

there are no
Dumb Questions

Q: If it's so common to leave the methods empty, why don't they have adapter classes like they have for event handlers—that implement all the methods from the interface? Is there any reason why your bean class can't extend a class that implements the `SessionBean` interface?

A: The API doesn't have adapter classes for `SessionBean` implementations (i.e. a class that implements all of the methods). But there's no reason you can't make one yourself. Keep in mind, though, that with real-world beans you probably *will* have code in one or more of the methods. And you might even be working with a bean-aware IDE that puts the methods in for you anyway.

Still, it might be handy to make yourself a generic bean that you typically extend from, that has all of the methods from `SessionBean`. With stateless beans, especially, you have to implement `ejbActivate()` and `ejbPassivate()`, even though they'll never be called! (Stateless beans are never passivated; you'll see more on that later in the chapter.)

Q: I just remembered that I read somewhere that enterprise beans don't support inheritance! What's *that* about?

A: Ah... a common misconception. Well, sort of. EJB supports regular Java *class* inheritance, but has no concept of *bean* inheritance. And now you're asking, "What the heck is the difference?" You already know what *class* inheritance is, it's the thing you do in Java when one *class* extends another. And you can do that with a bean, just like any other class.

But *bean* inheritance (if it were supported) would mean that a *bean* class could extend another *bean* class and inherit not just the class' inheritable members, but also its *beanness*. What kind of beanness might be inheritable? (Just in case they do decide to support this in the future, which is a possibility. Regular old non-enterprise beans *do* support bean inheritance.)

One idea might be to have your bean subclass inherit some of the deployment descriptor settings of its superclass, and then override the ones it wants to change with a much smaller, incomplete deployment descriptor. That might be cool; we're not sure. But right now, it's just our little fantasy.

In the meantime, go ahead and let your bean extend another class, if it makes sense for your OO design.

 Sharpen your pencil

For the exam, you have to know exactly which methods are in the `SessionBean` interface, so now is a good time to start memorizing them. See if you can remember the name of the method that matches the behavior described. We've included some pretty big hints here because it's your first time, but the mock exam questions will be much less obvious...

1. This method is called when the client tells the Container that he's done using a stateful session bean. The bean is NOT happy:

`ejbRemove()`

2. This method is called when the bean is put to sleep to temporarily conserve resources:

`ejbPassivate()`

3. This method is called when the previously-sleeping bean is called back to active duty to service a business method:

`ejbActivate()`

4. This method is called near the beginning of the bean's life, when the Container hands the bean a reference to the bean's special link to the Container:

`setSessionContext(SessionContext sc)`

Sharpen your pencil



You're responsible for making sure that when `ejbPassivate()` completes, your instance variables are in one of the states we listed a couple of pages ago. Don't look now! See if you can work out which of these will be safely passivated...

You can't passivate a non-Serializable, non-null value... so that means no Socket, and no Connection.

A transient non-null value is fine for passivation, but there is no guarantee that when it comes BACK from activation it will have default values. In other words, JUST using the 'transient' modifier without also setting the value to null might passivate OK, and activate OK, but the value after activation might be weird! (Moral: go ahead and use transient, but also set the values to null in `ejbPassivate()`)

Big explosion here (well, technically, one of the things the Container IS required to passivate might not actually be Serializable, but that's not your concern. The Container is still required to save those special things (like bean references, JNDI context, SessionContext, etc.) as if they WERE Serializable.

- reference to a `java.net.Socket` object
- reference to a `javax.sql.DataSource`
- reference to a bean's Remote component interface
- reference to a bean's JNDI context
- reference to a `java.sql.Connection`
- reference to a `javax.ejb.SessionContext` object
- a transient variable with a null value
- a non-transient, Serializable variable with a null value
- a transient variable with a non-null value
- a non-transient, non-Serializable variable with a null value
- a non-transient, non-Serializable variable with a non-null value*



Do the interfaces have to change when it goes from stateless to stateful?

Look at the two interfaces below, for the stateless version of the Advice bean. If needed, make any adjustments to the code in either or both of the interfaces, for what (if anything) needs to change to make this work with the revised stateful version of the bean.

```
package headfirst;

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

public interface AdviceHome extends EJBHome {
    public Advice create() throws CreateException, RemoteException;
}

public Advice create(String name) throws CreateException, RemoteException;
```

```
package headfirst;

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

public interface Advice extends EJBObject {
    public String getAdvice() throws RemoteException;
}
```

Nothing changes with the business method interface (even though the implementation will change)

Writing a Session Bean: your job as Bean Provider

I have a system when I sit down to write a Session bean... I always have to put in **three** kinds of things: home stuff, business methods, and the SessionBean methods.



```

<<interface>>
AdviceHome
create()
    
```

① HOME things: ejbCreate() methods

Write an ejbCreate() method in the bean to match each create() method in the home interface.

```

<<interface>>
Advice
getAdvice()
    
```

② COMPONENT things: business methods

Write a business method in the bean to match each method in your bean's component interface.

```

<<interface>>
SessionBean
setSessionContext()
ejbPassivate()
ejbActivate()
ejbRemove()
    
```

③ SESSION BEAN things: container callbacks from the SessionBean interface

Implement each of the four methods from the SessionBean interface, which your bean *must* implement in the official Java way (i.e. using the *implements SessionBean* declaration either in your bean class or one of its superclasses)



Of the three types of methods you put in your bean, check off the ones the compiler cares about.

Compiler-checked?

- Methods to match the Home interface
- Methods to match the Component interface
- Methods from the SessionBean interface



Given the following interfaces, write the bean class code (you can leave the method empty) at the bottom of the page. Pay special attention to the Home create method... what does it take to 'match' this in the bean? Will it have the same return type? Hints are at the bottom of the page.

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

public interface KennelHome extends EJBHome {
    public Kennel create(String custID) throws CreateException, RemoteException;
}

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

public interface Kennel extends EJBObject {
    public KennelLease placePet(Pet p) throws RemoteException;
    public void renewLease(KLease lease) throws RemoteException, ExpiredException;
    public Pet getPet(KLease lease) throws RemoteException, DeadPetException;
}
```

Write the bean class here:

```
public class KennelBean extends javax.ejb.EJBObject {

    public void ejbCreate(String custID) throws CreateException { }
    public void ejbActivate() { }
    public void ejbPassivate() { }
    public void setSessionContext(SessionContext ctx) { }

    public KennelLease placePet(Pet p) {
        return new KennelLease();
    }

    public void renewLease(KLease lease) throws ExpiredException { }
    public Pet getPet(KLease lease) throws DeadPetException { }
}
```

We don't throw `RemoteException`, but it's a good idea to declare `CreateException` on your create...

We don't throw `RemoteException`, but we're assuming that if you really implemented this class, you'd declare and use these other application-specific exceptions. But according to plain old Java rules, you don't have to declare the exceptions declared in the interface, unless your method really **DOES** potentially throw that exception...



Who does What?

From the list of words below, arrange them in the appropriate lists according to whether it's a responsibility of the Bean Provider, the Container, or the Client.



Bean Provider



Client



Container

implementing the `ejbActivate()` method
 implementing the `ejbCreate()` method
 implementing `SessionBean`
 creating the home interface

invoking a business method
 on the component interface
 invoking `create()`

invoking `ejbPassivate()`
 implementing the `create()` method
 implementing the `EJBObject` class
 invoking `setSessionContext()`
 implementing the `Handle` class
 invoking `ejbCreate()`
 creating the `Home` object class
 implementing the `SessionContext` class
 invoking `ejbRemove()`

