Table of Contents

Head First EJB™ By Bert Bates, Kathy Sierra ISBN: 0596005717 Publisher: O'Reilly
Print Publication Date: 2003/10/01

5 entity bean intro

Entities are Persistent

...and then I said, "You want a piece of me? Go ahead -- take your best shot buddy!" He didn't know he was messing with an **entity** bean. So he threw an exception, then he crashed the server, but I'm still here! I won't go down that easy, no siree. As long as I'm in the database, I'll just keep coming back, so do your worst!

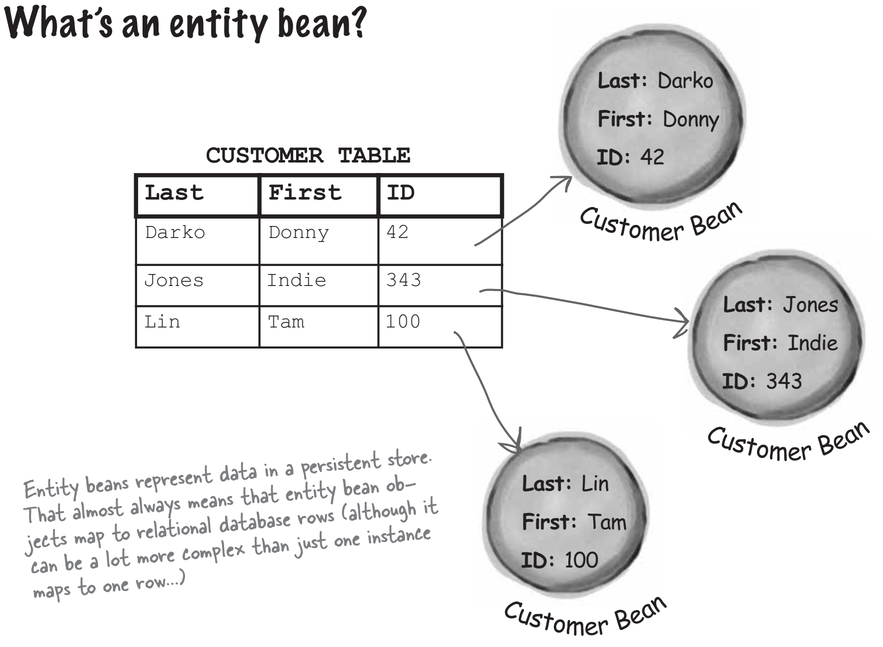
Entity beans persist. Entity beans exist. Entity beans *are*. They are object representations of something in an ***underlying persistent store***. (Think: ***database,*** because most entity beans represent something from a relational database.) If you have a Customer entity bean, then one bean might represent the entity Tyler Durden, ID #343, while another is the entity Donny Darko, ID #42. Two beans, representing two real *entities*. An entity bean is simply a *realization* of something that already exists in a persistent store. Something that already *is*. And these suckers are hard to kill! As long as the data is in the database, it can keep coming back in the form of an entity bean.

OBJECTIVES

Entity Beans

	Official:	*What it really means:*
5.1	Identify correct and incorrect statements or examples about the client view of an entity bean's local and Remote home interface, including the code used to locate an entity bean's home interface, and the home interface methods provided to the client.	You have to know the rules for an entity bean home interface, and especially how they're different from a session bean. You must know that you're not required to have a create() method, but that you must have at least one finder method, findByPrimaryKey(). You have to know that the return type of single-entity finders and create methods must always be the component interface type, but that multi-entity finders must return a Collection. You must know that single-entity finders throw an ObjectNotFoundException if there's no match in the database, but that multi-entity finders always return a Collection, even if its empty. You also have to know that home business methods aren't required to return the component interface— they can return anything that can be legally passed. You must also know that if an entity is deleted from the database, it can no longer be represented as an entity bean, but that if an entity bean *instance* dies (through an exception or even a server crash), the entity itself persists. As long as the entity is in the database, it can be represented by an entity bean.
5.2 **5.3**	Identify correct and incorrect statements or examples about the client view of an entity bean's local component interface (EJBLocalObject) and Remote component interface (EJBObject)	The rules for how you define things in an entity component interface are identical to the rules for a session bean component interface. But in objective 5.4, we'll look at how the methods *behave* differently.
5.4	Identify the use, syntax, and behavior of the following entity bean home method types for CMP: finder methods, create methods, remove methods, and home business methods.	The behavior of remove() is profoundly different for an entity bean, and you must understand that calling remove() really means, "delete this entity from the underlying persistent store!" You must know that calling create() on an entity bean means, "insert a new row in the database" (OK, technically it means, "create a new entity in the underlying persistent store.") You must know that if an entity is deleted from the underlying persistent store, the entity bean 'dies', even though the bean instance goes back to the pool.

What's an entity bean?


CUSTOMER TABLE
Last: Darko
First: Donny
ID: 42
Customer Bean
Last: Jones
First: Indie
ID: 343
Customer Bean
Last: Lin
First: Tam
ID: 100
Customer Bean
Entity beans represent data in a persistent store. That almost always means that entity bean objects map to relational database rows (although it can be a lot more complex than just one instance maps to one row...)

CUSTOMER TABLE

Last	First	ID
Darko	Donny	42
Jones	Indie	343
Lin	Tam	100

An entity bean is an object-oriented way of looking at in a persistent store. The spec says "persistent store", and doesn't specify what that persistent store must be. It could be anything that satisfies the requirements of 'persistence' including a relational database, an object database, or even something as lame and inefficient as storing serialized objects to files. But in most real-world scenarios, we're talking a relational database. And that means an individual row in a table maps to a unique entity bean. The object-to-relational (OR) mapping could be a lot more complex than a simple one row equals one bean scenario, but we'll get into that in the next chapter.

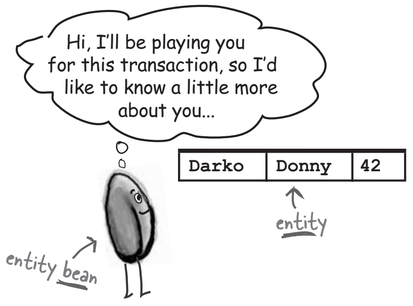
Entity beans are data objects. They are...things. *Nouns.* As opposed to session beans which are processes. *Verbs.* Entity beans might represent things like people, products, orders, bookings, inventory, animals...*things.* Entity beans would *not* represent things like credit card verification, advisor service, order submission, membership registration... *processes.*

Of course, in virtually all well-designed EJB applications, entity beans will be combined with session beans, where the client interacts with the *process* session bean, and the session bean uses entity beans when it needs *data* as part of the process.

Entities vs. Entity Beans

An *entity* is the real thing that the entity *bean* represents. It doesn't work in reverse! The entity does not represent the entity bean. That might seem like a subtle point, but it's not. The difference is in knowing which one is *real*, and which one is simply a *view* of the real thing.

In other words, if you delete the real *entity* (the *thing*) in the database, the entity *bean* disintegrates. Poof. But if the entity *bean* dies, as an instance on the heap, it doesn't kill the real entity in the database. (By the way, we're now going to say 'database' instead of 'underlying persistent store', but you and I both know that these two are not necessarily synonymous, and that the 'underlying persistent store' need not be—but usually is—a database. There, we've said it.)


Hi, I'll be playing you for this transaction, so I'd like to know a little more about you...
Darko
Donny
42
entity
entity bean

So you won't have the right perspective if you think of the entity bean as some data object that happens to be *backed up* to persistent storage. In nearly all applications, the vast majority of entities used in the app are not *realized* as entity beans at any given time. Instead, most entity beans become a representation of an underlying entity *only when that particular entity is needed in the application logic.*

For example, imagine your database has 10,000 customers. But at any given time, clients (through business processes in session beans) are using only 200 of those customers. In that case, you probably have only 200 entity beans actively representing customer entities. The rest of the 9,800 customer entities are just sitting there in the database, waiting for a client process to need them.

Think of entity *beans* as actors. In the scenario we just described, you have 200 actors, each playing a different role of someone in the database. But the other 9,800 entities in the database—the other customers—do not have anyone playing them.

But as soon as a client comes along and tries to get one of those other 9,800, say, to change that customer's address, or check its credit limit, an actor (entity bean) will be chosen to play the role of that selected customer.

there are no Dumb Questions

Q: Just so I've got this straight... your entire database doesn't get loaded in as entity beans, right? I mean, that would be insane.

A: You're right, at least about the first part. No, your entire database is NOT loaded in as entity beans. Think of entity beans as just-in-time representations of only the entities in the database that are actively needed by client business processes.

But does that mean it would be insane to load them all in? Not necessarily. While it would certainly up your resource requirements, especially if the data in the underlying database continues to grow, just think of how FAST it would be.

Although there is nothing in the spec that requires this, a vendor can choose to let you configure the app in such a way that it does pre-load all the entities in the database into beans. That gives you, essentially, a lightening-fast in-memory database, that still synchronizes itself with the real underlying store (you'll see how in a moment).

And to take it a step further, if your server gives you a way to tell it that your entity beans are your only access to the database, then the server can even eliminate the synchronization and just keep the database in memory, saving only what it needs as a backup in case of a crash.

Entity beans are an OO way of looking at data in a persistent store.

An entity is a real, uniquely-identifiable thing that exists somewhere outside of EJB, and the entity bean's job is simply to BECOME an OO view of that real, persistent entity.

The entity bean cannot exist without the real entity, but the real entity can exist without the bean.

An entity bean is NOT the real entity--it's just a representation or "realization" of the entity.

there are no Dumb Questions

Q: Why do you even NEED entity beans? Why not just go straight to the database from a session bean?

A: You can go from a session bean directly to the database. Heck, you can always go from the *client* to the database, but then we're back to the old non-scalable client-server two-tier architecture, and we all know why *that* is usually a bad idea (doesn't scale, business logic is in the client, hard to maintain, etc.)

If you use entity beans instead of direct calls to the database, you get to take advantage of all the Container's services, including the ability to wrap several database trips in one transaction. But as you'll see, one of the biggest benefits of using entity beans is that the Container automatically synchronizes between the database and the entity bean.

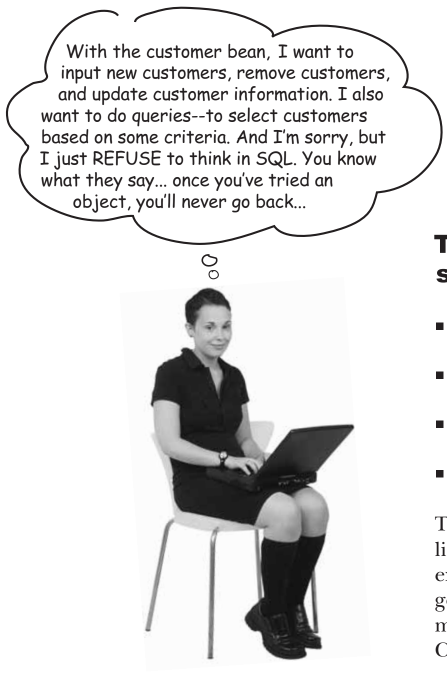
But the single most compelling reason for entity beans is that they take you from the relational world to the object world. In other words, you get to stay with objects all the way down, in your app, rather than mapping back and forth in your code. And if you're using CMP, which you almost certainly will be (for reasons that will become obvious a little later), you won't have a shred of SQL in your bean code. In fact, with CMP entity beans, you get to pretend that your entire database exists solely as objects on the heap.

Sure, maybe you're comfortable with JDBC, but what about the rest of the team? And let's face it—thinking in OO makes a lot more sense than having to shift between OO and entity code.

Relax **You don't need to know SQL or JDBC for the exam.**

Well, for the most part. You DO need to know EJB-QL, which is in some ways similar to SQL, but you don't need to know anything about SQL itself. And while the exam doesn't cover JDBC, you DO need to know that a bean lookup a javax.sql.DataSource, from which the bean can get a java.sql.Connection. You'll learn more about DataSource lookups in the chapter on EJB environment.

Entity beans from the client's point of view


With the customer bean, I want to input new customers, remove customers, and update customer information. I also want to do queries--to select customers based on some criteria. And I'm sorry, but I just REFUSE to think in SQL. You know what they say... once you've tried an object, you'll never go back...

The client wants to do database stuff, but in a Java OO way.

- Make a new entity (an SQL INSERT)
- Delete an entity (an SQL DELETE)
- Update entity state (an SQL UPDATE)
- Search/query on entities (an SQL SELECT)

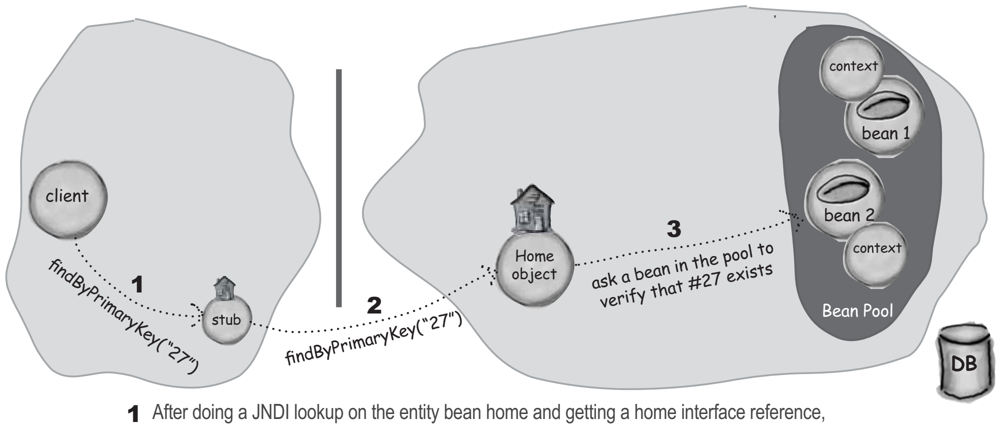
The client interface for an entity bean is a little different from that of a session bean. For example, when a session bean client wants to get a bean, so that it can call the bean's business methods, the client calls create() and the Container allocates a new EJB object.

But what if a client wants to use an existing entity bean? For example, what if the client doesn't want just some random entity, but wants the EJB object of a *specific* entity, say, Bart Simpson # 12? In that case, a create() won't work. The client doesn't want a *new* entity, but wants a reference to an *existing* entity. So as you'll see in a minute, the client interface for an entity bean adds (and *must* have) one or more *finder* methods.

The next two pages are high-level pictures of how entity beans are created (insert), and found (select). The scenarios in these pictures will be filled in with a lot more detail as we go through this chapter, but for now you can relax.

Entity bean overview

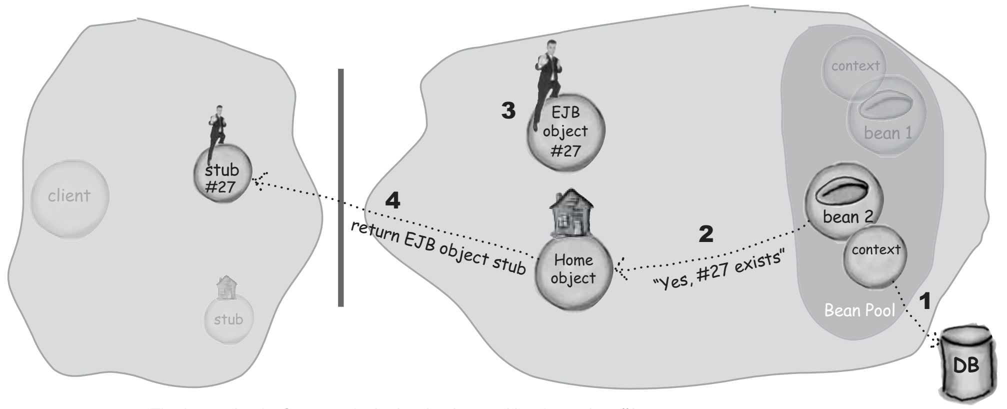
Scenario: client wants a reference to an existing entity



1 After doing a JNDI lookup on the entity bean home and getting a home interface reference, the client calls findByPrimaryKey("27") on the home stub.

2 The findByPrimaryKey("27") method invocation is passed to the home object.

3 The Container asks a bean in the pool to verify that #27 exists in the database.



1 The bean checks for an entity in the database with primary key #27

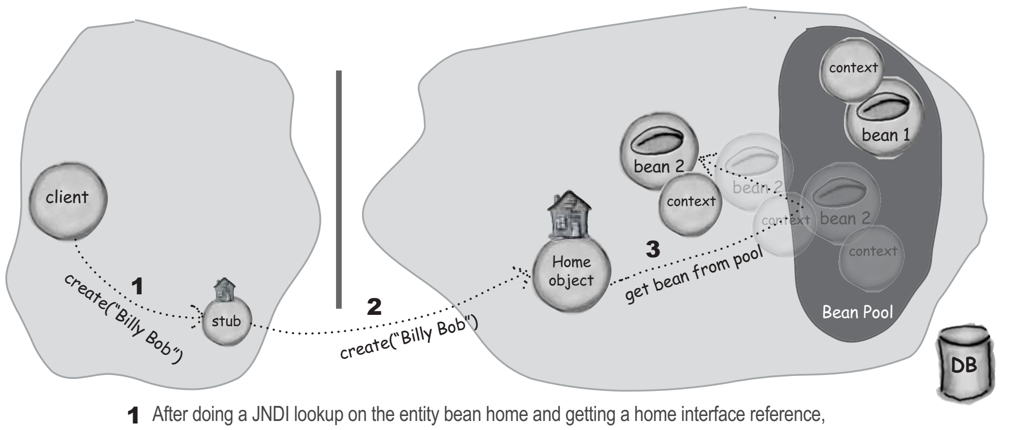
2 The bean tells the home that #27 is in the database

3 The Container makes or finds an EJB object for #27 (there might already be one)

4 The Container returns the stub for #27

Entity bean overview

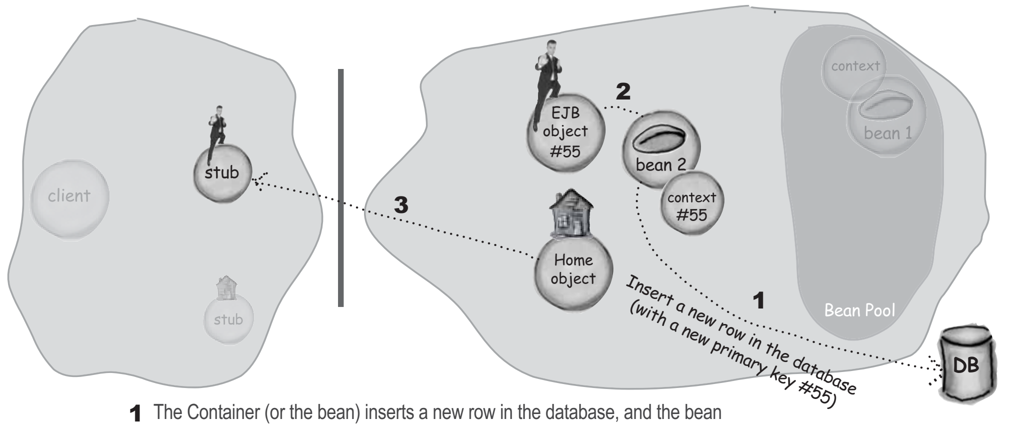
Scenario: client wants to create a new entity



1 After doing a JNDI lookup on the entity bean home and getting a home interface reference, the client calls create("Billy Bob") on the home stub.

2 The create("Billy Bob") method invocation is passed to the home object.

3 The Container pulls a bean from the pool for this bean type (could be any random bean in the pool). Notice that the bean already has its own context that sticks with the bean.



1 The Container (or the bean) inserts a new row in the database, and the bean generates a new primary key.

2 The bean is linked to an EJB object, and both the context and EJB object get the new primary key.

3 The Container returns a stub for the newly-created entity (Billy Bob, #55). If the client's only goal was to insert the new row, the client might not even care about the returned stub.

A very simple Customer entity bean

```
// package and imports here
public class CustomerBean implements EntityBean {
```

implement the EntityBean interface (instead of SessionBean)

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

The entity bean's state includes the fields that map to columns in the Customer database table, in this case-- first name, last name, and ID (primary key)

```
   private EntityContext context;
```

Your context is even MORE important when you're an entity bean instead of a session bean

```
   public String ejbCreate(String last, String first) {
     lastName = last;
     firstName = first;
     primaryKey = this.getPK();
     // DB INSERT
     return primaryKey;
   }
```

Doesn't this look a lot like a constructor? We take the client's args and assign them to instance variables that represent the entity's persistent state (i.e. columns in the database table). The ejbCreate() returns the primary key -- don't worry about it right now.

```
   public String getLastName() {
      return lastName;
   }
   public void setLastName(String name) {
      lastName = name;
   }
   public String getFirstName() {
      return lastName;
   }
   public void setFirstName(String name) {
      firstName = name;
   }
```

Plain old Java getters and setters for the persistent fields. The magic, as you'll see in a minute, is that the result of these setters will ultimately lead to a database update!

```
   public void ejbActivate() { }
   public void ejbPassivate() { }
   public void ejbRemove() {// DELETE }
```

Three container callbacks from EntityBean, that were also in SessionBean. Except... they have VERY VERY different meanings, as you'll see later in this chapter.

```
   public void setEntityContext(EntityContext ctx) {
      context = ctx;
   }
```

this is just like setSessionContext, except for an EntityContext. But with entity beans, the context is usually WAY more important than it is for session beans.

```
   public void unsetEntityContext() { }
   public void ejbLoad() {// SELECT }
   public void ejbStore() {// UPDATE }
```

these are three NEW container callbacks from the EntityBean interface (not found in SessionBean). They're IMPORTANT!

```
   private String getPK() {
     return ""+ (int) (Math.random() * 42);
   }
```

World's worst primary key algorithm. Normally we'd use client-supplied info or perhaps the database's auto-generated key.

```
   public String ejbFindPrimaryKey(String pk) {// SELECT, return pk }
   public Collection ejbFindByCity(String city) {// SELECT, return collection of keys }
}
```

(ignore these last two methods for now)

An entity bean's client view

Our Customer bean isn't finished, so don't get too attached to it! And in fact, the code is an example of bean-managed persistence (BMP) which we really won't be using in this book. We'll talk a little about BMP, and a *lot* about its much more popular counterpart—*container*-managed persistence (CMP) in this chapter and the next. For now though, we'll focus on the client view of an entity bean, and this simple bean is just to get you started looking at entity bean code.

Given that a Customer bean represents a Customer entity (i.e. a real customer) in the underlying database, what behaviors should the entity bean have? In other words, what kinds of things might the client want to do with either a single Customer or multiple Customers?

Things you'd do with a database record! The things we mentioned earlier including make a new Customer, delete a Customer, update a Customer's fields (columns), and query/search on the Customer database.

BRAIN POWER

Think about the following operations, and figure out which of the two client interfaces (component or home), is better suited for each operation. Keep in mind that the rules for entity bean interfaces might be different from session bean interfaces. If you think both interfaces are appropriate, check them both. (We've done the first one for you.)

✓ home	component	**Make a new customer**
home	component	**Change an existing customer's phone number**
home	component	**Find all the customers in Pleasantville**
home	component	**Delete all customers previously declared 'inactive'**
home	component	**Delete a specific customer**
home	component	**Get the street address of a specific customer**

Entity bean Remote component interface

An entity bean's component interface is just like a session bean's—it has business methods, and it extends javax.ejb.EJBObject. That means a client can see the methods you've declared in your component interface, as well as the methods from EJBObject (getHandle(), remove(), etc.).

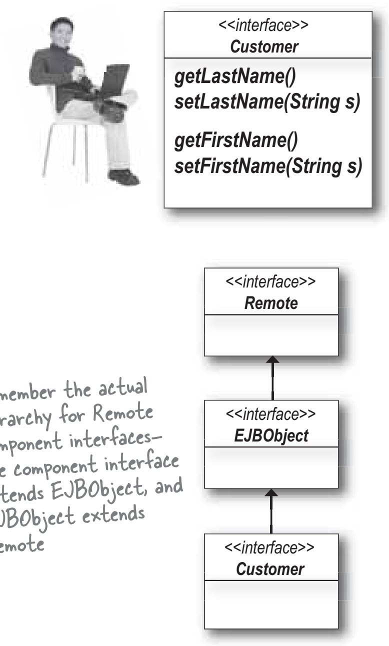
But what kinds of business methods go in the component interface?

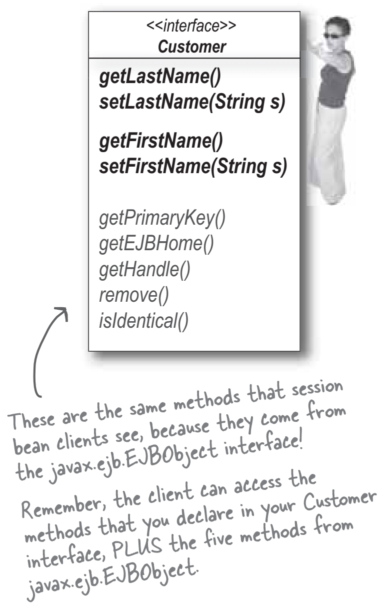
The methods related to a single entity!

When the client has a reference to an entity bean (which means a reference to the bean's EJB object, of course), the client has a reference to a single, specific entity. Fred Flintstone, #999. Marge Simpson, #728. Roy Rodgers, #1957.

So what might a client want to do with a reference to, say, Marge Simpson? Delete *her*, change *her* last name, get *her* handle, or get *her* home so that the client can get references to other customers. Keep in mind that our simple Customer bean isn't very useful yet, with methods to get or set only the Customer name. Later, we'll build it out.

What YOU write: **What the CLIENT sees:**





<<interface>>
Customer
getLastName()
setLastName(String s)
getFirstName()
setFirstName(String s)
getPrimaryKey()
getEJBHome()
getHandle()
remove()
isIdentical()
These are the same methods that session bean clients see, because they come from the javax.ejb.EJBObject interface!
Remember, the client can access the methods that you declare in your Customer interface, PLUS the five methods from javax.ejb.EJBObject.

Entity bean Remote component interface

```
package headfirst;

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

public interface Customer extends EJBObject {

    public String getLastName() throws RemoteException;
    public void setLastName(String lastName) throws RemoteException;

    public String getFirstName() throws RemoteException;
    public void setFirstName(String firstName) throws RemoteException;

}
```

Rules for the Remote component interface

1. Import javax.ejb.* and java.rmi.RemoteException
2. Extend javax.ejb.EJBObject
3. Declare one or more business methods, that throw a RemoteException
   - Arguments and return types must be RMI-IIOP compatible (Serializable, primitive, Remote, or arrays or collections of any of those)
   - You can have overloaded methods
   - Each method must declare a RemoteException
   - You can declare your own application exceptions, but they must NOT be runtime exceptions (in other words, they must be compiler-checked exceptions—subclasses of Exception but not subclasses of RuntimeException)
   - Methods can have arbitrary names, as long as they don't begin with "ejb".

Entity bean Remote home interface

How it's different from a session bean home interface:

1. You're more likely to ***find*** an *existing* entity than ***create*** a *new* one

   As a client, you'll probably spend a lot more time using references to *existing* customers, than you'll spend creating *new* customers. Whether you're updating a specific customer or doing a batch operation on many customers, you'll need home interface code that lets you select customers, more than you'll need creation methods. In fact, the create() method is optional for entity beans! (Because you might have a policy that says new entries in the database must be done directly through a database admin tool, for example.) But you're required to put at least one *finder* method *f*or an entity bean home—you can have as many finders as you like, but you must have findByPrimaryKey(String primaryKey) in every entity home. (Which means it might be the only method declared in the home interface.)

2. You might want to do queries that involve more than one entity

   Since session beans represent process, it doesn't make sense to, say, get multiple instances of the same process. But with entity beans, you might want to do the same things you'd do on a database table, like find all the customers who live in Helsinki and enjoy surfing.

What YOU write:

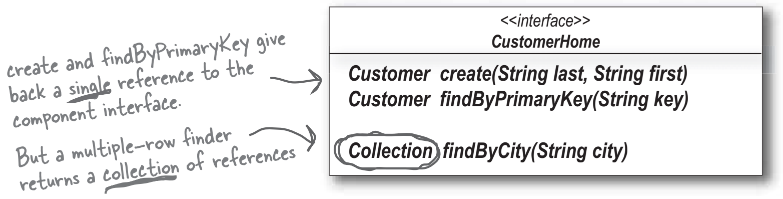
```
<<interface>>
CustomerHome
----------------------------------
create(String last, String first)
findByPrimaryKey(String key)
findByCity(String city)
```

What the CLIENT sees:

```
<<interface>>
CustomerHome
----------------------------------
create(String last, String first)
findByPrimaryKey(String key)
findByCity(String city)

getEJBMetaData()
getHomeHandle()
remove(Handle h)
remove(Object key)
```

An entity bean client can see the same four methods a session bean client can see, because both session and entity bean home interfaces must extend javax.ejb.EJBHome. Entity beans get to use BOTH home remove() methods, while session beans can use only the one that takes a Handle.

What does the client really want from an entity bean home?


create and findByPrimaryKey give back a single reference to the component interface.
But a multiple-row finder returns a collection of references
<<interface>>
CustomerHome
Customer create(String last, String first)
Customer findByPrimaryKey(String key)
Collection findByCity(String city)

With session beans, that was easy—a reference to the component interface. And that's exactly what the create() methods have to give back.

With entity beans, it's the same for create() methods—they must give back a reference to the component interface, in this case the component interface for the entity just created.

But what if you want to find an existing entity bean instead of making a new one? That's what the mandatory findByPrimaryKey() method is for, and it, too, must give back a reference to the component interface for the bean matching that key.

But what if there isn't a matching entity? If there's no entity with that key in the database, the client gets a javax.ejb.ObjectNotFoundException. So the return type of findByPrimaryKey() is always the same as it is for create(), the component interface for that bean type. (And of course, the rules for session bean client interfaces applies here as well—a Remote home interface must give back the Remote component interface, and the local home interface must give back the local component interface.)

This still leaves us with a method that *cannot* return the component interface: a multiple-entity finder, like our findByCity() method. Well, the client's goal doesn't change with multiple-entity finders; the client still wants a reference to the component interface, only this time it might be a whole *collection* of them. One for every customer entity in the city named in the method's argument.

Note: a client will *not* get an exception if a multi-entity finder can't find any matches! Instead, the client will still get a Collection, but it will simply be empty. A Collection with no elements. Only single-entity finders throw exceptions when nothing matches the find criteria.

The create and finder methods in an entity bean home always give back the bean's component interface.

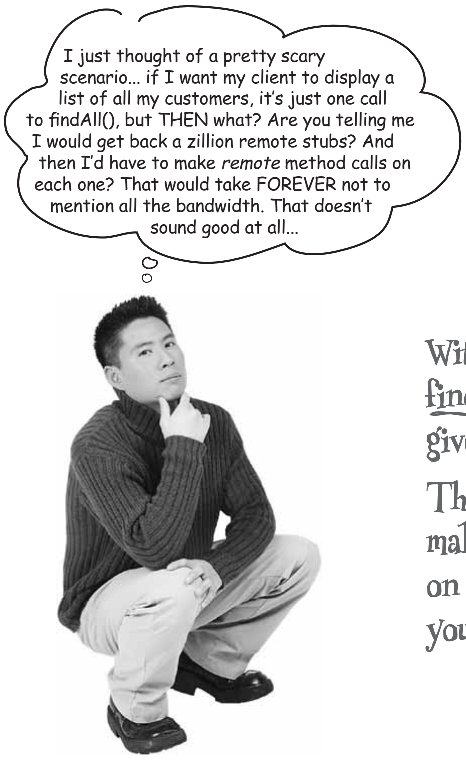
For create() and findByPrimaryKey(), the client gets a reference to one EJB object.

For multiple-entity finders, the client might get a whole PILE of references to EJB objects—one for each bean that matches the query.

Entity bean Remote home interface

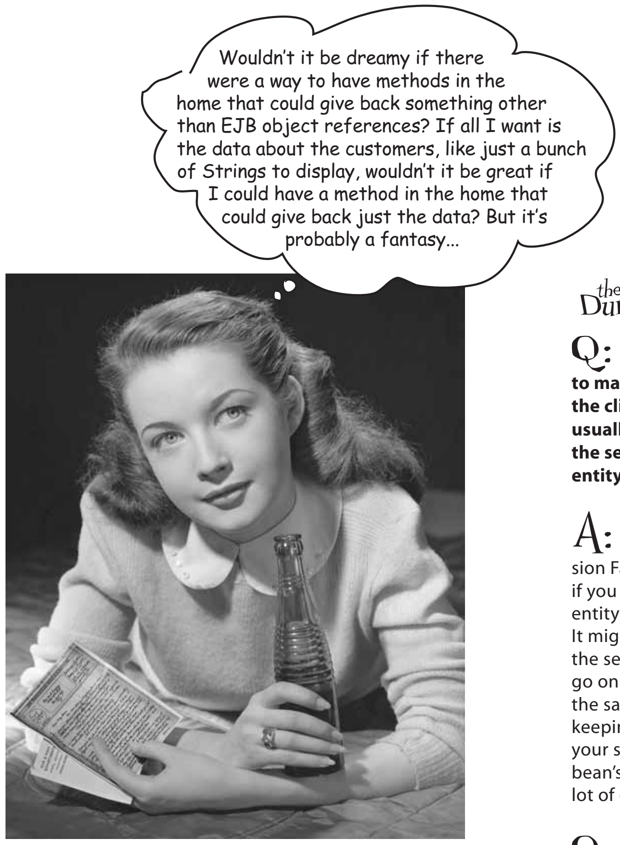
```
package headfirst;

import javax.ejb.*;
import java.rmi.RemoteException;
import java.util.Collection;

public interface CustomerHome extends EJBHome {


   public Customer create(String last, String first) throws CreateException, RemoteException;


   public Customer findByPrimaryKey(String key) throws FinderException, RemoteException;


   public Collection findByCity(String city) throws FinderException, RemoteException;
}
```

When finders have a dark side...


I just thought of a pretty scary scenario... if I want my client to display a list of all my customers, it's just one call to findAll(), but THEN what? Are you telling me I would get back a zillion remote stubs? And then I'd have to make *remote* method calls on each one? That would take FOREVER not to mention all the bandwidth. That doesn't sound good at all...

With a Remote interface, finder and create methods give back Remote stubs.

That means you have to make remote method calls on each one to get the data you want.


Wouldn't it be dreamy if there were a way to have methods in the home that could give back something other than EJB object references? If all I want is the data about the customers, like just a bunch of Strings to display, wouldn't it be great if I could have a method in the home that could give back just the data? But it's probably a fantasy...

there are no Dumb Questions

Q: Wait... I thought it was bad design to make all those method calls from the client ANYWAY. Wouldn't the client usually go through a session bean? And the session bean would then talk to the entity bean?

A: Kind of. You're thinking of the Session Facade J2EE design pattern. But even if you do put a session bean in front of an entity bean, *the session bean is still a client.* It might be a lot more efficient, because the session bean might not have as far to go on the network (and might even be on the same server as the entity). But if you're keeping location-independence, then your session bean is still using the entity bean's Remote interfaces, so there's still a lot of overhead.

Q: Couldn't you just have a business method in the component interface that returns data? Like, return a collection of Strings?

A: You're getting warmer. Yes, that's the way you might have done it in EJB 1.1. But that's kludgey, because you have to first get a reference to *some* customer, just so you can ask *that* customer to give you back the data for all customers.

Home business methods to the rescue

That's right... business methods aren't just for the component interface, when you're talking about entity beans. As of EJB 2.0, an entity bean home can have methods that—*drum roll*—don't have to return component interfaces! Home business methods can return anything (with the one restriction, of course, that Remote home methods return values that are RMI-IIOP compliant).

Home business methods are great for batch operations, or for query methods where the client doesn't need—or want—EJB object references, but simply wants the entity's data (in other words, the data for one or more of the entity's persistent fields). For example, we might put a home business method in the Customer bean like, getAllCustomerInfo(), that returns a collection of Strings, with whatever pieces of data you've decided make up the customer's info. Better yet, you can send back a collection of CustomerInfo objects, where CustomerInfo is a class that simply holds the data (and getters) for the Customer's persistent state. That way, the client can make local calls to get the data it needs out of the CustomerInfo objects, without having those calls be remote calls on the component interface.

A CustomerInfo class is an example of a Value Object class which is, in a nutshell, just a class with getters (and possibly setters, depending on the design) representing the entity's persistent fields). And it, too, has a dark side—the data starts to become stale the moment after the Value Object is created.

We could tell you now, but then we'd be robbing you of such a valuable opportunity to apply a little neural effort. So for now, why don't *you* think of why sending back CustomerInfo objects, that the client could then interrogate (i.e. call methods on) at its leisure, could have a downside. We'll use Value Objects a lot, but you have to be aware of the tradeoffs when choosing between using a home finder method that returns EJB object references (especially when the references are Remote) vs. a home *business* method that returns Value Objects.

Home business methods can return something other than EJB object references! They're perfect for queries where the client just wants the entity data, not references to the entities themselves.

They're also great for batch operations, or anything else you might want to do with more than one specific entity, when you don't want to return references to the component interface.

Rules for the Remote home interface

1. Import javax.ejb.* and java.rmi.RemoteException
2. Extend javax.ejb.EJBHome
3. Declare (optionally) one or more create() methods, which MUST return the Remote component interface, and declare both a RemoteException and a CreateException. Each create() method must begin with the prefix "create".
4. Declare the findByPrimaryKey() method, which MUST return the Remote component interface, and declare both a RemoteException and a FinderException
5. Declare (optionally) one or more other finder methods, which MUST return either the Remote component interface (for single-entity finders), or java.util.Collection (for multiple-entity finders). All finders must declare both a RemoteException and a FinderException
6. Declare one or more home business methods
   - Arguments and return types must be RMI-IIOP compatible (Serializable, primitive, Remote, or arrays or collections of any of those)
   - You can have overloaded methods
   - Each method must declare a RemoteException
   - You can declare your own application exceptions, but they must NOT be runtime exceptions (in other words, they must be compiler-checked exceptions—subclasses of Exception but not subclasses of RuntimeException)
   - Methods can have arbitrary names, as long as they don't begin with "create", "find", or "remove".

Head First EJB™ By Bert Bates, Kathy Sierra ISBN: 0596005717 Publisher: O'Reilly
Print Publication Date: 2003/10/01

Sharpen your pencil

For the four database operations (SQL commands) a client might want to do with an entity bean, list the methods in the bean's interface(s) that are related to those database operations. No, you don't have to know SQL, but you definitely have to understand the *implications* of the four database operations, and you must know how they correspond to methods in the bean class.

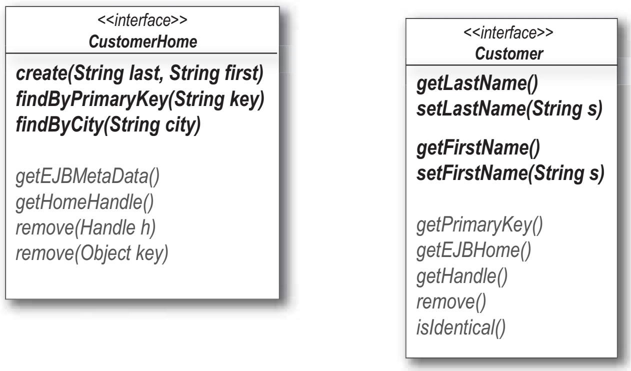
From the list of the methods in the interfaces, fill in the method or methods that correspond with the database operation.

INSERT:

DELETE:

UPDATE:

SELECT:



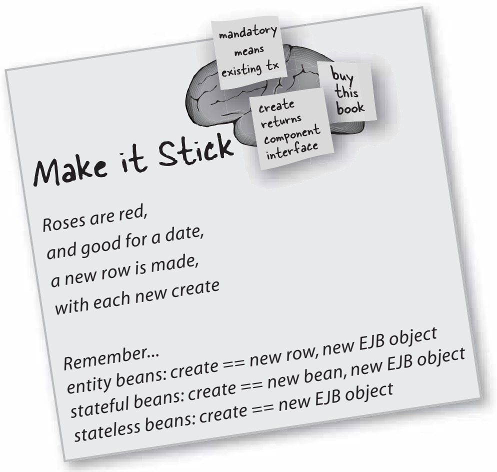
Session bean create() vs. entity bean create()

1 State*ful* session bean create()

- Client calls it to get an EJB object reference to a new just-for-me stateful session bean.
- It can (and frequently does) have arguments, that the bean uses to do client-specific initialization (before running any business methods).
- The Container makes a new session bean when the client calls create()

2 State*less* session bean create()

- Client calls it to get an EJB object reference to a bean
- It has no arguments, and the bean does not do any client-specific initialization (since at the time the bean's ejbCreate() is called, the bean has no association with a client!)
- The Container does *not* make a new session bean when the client calls create(), and does not pull one out of the pool until the client invokes a business method.


mandatory means existing tx
buy this book
create returns component interface

Make it Stick

Roses are red,
and good for a date,
a new row is made,
with each new create

Remember...
entity beans: create == new row, new EJB object
stateful beans: create == new bean, new EJB object
stateless beans: create == new EJB object

3 Entity bean create()

- Client calls it to insert a new row in the database!* Although the end result for the client is still an EJB object reference (in this case, to the newly-created entity).
- It will virtually *always* have arguments (although they aren't mandatory, but it's kinda hard to imagine a create() without them... like, "Hey database, create a new customer... no, I don't have any name or ID or anything... just make some stuff up").
- The Container does *not* make a new entity bean, but it does pull one out of the pool to run the ejbCreate() method. Remember, the ejbCreate() method has to take the create() arguments and somehow create a new entity in the underlying persistent store (or at least support the Container in creating a new entity).

Session bean remove() vs. entity bean remove()

① State*ful* session bean remove()

- Client calls it to tell the Container that he's done with the bean
- Container calls the bean's ejbRemove() (unless the bean is already passivated) and kills the bean (think: food for the garbage collector)
- Client will get an exception if he tries to use the EJB object reference after removing the bean.

② State*less* session bean remove()

- Client calls it to tell the Container that he's done with the bean
- Container gets the call and says, "Like I care? Do you honestly think you're that important? This bean is already back in the pool baby." The Container does *not* call a bean's ejbRemove(). Think about it—*which* bean's ejbRemove() would it call?
- Client will get an exception if he tries to use the EJB object reference after removing the bean.

③ Entity bean remove()

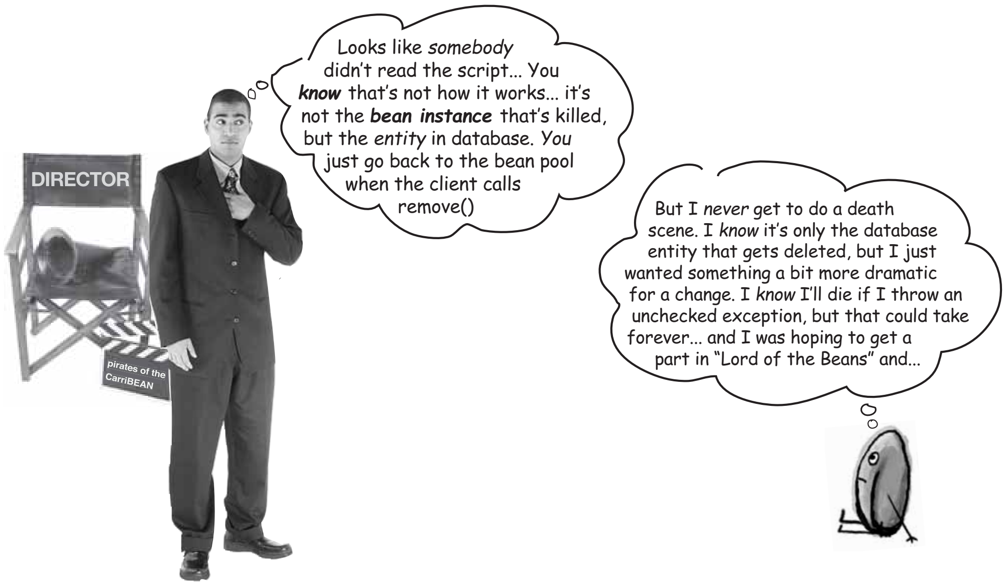
- Client calls it to tell the Container to delete the entity with this primary key.
- Container calls the bean's ejbRemove() method and—if the bean supports client-triggered removal—the entity is deleted from the underlying persistent store. In other words, the row in the database is history. Gone. Poof.
- Client will get an exception if he tries to use the EJB object reference after removing the bean.
- In fact, ***NO client will be able to use an EJB object reference to that entity!***

remove() on a session bean means the client is done with the bean.

remove() on an entity bean means EVERYONE is done with the bean!

I hate to do this, really, but I have no choice. #86 MUST be removed from the database. So if you've got any last words, you better do it in your ejbRemove()...
No, Please, No! I'll give you whatever you want, just don't call remove()!

Cut!
DIRECTOR
pirates of th
CarriBEAN


Looks like *somebody* didn't read the script... You ***know*** that's not how it works... it's not the ***bean instance*** that's killed, but the *entity* in database. *You* just go back to the bean pool when the client calls remove()
But I *never* get to do a death scene. I *know* it's only the database entity that gets deleted, but I just wanted something a bit more dramatic for a change. I *know* I'll die if I throw an unchecked exception, but that could take forever... and I was hoping to get a part in "Lord of the Beans" and...
DIRECTOR
pirates of the
CarriBEAN

Entity/bean/instance death

So we all know that an entity bean is a representation of some real entity in an underlying persistent store (usually as a row in a database, blah, blah, blah). But there's still some confusion about what distinguishes an *entity* from an entity *bean* from an entity bean *instance*.

Entity

The real thing in the underlying persistent store. The row in the database (although it can be more complex). An entity dies when its *row is deleted* from the underlying store, either through a direct database delete (like, someone using a database admin tool), or because someone calls remove() on the bean's home or component interface.

Entity bean

The component that represents the underlying real entity. But this one's tricky... is it the class? Is it the *interface*? Is it the *instance* of the bean *class*? During development and deployment, the entity *bean* is the whole component (the two interfaces, DD, and bean class). But at *runtime*, it can get a little fuzzy. S ometimes we use "entity bean" to describe the *possibility* of representing a particular entity as a bean. In other words, if there's an entity for Bo Rodgers in the database, then we can say that there is a Bo Rodgers entity *bean*, even if there's no bean instance currently representing that entity! If an *entity* exists for a particular bean type (like Customer Fred Foo), the entity *bean* for that entity is said to exist. An entity bean is said to die when its underlying entity is deleted, as in, "There's no Fred Foo entity bean." ***But***... that doesn't mean the *instance on the heap* dies.

Entity bean instance

The instance of the bean class on the heap. *Bean* death is intimately tied to the database, but bean *instance* death (as in, "you're headin' for garbage collection, pal") is tied to the whims of the Container, or a server crash.

Yes, it really is that confusing. You have to know the context to know how the word 'bean' is being used. If it means the *bean-representing-the-entity*, then that *bean* will die when the entity dies, and the EJB object for that entity goes away. But—and here's where it gets weird—the entity bean *instance* doesn't die; it just goes back to the pool. Think of the phrase "entity bean" as more conceptual than physical. In most cases, we won't have to distinguish between the bean and its instance, or the distinction will be so obvious that its not an issue.

A client can kill an entity, by calling remove() on a bean, or deleting the data from the database directly.

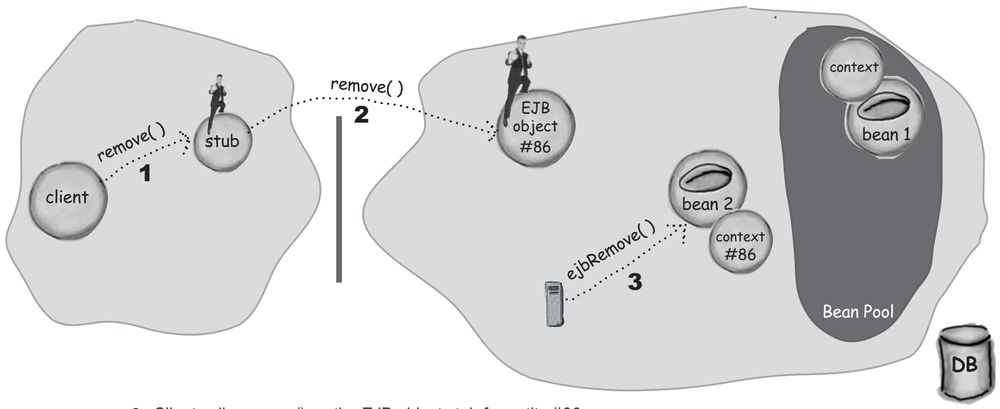
But only the Container or a server crash can kill a bean instance.

If we say "entity bean #42 was killed", the underlying entity is gone, and the EJB object for #42 is gone, but the bean instance that had been playing #42 survives.

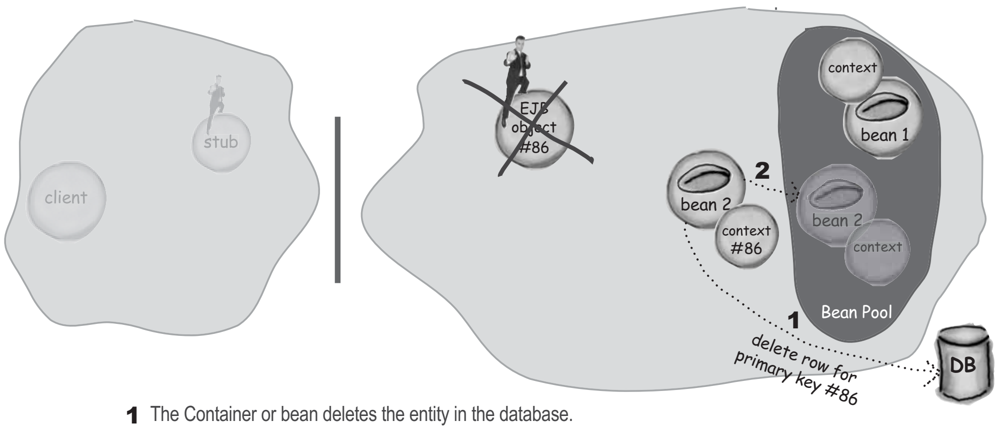
Well, it's been great playing you, really, but now it's time for you to go to a better place. I'll think about you while I'm floating on one of those little inflatable mattress things.
Darko | Donny | 42
entity DIES when a client calls remove()
bean instance LIVES even if the entity it plays is removed.

Head First EJB™ By Bert Bates, Kathy Sierra ISBN: 0596005717 Publisher: O'Reilly
Print Publication Date: 2003/10/01

Client calls remove()



1. Client calls remove() on the EJB object stub for entity #86
2. The remove() method invocation is passed to the EJB object.
3. The Container calls ejbRemove() on the bean.



1. The Container or bean deletes the entity in the database.
2. The bean loses its identity (in other words, it is no longer representing entity #86) and moves back to the pool. Meanwhile, the EJB object for #86 is deleted, so the client's stub will throw an exception if the client uses it to invoke a method.

Entity bean client view

BULLET POINTS

- An entity is a real thing that exists outside of EJB, in a persistent store, and an entity *bean* is an OO representation or realization of an entity.
- Clients use entity beans to do database operations, in an OO way. Operations include creating new entities (database inserts), deleting entities (database deletes), updating entity state (database updates), and searching for/on entities (database selects).
- An entity bean Remote component interface extends EJBObject. There's not a separate interface for session beans and entity beans. That means that the client will see all of the methods in your component interface, plus the five additional methods from EJBObject.
- Entity bean component interfaces usually contain getters and setters for field values that correspond to columns in a database table, such as getLastName(), getHomePhone(), setFirstName(), etc.)
- Entity bean component interface methods are usually meant to be run by a specific, uniquely-identifiable entity. For example, calling getLastName() on the entity with primary key #420, returns the last name of the entity in the database with the primary key #420, Dan Doof.
- The rules for how you write an entity bean Remote component interface are the same as the rules for session beans, including: extend EJBObject, declare RemoteExceptions on all methods, use only RMI-IIOP types for arguments and return values, don't begin method names with the prefix "ejb", etc.
- An entity bean home interface is substantially different from that of a session bean, because entity beans are typically found rather than created. In other words, the client is more likely to try to access an existing entity as opposed to making a new entity (which means a new row in the database.)
- In an entity home, a create() method is not required (since create() method in entity beans are for inserting new entities into the database, and you're not required to allow your clients to do that.)
- Entity bean home interfaces can have single-row or multi-row finder methods. Both create and finder methods return the component interface of a bean, although multi-entity finders return a Collection of component interface references.
- Every entity bean home is required to have at least one method—the findByPrimaryKey() method that searches for a particular entity and returns its component interface (i.e. a reference to that entity's EJB object), or throws an exception.
- Multiple-entity finders do not throw an exception if no matching entities are found. They simply return an empty collection.
- Entity home interface can also have home business methods, for operations that apply to more than one entity, as opposed to one specific entity. Batch updates would be a good use for a home business method.
- The real benefit of home business methods is that—unlike create and finder methods—home business methods can return something other than an EJB object reference. If the client wants only data, say, a Collection of Strings representing the name and phone number of each customer, a home business method can do that while a finder can not.
- An entity bean create() is very different from a session bean create(), because an entity bean create() inserts a new entity into the underlying persistent store (i.e. new row in the database).
- An entity bean remove() is dramatically different from a session bean remove()! When a client calls remove() on an entity bean, it's to delete the entity from the database! That means *everybody* is done with the bean.

COFFEE CRAM

Mock Exam

---

1 What is true concerning locating an entity bean's home interface?

❏ A. The **`narrow()`** method should be used for a local home interface.

❏ B. The **`narrow()`** method should be used for a remote home interface.

❏ C. The **`narrow()`** method should be used for both local and remote home interfaces.

❏ D. The **`narrow()`** method should be used for neither local nor remote home interfaces.

---

2 Which capabilities are found in an entity bean's remote component interface? (Choose all that apply.)

❏ A. creating new entity objects

❏ B. finding existing entity objects

❏ C. removing existing entity objects

❏ D. executing a home business method

❏ E. retrieving the EJBMetaData interface

---

3 Which are ways in which a client can get a reference to an existing entity object's local component interface? (Choose all that apply.)

❏ A. Call **`ejbCreate()`**

❏ B. Call **`getSessionContext()`**

❏ C. Obtain the reference from the handle.

❏ D. Receive the reference as a parameter in a method call.

❏ E. Use a finder method defined in the local home interface.

Head First EJB™ By Bert Bates, Kathy Sierra ISBN: 0596005717 Publisher: O'Reilly
Print Publication Date: 2003/10/01

4 How many create methods can be defined in an entity bean's local home interface?

❏ A. 0

❏ B. Only 1

❏ C. 0 to 1

❏ D. 0 to many

❏ E. 1 to many

5 Which are ways in which a client can get a reference to an existing entity object's remote component interface? (Choose all that apply.)

❏ A. Call **`ejbCreate()`**.

❏ B. Obtain the reference from the handle.

❏ C. Call a method on the entity object's primary key.

❏ D. Receive the reference as a parameter in a method call.

❏ E. Use a finder method defined in the remote home interface.

6 Which approach(es) can be used on a primary key class to determine if two keys refer to the same entity? (Choose all that apply.)

❏ A. Using the == operator.

❏ B. Using the **`equals()`** method.

❏ C. Using the **`isIdentical()`** method.

❏ D. none of the above.

7 Which approach(es) can determine whether two entity EJB object references refer to the same entity object? (Choose all that apply.)

❏ A. Using the == operator.

❏ B. Using the **`equals()`** method.

❏ C. Using the **`isIdentical()`** method.

❏ D. none of the above.

8 What's true about the client's view of an entity bean's remote component interface? (Choose all that apply.)

- ❑ A. Multiple clients can access the same entity object concurrently.
- ❑ B. New entity beans can be created using a method in this interface.
- ❑ C. Entity beans may not survive a crash of the container.
- ❑ D. Business methods cannot return a reference to the entity object.

9 How many finder methods can be declared within an entity bean's local home interface?

- ❑ A. 0
- ❑ B. Only 1
- ❑ C. 0 to 1
- ❑ D. 0 to many
- ❑ E. 1 to many

10 Which is a legal declaration for a local home interface's create() method? (Choose all that apply.)

- ❑ A. `public Cust create(int x);`
- ❑ B. `public void create(int x) throws CreateException;`
- ❑ C. `public Cust create(int x) throws CreateException;`
- ❑ D. `public Cust create(int x) throws CreateException, RemoteException;`

11` Which is a legal name for an entity bean home business method? (Choose all that apply.)

- ❑ A. create
- ❑ B. createCust
- ❑ C. removeAll
- ❑ D. findCust
- ❑ E. selectCust

COFFEE CRAM

Mock Exam Answers

1 What is true concerning locating an entity bean's home interface? (spec: 110)

- ❑ A. The **`narrow()`** method should be used for a local home interface.
- ✓ B. The **`narrow()`** method should be used for a remote home interface.
- ❑ C. The **`narrow()`** method should be used for both local and remote home interfaces.
- ❑ D. The **`narrow()`** method should be used for neither local nor remote home interfaces.

2 Which capabilities are found in an entity bean's remote component interface? (Choose all that apply.)

- ❑ A. creating new entity objects – home methods
- ❑ B. finding existing entity objects
- ✓ C. removing existing entity objects
- ❑ D. executing a home business method – home methods
- ❑ E. retrieving the EJBMetaData interface

3 Which are ways in which a client can get a reference to an existing entity object's local component interface? (Choose all that apply.) (spec: 119)

- ❑ A. Call **`ejbCreate()`** – we said existing :)
- ❑ B. Call **`getSessionContext()`**
- ❑ C. Obtain the reference from the handle. – handles are for Remote interfaces
- ✓ D. Receive the reference as a parameter in a method call.
- ✓ E. Use a finder method defined in the local home interface.

4 How many create methods can be defined in an entity bean's local home interface? (spec: 115)

- ❑ A. 0
- ❑ B. Only 1 – You don't have to allow clients to create new entities
- ❑ C. 0 to 1
- ✓ D. 0 to many
- ❑ E. 1 to many

5 Which are ways in which a client can get a reference to an existing entity object's remote component interface? (Choose all that apply.) (spec: 119)

- ❑ A. Call `ejbCreate()`. – we said EXISTING
- ✓ B. Obtain the reference from the handle.
- ❑ C. Call a method on the entity object's primary key.
- ✓ D. Receive the reference as a parameter in a method call.
- ✓ E. Use a finder method defined in the remote home interface.

6 Which approach(es) can be used on a primary key class to determine if two keys refer to the same entity? (Choose all that apply.) (spec: 120-121)

- ❑ A. Using the == operator.
- ✓ B. Using the `equals()` method. – if two keys pass the equals() test, they're the same key! Which means the same entity.
- ❑ C. Using the `isIdentical()` method. – isIdentical() is for comparing component interface references
- ❑ D. none of the above.

7 Which approach(es) can determine whether two entity EJB object references refer to the same entity object? (Choose all that apply.) (spec: 120-121)

- ❑ A. Using the == operator.
- ❑ B. Using the `equals()` method. – equals() is for comparing entity's primary keys
- ✓ C. Using the `isIdentical()` method.
- ❑ D. none of the above.

8 What's true about the client's view of an entity bean's remote component interface? (Choose all that apply.) (spec: 108)

- [✓] A. Multiple clients can access the same entity object concurrently.
- ❑ B. New entity beans can be created using a method in this interface.
- ❑ C. Entity beans may not survive a crash of the container.
- ❑ D. Business methods cannot return a reference to the entity object.

9 How many finder methods can be declared within an entity bean's local home interface? (spec: 116)

- ❑ A. 0
- ❑ B. Only 1
- ❑ C. 0 to 1
- ❑ D. 0 to many
- [✓] E. 1 to many – findByPrimaryKey() is required

10 Which is a legal declaration for a local home interface's create() method? (Choose all that apply.) (spec: 115)

- ❑ A. `public Cust create(int x);` – needs CreateException
- ❑ B. `public void create(int x) throws CreateException;` – can't return void, must return component interface
- [✓] C. `public Cust create(int x) throws CreateException;`
- ❑ D. `public Cust create(int x) throws CreateException, RemoteException;` – local interface can't throw RemoteException

11 Which is a legal name for an entity bean home business method? (Choose all that apply.) (spec: 114)

- ❑ A. create
- ❑ B. createCust
- ❑ C. removeAll
- ❑ D. findCust
- [✓] E. selectCust

"find" "create" & "remove" are reserved prefixes