeavor and thus do not reflect any official stance of Sun Microsystems.

Sun Microsystems is not responsible for any contents in this presentation.

Acknowledgments

- Large number of slides are made from the contents of the book "Applied Enterprise JavaBeans" Technology written by Kevin Boone (Sun Microsystems), published by Prentice Hall [1]
- Some slides are made from the contents of J2EE tutorial authored by Dale Green (Sun Microsystems) [2]
- Some slides (Relationship related) are made from Mastering EJBs authored by Ed Roman [3]
- Some slides are made from the contents of EJB codecamp material authored by Carol McDonald (Sun Microsystems)
Revision History & Things to Do

- 03/01/2003: version 1, Created (Sang Shin)
- 03/03/2003: Updated (Jeff Cutler)
- 03/24/2003: Updated (Sang Shin)
- 06/20/2003: Speaker notes are added for most slides (Sang Shin)
- 07/01/2003: Some slides which were out of place at the end are cleaned up (Sang Shin)
- 07/13/2003: advanced topics are separated out into AdvancedEntityBeans presentation (Sang Shin)
- 02/21/2004: Editing changes (thanks to Irina Chekanov)
- Things to do
  - Some slides still need speaker notes
This is the agenda. First, we will talk about what is and why and when you want to use entity beans as opposed to session beans. Then we will go through operational flow of key operations that can be performed through entity beans, for example, findX(), create() and remove() methods. We will then talk about the role of container for entity beans. Given that entity beans are basically object abstraction of typically rows of relational database, the concept of primary key is important.

We will also look into the codes bean developer and client programmer has to write. We will then compare CMP 1.1 and CMP 2.0.
One of the key enhancements of CMP 2.0 is relationship handling. So we will look into how you can handle relationships between entity beans using either BMP or CMP. We will also take a look at what EJB-QL is and why it is needed.

Some of the persistence strategies are then looked into. Other miscellaneous issues we will take a look are commit options, home methods, passing references to EJBs to other EJBs, and finally error handling.
So let's understand what an entity bean is. This is a review on what we learned during J2EE overview session.
Entity Beans

- Provides **object view of persistent data**
  - Its lifetime is not related to the duration of interaction with clients
    - Lives as long as data exists in database i.e. Long lived (reason why it is called persistent)
  - Provides database independence
    - In most cases, relational databases are used
- **Shared** access among many clients
- **Transactional**
  - Entity beans are always transactional

Entity EJBs represent persistent objects; their lifetimes are not related to the duration of interaction with clients. In nearly all cases, entity EJBs are synchronized with relational databases; this is how persistence is achieved.

Entity EJBs are always shared amongst clients: a client cannot get an entity EJB to itself. (And this is one of the differences of entity beans from session beans.) Thus entity EJBs are nearly always used as a scheme for mapping relational databases into object-oriented applications. Because of this, entity beans are always transactional. That is, whenever you perform either create, remove, or modify operation, that operation needs to be performed in a transactional manner, otherwise, the database will end up having corrupted data.
Entity Beans & RDBMS

- Persistent data is typically Relational database but does not have to be (even more true with CMP 2.0)
  - Could be object database
  - Could be XML-native database
  - Could be data model of legacy application
  - Could be data model of Enterprise Information Systems (SAP, PeopleSoft, etc.)
  - Could be custom persistent technology

Now persistent data in pretty much all real-life production environments is in the form of relational database since it is the most pervasive form of persistence technology. But in theory, it does not have to be. In fact, entity beans can represent many different types of persistence technologies such as object database, XML-native database, data model of legacy applications or enterprise information systems or even custom persistent technology. The key point is that entity beans provides abstraction of these persistence technologies, that is, entity beans (CMP beans) can be deployed over these diverse set of database technologies without changing its code.
Entity Beans Usage Pattern

- Clients normally look up (find) an existing entity EJB (existing row of database table)
  - Creation means adding a row to a database table
  - Finding means finding a row in a existing database table
  - Removal means deleting a row in a database table

- Entity bean instance has unique identifier called **primary key**
  - Primary key can be any class

Whereas a client normally creates a session EJB and removes it after use, clients normally look up an existing entity EJB. It is because creation of an entity EJB corresponds to adding new data item to the application (e.g., adding a row to relational database table) and typically database tables with appropriate set of rows are usually created through different means (maybe SQL scripts).

An important feature of entity EJBs is that they have identity, that is, one can be distinguished from another. This is implemented by assigning a primary key to each instance of the EJB, where ‘primary key’ has the same meaning as it does for database management. Primary keys that identify EJBs can be of any type, including programmer-defined classes. And we will talk about this in a bit more detail later on.
Examples of Entity Beans

Customer
- Customer data has to persist, thus is maintained in the database
- Customer data remains even after application closes
- Customer data has to survive server crash
- Customer data is shared by many clients
- Each customer has unique identification such as customer number

So Customer is an example data type that you want to represent in the form of entity bean (instead of session bean). First, customer data has to persist meaning once the data is created, you want to maintain in a persistent storage so that it can be used later by many applications. That is, the customer data remains even after your application closes. The data has to survive server crash meaning even if a server crashes, the data should remain in the persistence storage. Customer data is being used by many clients. Also each customer has unique identification such as customer number.
There are two types of entity beans: CMP (Container Managed Persistence) and BMP (Bean Managed Persistence).

The key difference between CMP and BMP is that under CMP, the persistence logic is provided by the container while under BMP it is the responsibility of the bean developer himself to provide the persistence logic code. Now under CMP, the bean developer specifies the persistence requirements such as which data fields need to be maintained persistently in what is called abstract persistence scheme which is a segment of deployment descriptor. The container after reading this requirement from the deployment descriptor then performs actual persistence logic such as calling JDBC calls to relational database.
When to Use CMP vs. BMP?

**CMP entity beans**
- With CMP 2.0, there is no reason not to use CMP
- Database independence
- Higher performance (especially with CMP 2.0)
- Easy to develop and deploy (no persistence logic to write in the code)

**BMP entity beans**
- Use it when more programmatic control is desired

So when do you want to use CMP and when do you want to use BMP? Now I would like to claim, with CMP 2.0, there is no reason not to use CMP because it provides database independence (because bean does not have to be tied up with a particular database type as would be the case in BMP). And because under CMP 2.0, vendor implementations provide well-optimized implementation of persistence logic, you can safely assume their implementation will perform better than the code you write in BMP. Again because you as a bean developer does not have write persistence logic code under CMP, it is definitely easier to developer.

Now there are cases that you still want to use BMP, for example, if you want to have more fine-grained programmer control in performing database operations.
We already talked about some differences between session bean and entity bean. This slide shows the summary comparison between session bean and entity bean.

Typically session bean represents business process or business logic while entity bean represents business data. Session bean instance is associated with a particular client while entity bean instances are shared by multiple clients. The life of session bean is short-lived because again its life-cycle is tied up with one of the client while entity bean is long-lived meaning it lives as long as the data is persistently maintained in the database.

Again session bean does not survive the server crash while entity bean should survive the server crash. And entity bean because it deals with database has to be always transactional.
Why Entity Bean
(as opposed to client-driven persistence)?

Now let's talk about why you want to use entity beans as opposed to client-driven persistence. By client driven persistence, I mean client programs performs the persistence logic such as calling JDBC calls as opposed to leveraging entity beans as an abstraction of persistent data.
Without Entity Beans, Client has to handle Persistence themselves

- Issues when client code itself handles persistence logic
  - Client has to manage the state, thus not Object-Oriented: In OO philosophy, object should manage its own state
  - Hard to deal with concurrent access to the same data
  - Client code itself has to deal with transaction demarcation

- Entity Beans moves the chore of persistence management to the container

Now without entity beans, client has to handle persistence themselves. This has several disadvantages. First, client has to maintain the state manually, which does not result in object oriented design. Second, the client code also has to handle concurrency control to the same data. For example, each client code has to check if someone else is using the data or not before it uses the data. And this could be a quite a bit of chore. Third, the client code also has to deal with transaction demarcation meaning it is the client's responsibility to start a transaction, then commit or abort later on.

Using entity beans moves these chores to the container.
Now let's talk about key operations under entity beans are working. First, findX() operation. There could be multiple findX() methods in an entity bean, thus X as part of the name. For example, for a customer entity bean, you might want to have findCustomersWhoseSalaryisOver10000Dollars(), findAllCustomers(), findCustomerWhoLiveInyTexas(), and so on in addition to mandatory findByPrimaryKey().
Example: findByPrimaryKey()

```java
Object ref = context.lookup("Customer");
CustomerHome home =
    (CustomerHome)PortableRemoteObject.narrow(ref, CustomerHome.class);

// Find a single instance of EJB Object via primary key.
// (Remember EJB Object is type of Remote object.)
Customer customer = home.findByPrimaryKey(1);

String name = customer.getName();
String address = customer.getAddress();
```

This is an example code in which findByPrimaryKey() method is invoked. By the way, all findX() methods are to be invoked from EJB Home object, which is represented here as CustomerHome in this code.

So findByPrimaryKey() method let you find an EJB instance via primary key. The return value of he the findByPrimaryKey() is EJB object type. Here in this code, it is Customer type. Once you get the EJB object, you can invoke business methods of it.

(Warning: Those of you who are not familiar with the concept of EJB Home object and EJB object, I suggest you stop right here and go back to J2EE Overview presentation and understand them first. Without through understanding on what these two objects are, you will have a hard time understanding the subsequent slides.)
This picture shows the interaction between client, bean instance, and container when client calls findX() method. (By the way, this picture is quoted from Kevin Boone's Applied Enterprise Beans book.)

In this picture, we assume the client already got the EJB Home object (or Home object) via JNDI. Once the client have the EJB Home object, it can invoke findX() method of it. (Step 1) The Home object then performs ejbFindX() method internally (Step 2) to get a primary key. It then creates EJB Object and then associate the primary key with it (Step 3). Please note that step 2 can be done in implementation specific way so it is not really important. The important steps you need to understand is Step 3 and Step 4. That is Home object somehow manages to create EJB object which has a primary key associated with it (Step 3) and then returns it to the client (Step 4). Once the client has the EJB object, it can perform the business operations.

Now the steps 7 to 11 show callback methods that are invoked by the container at appropriate time. (By callback methods, I mean the methods in your bean that are invoked by the container but by the client code.)

Steps P1 to P3 are used by container when the container create instances of the bean.
findX() Operational Flow

1. Client retrieves home object via JNDI lookup() and calls findX() method
2. Home object finds the primary key
   • BMP: Home object calls ejbFindX() method of any free bean instance
   • CMP: Home object does appropriate "SQL SELECT" operation itself using EJB QL finder logic specified in deployment descriptor
3. Home object finds/creates an EJB object corresponding to the primary key
   • Container maintains one EJB object for one primary key

This slide and the following slides show the steps I just talked about in the previous slide. (read the slide)
**findX() Operational Flow**

4. Home object returns reference to EJB object (actually reference to stub of remote EJB object for the case of remote EJB object) to client

5. Client calls business methods through EJB object

6. On receipt of the first business method call, EJB object gets a bean instance from the pool

(read the slide)
findX() Operational Flow

7. EJB object now has an associated Bean instance. Container (maybe through EJB Object) calls **ejbActivate()** of the bean instance
   - This is called “activation”
   - The state of bean instance changes from “passive” to “active” state
   - In short, you can regard an “active” bean instance has an associated EJB object which itself is associated with a particular primary key

Now through step 6, EJB object has a bean instance from a pool. Please note that the bean instance is still in “passive” state.

Now when container (through EJB object) calls ejbActivate() callback method of the bean instance, the state of the bean changes from passive state to active state. And this transition of the bean state is called activation.

Now a lot of people get confused about what an “activated bean instance” is. It is basically a bean instance that has an associated EJB object which itself has a particular primary key.
findX() Operational Flow

8. EJB object (or some part of Container) calls ejbLoad() of the bean instance
   › To instruct the bean instance to make its internal state match the database
     › BMP: Uses the primary key to perform “SQL SELECT” statement inside the ejbLoad() code
     › CMP: Container performs the operation

Now in step 8, the EJB object or some part of the container calls ejbLoad() callback method of the bean instance at the appropriate time to instruct the bean instance to make its internal state to be synchronized with the database.

Under BMP, it is the ejbLoad() which contains “loading data from the database” persistence logic code typically performing “SQL SELECT” JDBC calls.

Under CMP, it is the container that performs the operation.
findX() Operational Flow

9. EJB Object calls business methods of bean instance
10. Container calls ejbStore() of the bean instance
   - BMP: “SQL UPDATE”
   - CMP: Container handles it
11. When the container is finished with the bean instance, container calls ejbPassivate()
    - This is called “Passivation”
12. Container puts instance back to the pool

In step 9, EJB objects on behalf of the client calls business methods of the bean instance.

At some appropriate time (that is, at the time container decides the most optimum time for writing data back to the database), the container calls ejbStore() callback method of the bean. Again, under BMP, this is the routine that performs “SQL UPDATE” JDBC calls while under CMP container handles it.

When the container figures it is finished with the bean instance, it will call ejbPassivate() callback routine at step 11. The bean instance then becomes in “passive” state and it gets returned to the pool at step 12.
So this sequence diagram shows findX() operation on BMP entity bean. First, the client calls findX() method of the EJB home object, which in turn calls a corresponding ejbFind() method of the bean itself. The bean instance then performs SQL SELECT to get the primary key which is then returned as a return value of the ejbFind() method to the EJB home object.

EJB home object then creates EJB object, and associate the primary key with it. The EJB object is then returned to the client as a return value of findX() method.
This is the sequence diagram of the findX() method of CMP entity bean. Here the client calls findX() method of the EJB home object. EJB home object (as part of container) then performs SQL SELECT, creates primary key, then creates EJB object. The key point here is the bean instance does not get involved.
findX() method can return either a single instance of EJB object or a collection of them. So let's talk about how a collection of EJB objects can be returned from findX() method.
Single-EJB findX() vs Multi-EJB findX()

- Depends on findX() logic, either single EJB or multi-EJB could be returned.
- Example of Single-EJB in remote interface
  - findByPrimaryKey(<primary-key>)
  - findSingleCustomer(…)
- Example of Multi-EJB in remote interface
  - findAllCustomers(100000) // with income over 100000 dollars
  - findAllBooks(1999) // which are published in 1999

Whether a single instance of EJB object or collection of them is returned depends on semantics of findX() method.

Examples of single instance return include findByPrimaryKey() or findSingleCustomer() method.

Examples of multi instance return include, for example, “find all customers whose income is over 100,000 dollars” or “find all books which were published in 1999”, and so on.
Now as you probably know it, client calls findX() method of the EJB home object, which in turn calls corresponding ejbFindX() method of the bean instance. Now as a return value of findX() method, the client gets a collection of EJB objects while as a return value of ejbFindX() method, the EJB home object gets a collection of primary keys.
Now, a word of caution. You should use multi return findX() method with care because it could be an expensive operation from a performance standpoint. It is because, in multi-return findX() operation, the container has to instantiate many EJB objects. And these EJB objects, actually the stubs, have to be marshalled and unmarshalled between the client and the container.

The better approach is to use HOME method to carry out database SELECT operations and then return the results to client as “by value” semantics.
Example: Multi-EJB findX()

Object ref = context.lookup("CustomerHome");
CustomerHome home =
   (CustomerHome)PortableRemoteObject.narrow (ref,
   CustomerHome.class);

// Find all customers with income over 100000 dollars
Collection customers = home.findAllCustomers(100000);
Iterator i = customers.iterator();
while (i.hasNext()) {
   Customer customer = (Customer) PortableRemoteObject.narrow
      (i.next(), Customer.class);
   String name = customer.getName();
   String address = customer.getAddress();
}

This is an example code of using findByPrimaryKey(<primary-key>)
method, which is mandatory findX() method that every entity bean has to
have.

In this code, EJB home object is retrieved via JNDI call. You can tell call a
findX() method, through EJB Home object, in order to get EJB object.

Once you get a EJB object, then you can invoke business methods of the
bean.
OK. We just talked about operational flow when a client calls findX() method. Now let's talk about an operational flow when a client calls create() method.
This picture shows interactions among client, bean instance, and container when a client calls create(...) method. By the way, there can be multiple create(...) methods depending on the number parameters on the create(...) call.

Here client calls a create(...) method of the Home object (step 1), which in turn asks the bean instance pool manager to allocate a bean instance (step 2), then calls ejbCreate(...) method of the bean instance (step 3). The bean instance might be talking to database to create a row in a database table (step 3.5). The ejbCreate(...) then returns a primary key to the Home object. The Home object then creates EJB object with the returned primary key (step 4). The Home object then calls ejbPostCreate() method of the bean instance (step 5). The Home object then returns EJB object to the client (step 6). And the client then invoked business methods through EJB object (step 7).
create(..) Operational Flow

1. Client calls create(..) method on Home object
2. Home object (or some other part of Container) gets a free bean instance
3. Container calls ejbCreate(..) of the bean instance passing same arguments
   - Bean instance set the internal state with the passed arguments
     - BMP: Creates a new row via “SQL INSERT”, returns a Primary Key
     - CMP: Container does this after calling ejbCreate(..), returns null

So this slide and the few slides following this explains what I just said in the previous slide in a bit more detail. (please read the slide)
create(..) Operational Flow

4. Container creates a new EJB object and sets the primary key
5. Container calls `ejbPostCreate()` on the bean instance
6. Home object returns a reference to the EJB object to the client
7. Client calls business methods

(please read the slide)
Why Do we need ejbPostCreate()?

- Is used to do some initialization (by the bean instance) that require the existence of EJB object
  - Container cannot create EJB object until after ejbCreate() is returned
  - Container needs primary key in order to create EJB object
  - Primary key can be obtained only after ejbCreate() is done
- Used when Entity bean has association with other Entity beans and they have to pass references to themselves during creation period

Now let's talk a bit why there is ejbPostCreate() method call. (By the way, this is not terribly important for you to move on so if you want to skip this slide, that is fine. But understanding this might help you to understand the whole picture a bit better.)

In a nutshell, ejbPostCreate() is used to do some initialization by the bean instance that require the existence of EJB object. Now intuition says that you want to perform this initialization as part of ejbCreate() method instead of an extra method like ejbPostCreate(). Now let's assume that some initialization might involve passing a reference to a bean instance itself in the form of EJB object, for example, initializing another bean instance passing the EJB object as a parameter. Now the problem is the container cannot create EJB object until after ejbCreate() is returned. And it is because the container needs primary key and the primary key can be obtained only after ejbCreate() is done.

So EJB design team came up with a work-around, ejbPostCreate() method. Most of the time, this method contains nothing.
This sequence diagram shows the creation of CMP bean instance. Here the client first calls create(..) method of the EJB home object. EJB home object, in turn, internally calls ejbCreate() method of the bean instance. A new row is inserted into the database after this. EJB home object then creates EJB object and invokes ejbPostCreate() method.

Again under CMP, all these are performed by the container (i.e. EJB home object) not by the bean itself.
This picture shows the create(..) operation in a picture. Again the client calls create(..) method of the EJB home object. The key point in this picture is that it is the container that performs “SQL INSERT” not the bean instance itself.
Creation of an BMP Bean Instance

This sequence diagram shows the creation of BMP based bean. Here client calls create(..) method of the EJB home object, which in turn calls ejbCreate() method of the bean instance. Now under BMP, it is the bean instance that actually creates a row in a database table.
This picture also shows what I just said in the previous slide. There it is the bean instance that performs “SQL INSERT” to the database.
Now let's talk about remove() operation.
This picture shows interaction among client, bean instance, and container when a client calls remove() method. First, a client calls either remove() method to EJB Object (step 1) or remove (<primary key>) to the Home object (step 2). In the former, the EJB object in turn calls Home object with primary key. The Home object then calls ejbRemove() of the bean instance (step 4).
**remove() Operational Flow**

1. Client calls `remove()` on the EJB Object or `remove(<primary-key>)` on the Home object
2. EJB Object delegates the call to Home object
   - This step is not necessary if client calls Home object directly
3. Home object obtains an free bean instance
remove() Operational Flow

4. Home object calls `ejbRemove()` on the bean instance
   - BMP: performs “SQL DELETE” inside the code
   - CMP: Container already have deleted the data and any dependent objects

5. Bean instance is returned to the pool

(read the slide)
Removal of an BMP-based Entity Bean

This sequence diagram shows the removal of the CMP based entity bean.
Removal of an CMP-based Entity Bean

This sequence diagram shows how removal of CMP-based entity bean works. When a client calls remove() method, it is the container (via EJB object) that actually calls ejbRemove() method of the bean class and then removes an entry from a database.
Now let's talk about operational flow when a client calls business method.
Calling Business Methods of Entity bean

This picture shows sequence diagram of client calling business methods of the EJB object. EJB object in turn calls business methods implementations of the bean instance.
Now let's talk about life-cycle of an entity bean instance. I said bean instance here to make it clear that we are talking about bean instances.
create(..) & remove() & findX() of Entity Bean

- create(..) operation of Entity bean is **not the same** as instantiating a Java object
  - create(..) operation of Entity bean creates a corresponding piece of persistent data
  - remove() operation of Entity bean removes the corresponding piece of persistent data
- Clients of Entity beans typically use **findX()** instead of create(..) since the persistent data (i.e.) might have been typically created via different means

Now I want to make sure you understand the difference between calling create(..) method of an entity bean and creating an ordinary object instance.

Creating an entity bean creates a corresponding piece of persistence data, for example, it will create a row in a database table. Removing an entity bean does the opposite. It removes a row from a database table.

As for findX() method, clients of entity beans typically use findX() instead of create(..) method since the persistent data might have been created via different means.
This slide compares home interface of session bean against the home interface of entity beans.

First create() method of session bean does not take any argument while for entity beans, create(...) methods take arguments. In fact, depending on what combination of arguments you provide, there can be multiple create(...) methods for entity beans.

Again create() method of session bean does not insert a new row to a database table while create() method of entity beans exactly does that.

Session bean does not have findX() method while entity beans can have multiple findX() methods.

Again remove() method of session bean does not remove a row from a database table while remove() method of entity bean does that.
This picture illustrates the life-cycle of an entity bean,
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.
Now let's talk about the role of the container for entity beans.
What Container does for Entity Beans

- Bean instance pool management
- Life cycle management of bean instances
- Transaction coordination
- Persistence management
  - Figuring out when to synchronize with database (for both BMP and CMP)
  - Providing optimized access to database (CMP)
- Security
  - Access control

This slide shows the things container does for the entity beans. First, container manages the bean pool. It also manages the life-cycles of the bean instances. That is, the containers creates and removes the bean instances at appropriate time. The container also manages transaction coordination.

As for persistence, the container knows when is the right time to synchronize the bean with the backend database. For it called ejbLoad() and ejbStore() methods at right time. As for the CMP, the container also performs the persistence logic as well.

The container also handles the security, for example, depending on how the security attributes are set for each bean, the container will enforce the access control.

Now let's talk about some of these in a bit more detail.
Life cycle and Pool management of Bean instances

- Container knows when is the best time to passivate and activate bean instances
  - based on the resource usage of the system
- Container knows when is the best time to create and remove bean instances

In terms of life-cycle management, the container knows when is the best time to passivate and activate the bean instance depending on the resource availability.

Container also knows when is the best time to create and remove bean instances from the pool.
Container knows when is the right time to synchronize state of bean instances with persistent data storage
- Transaction semantics specified by bean developer/deployer are used
- Container can use various information it maintains to optimize the synchronization with the persistent data storage
  - dirty writing, caching, optimistic locking, lazy loading
- It calls ejbLoad() and ejbStore() at right time

(Read the slide)
Now let's talk a little bit on object view of database.
This picture shows how entity beans can be mapped into tables of relational database. Please note that there does not have to be one-to-one mapping between EJB and relational database table. In fact, in real-life situations, it is rather difficult to achieve this one-to-one mapping.

In this example, we have Account EJB which is represented by Account database table. Each row of the table is then represented by Account EJB bean instances.
Entity beans and Persistent Data type

- Entity beans can represent any persistent data
  - typically relational database
  - object database
  - data model of a legacy application
  - data model of EIS (PeopleSoft, SAP, etc.)
- For BMP, programmers have to code persistent data type specific calls to access the data
  - JDBC for relational database
- For CMP, containers handle the access logic to the data

As was mentioned before, entity beans present any persistent data. In most of the cases, the persistent data are maintained in the form of relational database. But it can be any database technology.

Under BMP, the bean developer has to make database technology specific calls in order to perform persistence logic operations while under CMP the container handles the persistence logic (access logic) to the various database technology. Please note that however in reality this is also constrained by vendor support of different database technologies.
Primary Keys in Entity Beans

Understanding primary keys are important in understanding entity beans. So let's talk a little bit on entity beans.
What is Primary Key?

- Any piece of data that uniquely and unequivocally identify a particular entity bean
  - If we know the primary key, we can always find the rest of the data for the entity bean
  - Entity beans with same primary keys are considered identical
- Container uses primary key to manage Entity bean instances and EJB objects
How is Primary Key represented?

- Represented by Java class
  - In many cases, standard Java classes such as `java.lang.String` or `java.lang.Integer` are used
    - CustomerID is `java.lang.Integer` type
  - If standard Java classes are not sufficient, then custom Java class can be used
    - If there is no ready-to-use CustomerID field, that uniquely identifies an instance, a combination of fields can constitute a custom primary key
Now when you have to create your own custom primary key, there are certain set of rules you have to follow. (read the slide for the rules)
This is an example of primary key class. As you can see, the class has to be serializable. And it has its own hashCode() and equals() methods.
Now let's talk about programming APIs.
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 for the Account Bean. 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.
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.
* HomeHandle getHomeHandle() - Obtain a handle for the home object.
* Void remove(Handle 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.
* java.lang.Object getPrimaryKey() - Obtain the primary key of the EJB™ object.
* Boolean isIdentical(EJBOBJECT obj) - Test if a given EJB™ object is identical to the invoked EJB™ object.
* void remove() - Remove the EJB™ 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 SessionBean interface is implemented by every session enterprise Bean class. The container uses the SessionBean methods to notify the enterprise Bean instances of the instance's life cycle events.

Method Summary:

void ejbActivate() - The activate method is called when the instance is activated from its "passive" state.

void ejbPassivate() - The passivate method is called before the instance enters the "passive" state.

void ejbRemove() - A container invokes this method before it ends the life of the session object.

void setSessionContext(SessionContext ctx) - Set the associated session context.
The SessionContext interface
A container provides the session bean instances with a SessionContext, which gives the session bean instance access to the instance’s context maintained by the container. This give information about the bean’s home object, current transaction information, security role information… The SessionContext interface has the following methods:
The getEJBObject method returns the session bean’s remote interface.
The getEJBHome method returns the session bean’s home interface.
The getCallerPrincipal method returns the java.security.Principal that identifies the invoker of the bean instance’s EJB™ object.
The isCallerInRole method tests if the session bean instance’s caller has a particular role.
The setRollbackOnly method allows the instance to mark the current transaction such that the only outcome of the transaction is a rollback. Only instances of a session bean with container-managed transaction demarcation can use this method.
The getRollbackOnly method allows the instance to test if the current transaction has been marked for rollback. Only instances of a session bean with container-managed transaction demarcation can use this method.
The getUserTransaction method returns the javax.transaction.UserTransaction interface. The instance can use this interface to demarcate transactions and to obtain transaction status. Only instances of a session bean with bean-managed transaction demarcation can use this method.
Now we talked about already several times on home interface, remote interface and bean class itself. So you probably have a decent understanding on these already. Here we will talk about how they are related to each other from programming API standpoint.
This picture shows how your entity bean class (AccountBean in this example) implements EntityBean interface and which methods of remote interface (Account interface in this example) and home interface (AccountHome in this example) should have corresponding methods.

For example, the create() and findX() methods of the home interface have corresponding methods in bean class in the form of ejbCreate() and ejbFindX() methods.

The business methods of the remote interface should be also implemented in the bean class. And callback methods such as ejbLoad() and ejbStore() also need to be in the bean class.
Now let's talk about the code bean developer has to write in implementing an entity bean.
First, you have to write home interface, remote interface, bean class itself, and then deployment descriptor.

As we talked about before, home interface defines findX() and create() methods as well as Home methods (we have not talked about Home methods yet) while remote interface defines business methods.
Now let's talk about home interface.
Home interface (local or remote)

- Must extend `javax.ejb.EJBHome` (remote) or `javax.ejb.EJBLocalHome` (local) directly or indirectly
- Contains methods that allow a client to create an entity bean or beans
  - Multiple `create(…)` methods
  - Multiple `findX(…)` methods
- Contains Home methods

Home interface can be either local or remote. Your home interface must extend `javax.ejb.EJBHome` or `javax.ejb.EJBLocalHome` interface directly or indirectly.

Home interface contains methods that allow a client to create or find an entity bean or entity beans via `create(…)` or `findX(…)` methods.

Home method can also contain Home methods. (We have not talked about Home methods yet.)
For each `create(...)` method of Home interface, your bean class must have matching `ejbCreate(...)` method.

`Create()` method returns EJB object which is the type of remote interface.
FindX() method of the home interface must have matching ejbFindX(...) methods. By the way, findByPrimaryKey() method is mandatory.

As with Create(..) methods, findX() methods also return EJB object or a collection of EJB objects.
Now let's talk about Home methods of Home interface. The names of Home methods can be anything, let's call it X(), but the bean class must have matching ejbHomeX() methods.
This slide shows an example of Home interface. The same home interface is shared by both BMP and CMP.

Here we have one create(..) method and several findX() methods and a single Home method.
Since we just talked about home methods that can be defined as part of home interface, let's talk about Home methods a bit here.
Home Methods

- Introduced in EJB 2.0
- Methods that are not tied with a bean instance
  - Conceptually similar with Static method of Java class
- Methods can be specified in Home interface
  - Implementation of these methods in Bean class should start “ejbHome”
- Available only for Entity beans

Home methods are introduced in EJB 2.0. The difference between home methods and other methods is that home methods are not tied up with any particular bean instance. In that sense, home methods are like Static methods of Java programming language.

As was mentioned, home methods are defined in home interface and the actual implementation in bean class of these home methods should take the form of ejbHomeX() where X is the home method defined in home interface.

Home methods are available only for entity beans.
When to Use Home Methods?

- When you want to return a number of rows in a database table
- In EJB 1.1, two ways to implement this
  - Use session bean instead of Entity bean
  - Use dummy Entity bean that does not map to database
    - findByPrimaryKey("")
- In EJB 2.0, use Home method
  - Any instance in the pool can be used
  - No database synchronization is needed

So home methods are used when you need to make a method call which does not have to be tied up with a particular bean instance. One example could be when you want to return number of rows in a database table.

In EJB 1.1, a kludge is to use dummy entity bean instance by providing "" primary key or you have to use session bean instead.

In EJB 2.0, you can use home method. By using home method, any instance in the bean pool can be used.
Remote Interface (or Local Interface)

OK, we just talked about home interface. Now let’s talk about remote interface (or Local interface).

By the way, I know the names of these interfaces are really confusing. Remote interface is for remote beans and Local interface is for local beans. And they contain business methods.
Remote interface (or Local interface)

- Must extend `javax.ejb.EJBObject` (remote interface) or `javax.ejb.EJBLocalObject` (local interface) directly or indirectly.

- Contains business methods
  - For remote interface, arguments and return values must be valid RMI types
  - For local interface, arguments and return values do not have to be valid RMI types

- Bean class must have matching business methods

Remote interface must extend `javax.ejb.EJBObject` while local interface must extend `javax.ejb.EJBLocalObject` directly or indirectly.

Remote and local interface contain business methods and bean class must have matching business methods with actual implementation code.
Three types of Business Methods

- Methods that manipulate properties
- Methods that manipulate relationships (associations)
- Methods that manipulate properties through value objects
Example: Remote interface

```java
import javax.ejb.EJBObject;
import java.rmi.RemoteException;
import java.math.BigDecimal;

public interface SavingsAccount extends EJBObject {
    // Business methods that manipulate properties
    public void debit(BigDecimal amount) throws InsufficientBalanceException, RemoteException;
    public void credit(BigDecimal amount)
        throws RemoteException;
    public String getFirstName()
        throws RemoteException;
    public String getLastName()
        throws RemoteException;
    public BigDecimal getBalance()
        throws RemoteException;
    // Business methods that manipulate relationships (this example has none)
    // Business methods that manipulate properties via value objects (this example has none)
}
```

This is an example of remote interface. In this example, we have only the first kind of business methods, manipulating properties.
Now let's talk about how to write bean class.
Bean Class

- Implement create(..), findX(..), and Home methods defined in Home interface
  - ejbCreate(..), ejbFindX(..), ejbHomeX(..)
- Implement business methods defined in Remote interface (or Local interface)
- Implement methods of java.ejb.EntityBean interface
  - ejbLoad() & ejbStore()
  - ejbActivate() & ejbPassivate()
  - ejbRemove()
  - setEntityContext(EntityContext ctx) & unsetEntityContext()

As was talked about before, bean class implements various methods defined in home interface and remote interface. It also implements callback methods of the EntityBean interface.
Bean Class

- BMP and CMP implement methods of Bean class differently
  - In BMP, code contains persistence logic (database access)
  - In CMP, no persistence logic is required in the code, instead persistence logic is specified in deployment descriptor in declarative fashion
    - easier to develop beans
    - faster since container vendors can use various optimization techniques (especially in CMP 2.0)
    - can have `ejbSelect(..)` methods

(read the slide)
**ejbCreate(..) methods**

- Performs
  - Inserts the entity state into the database in BMP
  - Initializes the instance variables
  - Returns the primary key
- There could be multiple **ejbCreate(..) methods**
  - must have matching create(..) methods in Home interface
- Must have matching **ejbPostCreate() method**

(read the slide)
Example: BMP ejbCreate(..)

```java
public String ejbCreate(String id, String firstName,
String lastName, BigDecimal balance)
throws CreateException {
    if (balance.signum() == -1) {  
        throw new CreateException("A negative initial balance is not allowed.");
    }
    // Perform database insert a row operation here.
    try {
        insertRow(id, firstName, lastName, balance);
        // catch (Exception ex) {
        //    throw new EJBException("ejbCreate: "+
        //     ex.getMessage());
        //}  
        // Initialize instance variables
        this.id = id;
        this.firstName = firstName; this.lastName = lastName;
        this.balance = balance;
        // Return primary key
        return id;
    }

This is an example code of BMP ejbCreate(). As you can see, you actually insert a row to the database table in your code. (In this code, insertRow() is a utility class that performs “SQL INSERT”.)
**ejbPostCreate() methods**

- Invoked after `ejbCreate()` is invoked
  - After Primary key and EJB object have been assigned
- `ejbPostCreate()` method can invoke the `getPrimaryKey()` and `getEJBObject()` methods of the `EntityContext` interface
  - if it needs to pass EJB Object as a parameter
- Typically empty

(Read the slide)
This is an example code of ejbPostCreate() method. As was mentioned, in most cases, this method contains nothing.
ejbFindX(...) methods

- **BMP**
  - Performs SQL SELECT operations inside the code

- **CMP**
  - Container performs SQL SELECT operations itself based on the EJB-QL logic declared in the deployment descriptor

The `ejbFindX()` method. Under BMP, this code performs SQL SELECT itself. Under CMP, it is the container that performs the SQL SELECT operation based on EJB-QL script specified in the deployment descriptor.
Example: BMP ejbFindX(..), Single EJB

```java
public String ejbFindByPrimaryKey(String primaryKey)
throws FinderException {
    boolean result;
    // Perform database operation here
    try {
        result = selectByPrimaryKey(primaryKey);
    } catch (Exception ex) {
        throw new EJBException("ejbFindByPrimaryKey: " + ex.getMessage());
    }
    // Return primary key
    if (result) {
        return primaryKey;
    } else {
        throw new ObjectNotFoundException("Row for id " + primaryKey + " not found.");
    }
}
```

This is example code of ejbFindX() under BMP. Under BMP, you have to perform finder logic yourself in your code.
Example: BMP ejbFindX(..), Multi-EJB

```java
public Collection ejbFindByLastName(String lastName) throws FinderException {
    Collection result;
    // Perform database operation here
    try {
        result = selectByLastName(lastName);
    } catch (Exception ex) {
        throw new EJBException("ejbFindByLastName " +
                                ex.getMessage());
    }
    // Return Collection of primary keys
    return result;
}
```

This is an example code in which collection of primary keys are returned as return value. Again you have to perform the finder logic yourself in your code.
ejbHomeX(...) methods

- Contains the business logic that is not tied with a particular bean instance
  - Normal business methods are always tied with a particular bean instance (with unique identity)
  - Cannot access persistent instances nor persistent relationships
  - Example:
    - return total value of purchase orders

As was mentioned, a Home method X() defined in home interface has to have corresponding ejbHomeX() in bean class.
Example: BMP ejbHomeX(..)

```java
public void ejbHomeChargeForLowBalance (
    BigDecimal minimumBalance, BigDecimal charge)
    throws InsufficientBalanceException {
    try {
        SavingsAccountHome home = 
            (SavingsAccountHome)context.getEJBHome();
        Collection c = home.findInRange(new BigDecimal("0.00"),
            minimumBalance.subtract(new BigDecimal("0.01")));
        Iterator i = c.iterator();
        while (i.hasNext()) {
            SavingsAccount account = (SavingsAccount)i.next();
            if (account.getBalance().compareTo(charge) == 1) {
                account.debit(charge);
            }
        }
    } catch (Exception ex) {
        throw new EJBException("ejbHomeChargeForLowBalance: "+ ex.getLocalizedMessage());
    }
}
```

Typically, a home method locates a collection of bean instances and invokes business methods as it iterates through the collection. This approach is taken by the ejbHomeChargeForLowBalance() method of the SavingsAccountBean class.

The ejbHomeChargeForLowBalance() method applies a service charge to all savings accounts with balances less than a specified amount. The method locates these accounts by invoking the findInRange method. As it iterates through the collection of SavingsAccount instances, the ejbHomeChargeForLowBalance() method checks the balance and invokes the debit business method. Here is the source code of the ejbHomeChargeForLowBalance method:
Business methods

- Contains business logic
- Implementation of business methods defined in Remote interface (or Local interface)

Bean class also contains implementation of the business methods that are defined in the remote interface (or local interface).
Example: Business Methods

```java
public void debit(BigDecimal amount) throws InsufficientBalanceException {
    if (balance.compareTo(amount) == -1) {
        throw new InsufficientBalanceException();
    }
    balance = balance.subtract(amount);
}

public void credit(BigDecimal amount) {
    balance = balance.add(amount);
}

public String getFirstName() {
    return firstName;
}

public String getLastName() {
    return lastName;
}

public BigDecimal getBalance() {
    return balance;
}
```

This is an example of business methods. Each business method contains whatever business logic for the application. In this example code, we have debit() and credit() business methods in which balance is adjusted.
**ejbLoad() & ejbStore() methods**

- For BMP, ejbLoad() contains code for retrieving data from the database while ejbStore() contains code for writing data to the database.
- For CMP, they contain nothing.

(Read the slide)
Now let's talk about ejbLoad() and ejbStore() and how they are related to business methods and transaction.

EjbLoad() and ejbStore() methods always execute under a transaction context, which is determined by transaction attribute of each business method. If a business methods is running under a transaction, which should be the case in most cases, the container invokes ejbLoad() before it calls business methods. Also the container calls ejbStore() immediately after business method get executed.
This is an example code of ejbLoad() and ejbStore() of BMP. In BMP, the ejbLoad() and ejbStore() have to contain code that performs “loading data from the database” and “storing data to the database” respectively.
One of the confusions people have on entity beans is `ejbActivate()` and `ejbPassivate()`. Again, under CMP, programmer really do not have to be concerned about these two methods.

When `ejbActivate()` method of a bean instance is called by the container, the bean instance is in a state in which it is associated with a specific EJB object. When `ejbPassivate()` method of the bean instance is called by the container, the bean instance is no longer associated with a EJB object.
Example: BMP ejbActivate() & ejbPassivate()

public void ejbActivate() {
    id = (String)context.getPrimaryKey();
}

public void ejbPassivate() {
    id = null;
}
**ejbRemove() method**

- A container invokes this method before it removes the EJB object that is currently associated with the instance.
- This method is invoked when a client invokes a remove() operation on the enterprise Bean's home interface or the EJB object's remote interface.
- This method transitions the Bean instance from the ready state to the pool of available instances.
Example: BMP ejbRemove(..)

```java
public void ejbRemove() {
    // remove a row from database
    try {
        deleteRow(id);
    } catch (Exception ex) {
        throw new EJBException("ejbRemove: "+ex.getMessage());
    }
}
```
setEntityContext(EntityContext ctx) & unsetEntityContext()

- Container calls setEntityContext() after it creates Bean instance
- Container calls unsetEntityContext() after it removes a Bean instance
  - This is the last method Container calls to the bean instance
  - The instance will be garbage-collected
Example: BMP setEntityContext(.) & unsetEntityContext()

```java
public void setEntityContext(EntityContext context) {
    this.context = context;
    try {
        makeConnection();
    } catch (Exception ex) {
        throw new EJBException("Unable to connect to database. " + ex.getMessage());
    }
}

public void unsetEntityContext() {
    context = null;
    try {
        con.close();
    } catch (SQLException ex) {
        throw new EJBException("unsetEntityContext: " + ex.getMessage());
    }
}
```
Now let's talk about `ejbSelectX()` method. The concept of selector is introduced in CMP 2.0. The key difference of selector and finder, which are represented by `ejbSelectX()` methods and `ejbFindX()` methods in the bean class respectively, is that selector methods are used only internally by EJB. That is, selector methods are not exposed to the client. So `ejbSelectX()` methods have no corresponding `SelectX()` methods in the remote (or local) interface.

Now as is the case for persistent fields and relationships, `ejbSelectX()` methods are defined as abstract methods. And as is the case for finder, the selector logic is expressed in the deployment descriptor in declarative fashion.

Another significant difference between finder and selector methods is that selector methods can return not only entity beans but also dependent value objects.
Example: CMP 2.0 ejbSelectX(..)

```java
public abstract class PlayerBean implements EntityBean {
    private EntityContext context;

    // Access methods for persistent fields
    public abstract String getPlayerId();
    public abstract void setPlayerId(String id);
    ...

    // Access methods for relationship fields
    public abstract Collection getTeams();
    public abstract void setTeams(Collection teams);

    // Select methods
    public abstract Collection ejbSelectLeagues(LocalPlayer player)
                               throws FinderException;
    public abstract Collection ejbSelectSports(LocalPlayer player)
                               throws FinderException;
```

This is CMP 2.0 ejbSelectX() example code. Please note that the ejbSelectX() methods are abstract methods. Again, you specify the selector logic in the form of EJB-QL in the deployment descriptor as we will see in the following slide. And it is the container who actually implements these abstract methods by taking a look at the selector logic from the deployment descriptor.
Example: CMP 2.0 `ejbSelectX(..)`

Deployment Descriptor

```xml
<entity>
  <display-name>PlayerEJB</display-name>
  <ejb-name>PlayerEJB</ejb-name>
  ...
  <query>
    <description></description>
    <query-method>
      <method-name> ejbSelectLeagues </method-name>
      <method-params>
        <method-param>team.LocalPlayer</method-param>
      </method-params>
    </query-method>
    <result-type-mapping>Local</result-type-mapping>
    <ejb-ql>select distinct t.league from Player p, in (p.teams) as t where p = ?1</ejb-ql>
  </query>
  ...
</entity>
```

This slideshow deployment descriptor that specifies a selector logic in the form of EJB-QL script.
Now let's take a look at how client code looks like.
So far, we have talked about how you create entity beans in the server side. Now let's talk about the client side.

One key point you have to understand is that the implementation form of the entity bean on the server side - that is, whether it is implemented a CMP 2.0, CMP 1.0, or BMP - should not have any bearing on how client code is written. As far as client is concerned, the only thing the client cares about is two interfaces, home interface and remote interface. This is the reason why the server side implementation can change or evolve without affecting the compatibility of the client programs.
This is the first slide of the 4-slide client code example. The client code works in the following sequence:

- Get EJB home object via JNDI
- From the EJB home object, get EJB object
- Once you get an EJB object which is the type of remote interface, you can then invoke business methods
- After invoking business methods, you can remove the EJB object
In this code, you create another EJB object by providing a new set of arguments and then perform business functions.
Example: Client code

c = home.findInRange(new BigDecimal("20.00"),
    new BigDecimal("99.00"));
for(i.c.iterator();
    SavingsAccount account = (SavingsAccount)i.next();
    String id = (String)account.getPrimaryKey();
    BigDecimal amount = account.getBalance();
    System.out.println(id + " : " + amount);
}

SavingsAccount pete = home.create("904", "Pete", "Carlson",
    new BigDecimal("5.00"));
SavingsAccount sally = home.create("905", "Sally", "Fortney",
    new BigDecimal("8.00"));
home.chargeForLowBalance(new BigDecimal("10.00"),
    new BigDecimal("1.00"));
Example: Client code

```java
BigDecimal reducedAmount = pete.getBalance();
System.out.println(reducedAmount);
reducedAmount = sally.getBalance();
System.out.println(reducedAmount);
System.exit(0);
}
}
```
OK. Now let's talk about the limitations of CMP 1.1. We will also talk about CMP 2.0 later on.
1. Application logic and schema mapping are entangled
2. Inefficiencies in using instance variables
3. Tight coupling of client view of persistence with EJB view
4. No control over granularity of client-EJB integration

These are the issues with CMP 1.1. And we will talk about each of these in a bit more detail in the following slides.
Issue 1: Application logic and Schema mapping are entangled

- Container does not know the relationship between instance variables and the fields of database tables
  - Instance variable names are not the same column names of the table
- Container does not know how to perform finder logic of findX()
  - Semantical meaning of findCustomers(income) is not known to Container
- Container specific coding results non-portable code

Issue number 1. In CMP 1.1, the application logic and schema mappings are entangled. That is, the application logic (that is, programmer) has to have intimate knowledge of schemas.
Solutions to Issue 1

- Mapping of instance variables to database tables (Schema mapping) can be separated from application logic
- Schema mapping can be done at deployment time
- Finder logic should be separated from schema mapping
  - Schema mapping can be done without affecting the finder logic
- Finder logic should be defined in a portable fashion
One very visible deficiency of CMP 1.1 is that under CMP 1.1, instance variables are directly declared as public variables in a bean class. This has a serious implication in that container has no effective means (that is, there is no interception point) to determine which instance variables will be read and written by business methods since business methods can directly access those instance variables. It also means that the container has to read and write the instance variables from and to the database regardless of the fact that the variables are in fact used or not by the business methods.

The bottom line is under CMP 1.1, the container vendors are not given any opportunities for optimization in terms of loading and writing data from and to the database.
Solutions to Issue 2

- Allow container to load values of instance variables only when they are needed
  - It is called "lazy-loading"
- Allow container to write only the instance variables that were actually changed during a transaction
  - It is called "selective writing"
- Allow container to load and write persistent relationship only when needed
  - Lazy-loading and selective writing for relationships

The solution to this problem is to allow container to load values of instance variables only when they are needed (lazy loading) and to allow container to write only the instance variables that were modified (selective writing).

Another significant performance boost could be to allow container to load and write persistent relationship only when they are needed.

Again as we will see later on, these solutions are provided as part of CMP 2.0.
Now another issue people raised on CMP 1.1 is that, under CMP 1.1, there is no support of internal finder logic. For example, many EJB bean developers have a need to perform finder logic internally yet they do not want to expose it to the client application. For example, let's say there is a customer entity bean and one of the business method of the bean has to “find all the products the customer has bought in the past month”. Now the business method might have to call a finder logic something like “find all the product beans” internally in order to fulfill the business logic. Under CMP 1.1, since there is no internally callable finder method, the only option for the bean developer is expose “find all the product beans” finder method whether there is a need or not.

Another issue under CMP 1.1 is that it is not possible to perform internal finder logic over dependent objects. Not all persistent data need to be exposed as entity beans. Instead some persistent data should be abstracted out as Java classes, which are called dependent objects or dependent value classes instead of entity beans. (Please note that every EJB bean has inherent overhead over plain Java classes.)

As we will talk about later on, CMP 2.0 came up with internal finder scheme called Selector. Using selector, bean developers now can perform internal finder logic over not only entity beans but also dependent objects.
Solutions to Issue 3

- Allow EJB to locate persistent associations using internal-usage only finder method
  - while not exposing the persistent associations directly to clients
  - De-couple client view of finder logic from that of internal EJB's

So the solution to the issue #2 is to allow EJB beans to locate persistent associations using internal usage only finder method - under CMP 2.0, this is named as Selector - while not exposing the association directly to the clients.
Issue 4: No control over granularity of client-EJB integration

- In CMP 1.1, all persistent objects have to be represented as EJB's unless the developer wants to use helper classes and manage the persistence in code
  - If one object is dependent on another (dependent object), it does not have to be EJB and it does not have to be exposed to clients
  - Requiring dependent objects to be in EJB results in unnecessary overhead
Solutions to Issue 4

- Define a standard way of managing the relationship between entity bean and its dependent objects
A few words on CMP 1.1

- Due to these limitations, CMP 1.1 is not considered to be useful, thus is not used widely
  - Many complaints on EJB’s (un)usefulness are in fact related to the immaturity of CMP 1.1
- Yet, the concept of Container Managed Persistence (CMP) is a good one
- Interim solution products such as Toplink or Cocobase give developers “Good CMP like” features but in a proprietary way

(read the slide)
OK. We looked into the limitations of CMP 1.1. Now let's talk about CMP 2.0, which corrected all the problems that plagued CMP 1.1.
So CMP 2.0 addresses all the limitations of CMP 1.1.
Now let's talk about characteristics of CMP 2.0.

Under CMP 2.0, you as a bean developer write your entity bean as an abstract class. And it is the container who actually creates concrete implementation of your entity bean.

Unlike the entity bean of previous version, you don't declare any fields inside your bean. Instead you define abstract getter and setter methods for those fields. And it is the container who actually implements these abstract getter and setter methods probably by maintaining instance variables for these fields internally.

Now how do you specify the contract between your bean and the container? That is, how do you tell container which fields are to be concretely implemented? The contract is called Abstract persistence schema, which is part of deployment descriptor.
Why CMP 2.0 Entity Bean?

- Enables separation of data handling logic from persistence logic
- Data handling logic is coded in the bean by bean provider
- Persistence logic is provided by container
  - Container handles JDBC calls
  - Container can employ various optimization techniques
    - lazy loading, dirty checking and update, caching, optimistic concurrency control

Now let's talk about value proposition of CMP 2.0.

First in CMP 2.0, the data handling logic has been separated from persistence logic. And you, as a bean developer, are concerned only about data handling logic while persistence logic is provided by the container. What that means is that the container now invokes JDBC calls or other persistence logic code. Now because container has the total control of handling persistence logic, it can employ various optimization techniques such as lazy loading, dirty checking, caching and optimistic concurrency control and so on.
This picture shows the class hierarchy of CMP 2.0 based entity bean. The key point in this picture is that you as a bean developer create abstract CMP 2.0 entity bean while the container actually implements a subclass extending your abstract entity bean and implement the abstract methods.
Example: CMP 2.0 Entity Bean

```java
public abstract class AccountBean implements EntityBean{
  // No fields

  // Abstract getter/setter methods
  public abstract String getOwnerName();
  public abstract void setOwnerName(String ownerName);

  // Data logic methods using the above getter/setter methods
  ...

  // Business logic methods
  ...

  // EJB required methods follow
  ...
}
```

So this is an example of CMP 2.0 entity bean code. Please note that there are no fields defined here. Instead there are getter and setter methods. For example, we have defined getOwnerName and setOwnerName methods as abstract methods.

And you also have data handling logic and business logic in your code.
Now let's talk about CMP 2.0 abstract persistence schema in a bit more detail.

Abstract persistence schema is declaratively expressed in the deployment descriptor. And it is created by bean provider as an instruction to the container in terms of how to implement persistence logic. That is, the container inspects the abstract persistence schema and generates subclass accordingly.

I mentioned that container now implements abstract getter and setter methods of the fields. How does the container know which are the fields whose getter and setter methods are to be concretely implemented in the subclass it generates. It is called `<cmp-field>` element in the abstract persistence schema.
Example: Abstract Persistence Schema

... 

<cmp-version>2.x</cmp-version>

<abstract-schema-name>AccountBean</abstract-schema-name>

<cmp-field>
  <field-name>accountId</field-name>
</cmp-field>

<cmp-field>
  <field-name>ownerName</field-name>
</cmp-field>

...

So this is an example of abstract persistence schema that contains a couple of <cmp-field> elements. Here the accountId and ownerName are the fields whose getter and setter methods are defined in your bean code and actual persistence logic gets implemented by the container.
### BMP versus CMP 2.0

<table>
<thead>
<tr>
<th>BMP</th>
<th>CMP 2.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class definition</td>
<td>Abstract</td>
</tr>
<tr>
<td>Class definition</td>
<td>Generated by tools</td>
</tr>
<tr>
<td>Class definition</td>
<td>Not abstract</td>
</tr>
<tr>
<td>Persistent state &amp; Access methods for persistent and relation fields</td>
<td>Represented by virtual persistent fields in deployment descriptor</td>
</tr>
<tr>
<td>FindByPrimaryKey</td>
<td>Required</td>
</tr>
<tr>
<td>Customized finder methods</td>
<td>Handled by container</td>
</tr>
<tr>
<td>Select methods</td>
<td>Handled by container but developer must defined EJB QL queries</td>
</tr>
<tr>
<td>Return value of ejbCreate()</td>
<td>Should be null</td>
</tr>
<tr>
<td></td>
<td>Must be primary key</td>
</tr>
</tbody>
</table>

- BMP: Coded by developers
- CMP 2.0: Coded by developers
- CMP 2.0: None
- BMP: Coded by developers
- BMP: Coded by developers
- BMP: Coded by developers
Now let's talk about EJB-QL (EJB Query Language)
What is EJB QL?

- **Portable, abstract definition** of finder and selector methods for entity beans and dependent objects
  - Declaratively expressed in deployment descriptor
  - SQL like language
  - Compiled to native language (i.e. SQL) of data store during runtime by container
    - Container can optimize the implementation
- Enables querying through navigation
  - Difference from SQL

*So what is EJB QL? It is portable definition of finder methods for entity beans with container managed persistence. That is, query statements written EJB Query Language (EJB QL) specify how the container should implement the finder methods defined in the home interface of an entity bean.

*And as we talked about, the syntax of EJB QL is rich enough to allow a bean provider to specify complex query operation. Again specifying complex query operation in a portable way was not possible in EJB 1.1.

*The query operation is expressed in declarative language which makes it portable over various persistent managers and data store. The declarative expression of this query operation, which we call EJB QL statements, are compiled and implemented by the persistent manager.

*EJB QL is a specification language that can be compiled to a target language, such as SQL, of a persistent store used by a persistence manager. This allows the responsibility for the execution of queries to be shifted to the native language facilities provided for the persistent store (e.g., RDBMS), instead of requiring queries to be executed directly on the persistent manager’s representation of the entity beans’ state. As a result, query methods are optimizable as well as portable.
So why EJB-QL? Without EJB-QL, you have to code the finder logic yourself in your bean, which is a lot of work in the first place and which is not going to be portable, for example, suppose you are going to use object database instead of relational database, the finder logic has to be recoded. And finally the it is hard to do any optimization.

With EJB-QL, it is now the container's responsibility to implement the finder logic and you specify the finder logic in a declarative fashion in the deployment descriptor. So with EJB-QL, you don't have do coding for finder logic. And since the finder logic is expressed in EJB-QL script, it is portable over different persistence technologies. And since the finder logic gets implemented by the container, the container has a freedom to optimize finder logic.
Example: EJB-QL

- Find all accounts whose balance is over X dollars
  - X is a parameter
- EJB-QL script

```java
SELECT OBJECT(a)
FROM Account AS a
WHERE a.balance > ? 1
```
Passivation & Activation in Entity Bean
Passivation of CMP-based Entity Bean

Business method

client
EJBHome
EJBOBJECT
container
bean
stateful
database

Business method

- ejbStore()
- ejbPassivate()
- ejbActivate()
- ejbLoad()

Business method

- Get container-managed fields
- Update entity state in database
- ejbPassivate()

Business method

- Read entity state from database
- Set container-managed fields
- ejbActivate

Business method
Resources
Resources Used

- EJB Codecamp by Carol McDonald (Sun Microsystems, Inc.), 2001 [4]
Live your life with Passion!