Table of Contents

3 the client view

Exposing Yourself

You can't keep your bean private. Clients need to see what you've got. (Except for message-driven beans, which don't *have* a client view). The Advice Bean exposed the getAdvice() method in its Component interface—the place where you declare business methods. But that's not *all* the client sees. Remember, the Advice interface extended EJBObject, an interface with methods of its *own*. Methods the client can see. Methods the client can *call*. And it works the same way with the Home interface. In this chapter, we'll learn what you *really* expose to the client, and how the client works, including both Remote and *local* interfaces.

The Client View

Official: | **What it really means:**

2.1 Identify correct and incorrect statements or examples about the client view of a session bean's local and remote home interfaces, including the code used by the client to locate a session bean's home interface.

You have to know everything about the *home* interface. This particular objective doesn't include the special characteristics of an entity bean home, but *most* of the details about the client's view of a bean's home are still covered in this objective (and this chapter).

For example, you have to know exactly which methods are in javax.ejb.EJBHome (the Remote home interface), and which methods are in javax.ejb.EJBLocalHome (the local home interface). And it's not enough to know what the methods *are*—you also have to know the circumstances under which they can be *called*. You have to know, for instance, that a *Remote* session bean client can *remove* a bean using the bean's home, but a *local* client *cannot*. And you have to know that a local home has fewer methods than a Remote home, and what that means for the client.

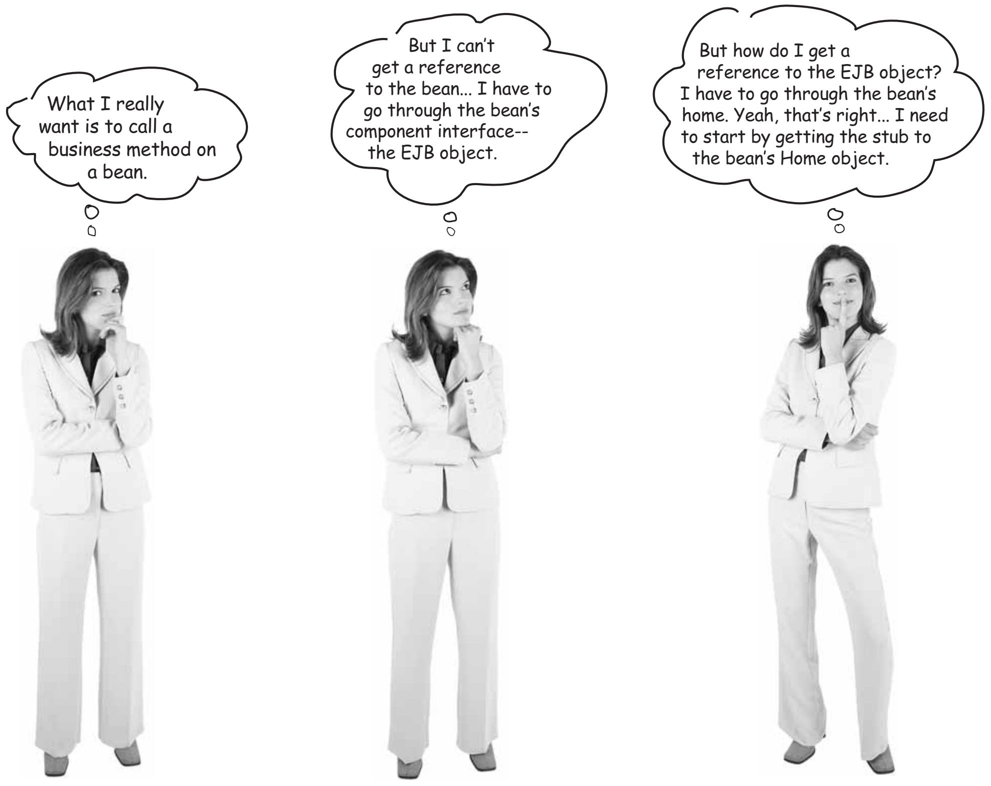
Finally, for Objective 2.1, you have to know the ins and outs of how a client does a JNDI lookup on a bean's home interface. That includes the syntax of the client's lookup code, the rules for performing the lookup, and how to use the home interface to get a reference to a bean's component interface. You have to know, for example, the rules for narrowing a home stub, and, given a code snippet, you must be able to recognize whether the client is local or Remote.

2.2 Identify correct and incorrect statements or examples about the client view of a session bean's local and remote component interfaces.

This objective is just like 2.1, except it's about the *component* interface. But again, you must know all the methods of both javax.ejb.EJBObject and javax.ejb.EJBLocalObject, and how they're used by the client, and you must be able to recognize the difference between a Remote and local client, just by looking at code.

What the client really wants

The client has a goal. A vision. A quest. She wants to call a business method on the bean! Something exposed in the component interface. Never forget that ultimate goal; it *is* easy to get bogged down in all the details. But if you keep focused on the client's driving need, you'll have a much easier time remembering things like, say, the return type of a session bean's home create() method.


What I really want is to call a business method on a bean.
But I can't get a reference to the bean... I have to go through the bean's component interface-- the EJB object.
But how do I get a reference to the EJB object? I have to go through the bean's home. Yeah, that's right... I need to start by getting the stub to the bean's Home object.

It all starts with the home interface

The client wants the bean. Well, too bad. ***The client will never get the bean***, because *nobody* talks to the bean (except the container). The best the client can hope for is a reference to the bean's *bodyguard*—the component interface. And the client gets a reference to the bean's *component* interface by calling a method on the bean's *home* interface.

What the client REALLY wants is a reference to the bean. But the best the client can do is get a reference to the bean's component interface--the EJB object*.

But if she wants an EJB object reference, the client has to get a reference to the bean's home interface.

So that's where it begins... the client does a lookup on the bean's home.

*We use the word "EJB object" to mean the bean's component interface (the bodyguard), i.e. the thing receiving method calls meant for the bean, regardless of whether the client is local or remote.

there are no Dumb Questions

Q: How come you said, "And the client gets a reference to the bean's component interface..." You can't *have* a reference to an interface in Java—you can reference an *object*, but you can't reference an *interface*. The reference variable can be declared as an interface type, but that's not the same thing.

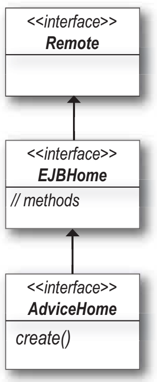
A: Well, actually it *is* the same thing. In this book, and in the spec, and in the exam, everywhere you see the phrase, "reference to an interface", do a mental search and replace to make it, "reference to an *object that implements* the interface."

It can feel a little strange, if you haven't read documents that use this convention, but you better get used to it. But don't worry, by Chapter 4, you'll wonder how anyone ever said it differently...

Q: You said the client's ultimate goal is to call methods on the bean. (OK, the component interface, but you know what I mean.) But with entity beans, you can have business methods in the *home*, right? So with entity beans, isn't it true that sometimes the client's goal is JUST to use the home?

A: Yes, you're right. They're called "*home* business methods" (as opposed to plain old "home methods" or plain old "business methods"). But they're a *special case* we'll look at later in the book. There are other reasons, too, for why you might need only the home of an entity bean. For example, if you want to create a bunch of new customers in a database, but you don't want to do any other operations on references to those entity customers.

How a client uses a session bean: create, use, and remove



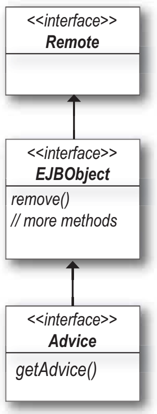
Note: over the next several pages, we're looking at only the REMOTE interfaces for a bean. Later we'll see the special-case "local" versions).

① Create

Client asks the *home* interface for a reference to the bean's *component* interface.

(Which means the client calls a create() method on the home stub.)

The bean developer must put a create() method in the bean's home



② Use

Client calls business methods declared in the component interface.

(Which means the client calls methods on the EJB object stub.)

The javax.ejb.EJBObject interface has a remove() method the client can call.

③ Remove

Client tells the bean that he's done using it.

(Which means the client calls remove() on the bean's EJB object stub.)

In the component interface, the developer puts in the business methods, but since the component interface EXTENDS the EJBObject interface, the client sees the methods from BOTH interfaces.

But first, we have to get a home interface reference

In other words, we have to get the stub to the home object... the thing we use to call create(), so that we can get what we really want— the EJB object stub!

When you (the client) want a reference to a home interface, you go through JNDI. The process is pretty straightforward: you give JNDI a logical name (the name the deployer told the server to use), and you get back something that implements the home interface.

```
Context ic = new InitialContext();

Object o = ic.lookup("Advisor");

// a few more steps...
```

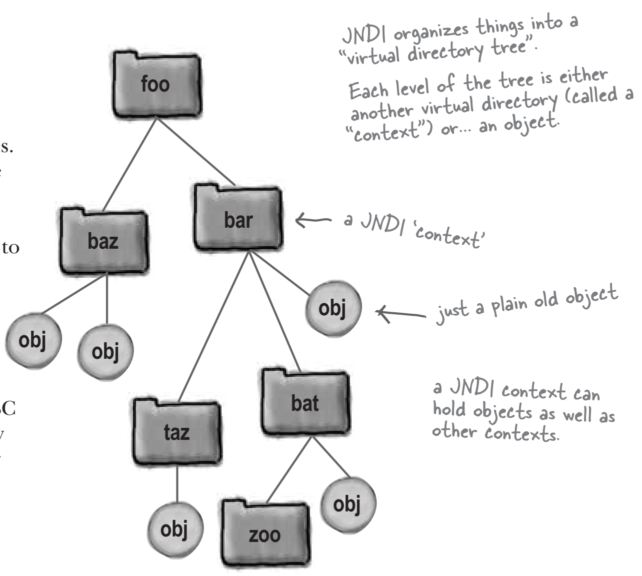
The InitialContext object assigned to "ic" is a reference to the JNDI lookup service.

Give it a name (whatever the bean deployer used to register that bean with the server) and get back an object.

What's JNDI?

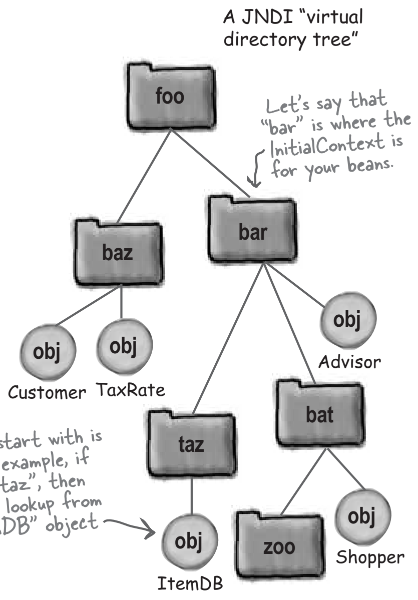
JNDI stands for Java Naming and Directory Interface, and it's an API for accessing naming and directory services. Although JNDI is quite powerful, there are only a few pieces of it you need to know for EJB—how clients *find* it, how clients *use* it, how *beans* use it, and how to put things *into* it.

The JNDI API can work with many different services, as long as that service has a JNDI driver (called a Service Provider). It's a lot like JDBC, where you (the developer) use the JDBC API to send SQL statements to a variety of different databases. The JNDI driver translates the method calls you make on the JNDI API into something the underlying naming/directory service understands.


JNDI organizes things into a "virtual directory tree".
Each level of the tree is either another virtual directory (called a "context") or... an object.
foo
baz
bar
a JNDI 'context'
obj
obj
obj
just a plain old object
taz
bat
a JNDI context can hold objects as well as other contexts.
obj
zoo
obj

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

Getting the home interface stub


A JNDI "virtual directory tree"
foo
Let's say that "bar" is where the InitialContext is for your beans.
baz
bar
obj
Customer
obj
TaxRate
obj
Advisor
taz
bat
Whichever context you start with is your InitialContext. For example, if the InitialContext was "taz", then the only thing you could lookup from here would be the "ItemDB" object
obj
ItemDB
zoo
obj
Shopper

1 Get an InitialContext

```
Context ic = new InitialContext();
```

The InitialContext (a subtype of Context) is your entry point into the JNDI tree. It's simply the first context from which you start navigating. Kind of like your current working directory (the one you cd to).

The InitialContext constructor figures out where you should start. (We'll talk about that in a minute.) Both Context and InitialContext are part of the javax.naming package, part of J2EE, but also added to J2SE with version 1.4.

2 Lookup the bean's home using the InitialContext

```
Object o = ic.lookup("Advisor");
```

The lookup method takes a String that must match the name assigned to this bean's JNDI deployment. If the deployer assigned an additional context to the bean, by naming it (at deploy-time) "bat/Advisor", then the lookup code would change to:

```
ic.lookup("bat/Advisor");
```

3 Assign the result of the lookup to the Home interface reference

```
AdviceHome home = (AdviceHome) o;
```

Warning: This code isn't quite right... although it LOOKS like it should be. We'll find out what's wrong in just a few pages.

The return type of the context lookup method is Object, so you have to cast it back to the bean's home interface type, before you can call AdviceHome methods.

there are no Dumb Questions

Q: How do I know what the developer named the bean?

A: Actually, it's not up to the bean *developer* (the EJB role known as Bean Provider—the one who actually wrote the bean code), to give the bean its JNDI name. Remember, the Bean Provider might have written that bean as a reusable component for Beans 'R' Us and thus might have no idea where and how the bean will be used.

It's the *deployer*—the person who actually gets the bean running in the server as part of some application—who registers the bean under a logical name. But the bottom line is that there's no standard or automatic mechanism for learning the names of registered beans. As a client, somebody, somehow, has to tell you that, say, the bean was registered as "Advisor".

Notice that "Advisor", while describing the service, is not a String that corresponds directly to the names of any of the other pieces of the bean. Remember, the component interface was Advice, the home interface was AdviceHome, and the bean itself is AdviceBean. The name "Advisor" was just something the deployer thought had a nice ring to it.

Of course, in your company, you might (and probably will) have strict naming guidelines to follow for how beans are registered with JNDI at deployment time.

(Unless it's your own company, in which case you can do whatever you darn well please, including naming each bean after your favorite rock star or Matrix character.)

Q: I just thought of an even BIGGER problem... how the heck do I know where to find the server? And how do I specify it? I didn't see any code for an IP address or TCP port number.

A: Good catch. Yeah, that's all a bit of a mystery, isn't it? We have three answers for now:

1) We're cheating a little, because the code we're using works only because we're using the Reference Implementation, and even then... only because we're running the server on the same physical machine as the client. So, we're taking advantage of default settings that are in place, automatically, because we're running the Reference Implementation.

2). We lied a little in point number 1, above, because this code *could* be correct if it were inside a bean. (We'll get to that in the Bean Environment chapter.)

3) In reality, a client *does* need to know how to find the JNDI service where a bean's home is registered. There are several ways you can do this—you could pass information to the InitialContext constructor (a Properties object that contains everything the InitialContext needs to find the server and the starting context). Or, there are several places where JNDI properties can be placed on the client's machine. In either case, the client MUST be given something—either info for the InitialContext constructor, or a properties file. It's different for each vendor's server, too, so you have to check your documentation in order to know what the client needs.

For the exam, you don't need to know much about JNDI!

For the client-related objectives (the ones from this chapter), all you need to know is the fundamental process for doing a JNDI lookup... that you need to start with an InitialContext and then call lookup(), which returns something of type Object.

You do NOT need to know how the client or server finds and gets a reference to the correct InitialContext, only that an InitialContext is needed.

In the Bean Environment chapter, we'll add a tiny bit more JNDI info, for how the bean itself uses JNDI to look up things that have been specifically placed there for the bean.

But that's about it for your JNDI knowledge. You don't have to know any details about the rest of the JNDI API other than the Context.lookup() method.

Let's take another look at the complete client code

```
import javax.naming.*;
import java.rmi.*;
import javax.rmi.*;
import headfirst.*;
import javax.ejb.*;

public class AdviceClient {

   public static void main(String[] args) {
      new AdviceClient().go();
   }

   public void go() {
      try {
         Context ic = new InitialContext();
         Object o = ic.lookup("Advisor");

         AdviceHome home = (AdviceHome) PortableRemoteObject.narrow(o, AdviceHome.class);

         Advice advisor = home.create();

         System.out.println(advisor.getAdvice());

      } catch (RemoteException rex) {
           rex.printStackTrace();
      } catch (CreateException cex) {
           cex.printStackTrace();
      } catch (Exception ex) {
           ex.printStackTrace();
      }
   }
}
```

A bunch of imports, we'll look at each one individually at the bottom of the page

InitialContext is our entry point into the JNDI naming service, where we do the lookup on the name "Advisor"

What is THIS??? Why not just a plain old cast?

Call create on the home to get us what we REALLY want — the component interface.

The point of everything! To call a business method on the bean (via the EJBObject stub)

Not a good way to handle (or rather, NOT handle) exceptions here... but we want to show some of the checked exceptions...

Sharpen your pencil

Match the class name with the package it's from. You can use the same package name more than once.

If you're not sure, make your best guess.

Package Name	Class Name
`javax.naming`	`InitialContext`
`java.rmi`	`AdviceHome`
`javax.rmi`	`PortableRemoteObject`
`headfirst`	`RemoteException`
`javax.ejb`	`Advice`
	`CreateException`

Just when you thought a simple cast would be enough...

The return value of the Context.lookup() method is type Object. So we're thinking a simple cast should be enough to force the object referenced by ***o*** back to the AdviceHome implementation that we know it really is:

```
Context ic = new InitialContext();
Object o = ic.lookup("Advisor");
AdviceHome home = (AdviceHome) o;
```

This LOOKS right, but isn't. With a Remote home interface, casting is not enough.

But NO. You have to narrow the object as well!

Narrowing forces the object returned from the JNDI lookup to be absolutely, positively, something that implements the home interface. In other words, something you can cast to AdviceHome.

```
Context ic = new InitialContext();
Object o = ic.lookup("Advisor");
AdviceHome home = (AdviceHome) PortableRemoteObject.narrow(o, AdviceHome.class);
```

OK, I'll bite. Why can't you just do a plain old cast?

According to the spec, you—the *client*—must assume that the server is using RMI-IIOP rather than regular old RMI. Normal RMI uses JRMP as the wire protocol, which assumes that we're always talking Java all the way down. If this were plain RMI, you'd always know that what you get out of the lookup is (polymorphically) something that IS-A home interface. In other words, *an object whose class type implements the home interface for that bean.* And for that scenario, a normal Java language cast would let you assign the object back to the home interface type, so that you can call the home methods! Otherwise, remember, you'd be stuck calling only methods of type Object (equals(), hashCode(), toString(), etc.) when what you *really* want to call is *create()*.

But when the wire protocol is IIOP, the rules change a little. The narrow() operation gives you something that *is* castable.

PortableRemoteObject.narrow()

The javax.rmi.PortableRemoteObject's narrow() method runs code written by the server vendor. But all we care about is that it takes the object we got from JNDI and gives us back something that really *does* implement the home interface.

In other words, it gives us back something we can then cast to the home interface type, and call create().

```
PortableRemoteObject.narrow(o, AdviceHome.class);
```

the object you got from JNDI

the interface type you want it to be. It must be a Remote interface!

Relax

You don't need to know the details of IIOP

For EJB, you don't need to know how IIOP works, unless you intend to develop non-Java CORBA clients. But we won't go there. Not here, in the book, and not in the exam, either. The details are way out of scope.

But... you DO need to know that the spec requires Bean Providers to assume that the container is using IIOP, and that your bean must be IIOP-compliant. What does it take to be IIOP-compliant? Narrowing the Remote stubs, when needed. And there are a couple of other tiny IIOP issues we'll look at in this chapter. The bottom line is that you DO have to be aware that IIOP is being used, but you don't need to know much more than that. We'll cover everything you need to know about IIOP before the chapter's done.

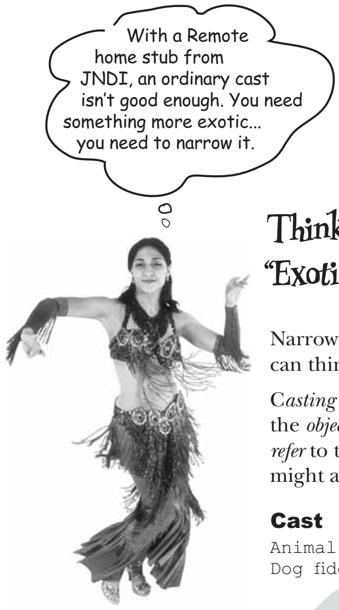
The home stub returned from a JNDI lookup might not implement the home interface!

You might get back an IIOP stub that isn't castable to the home interface of your bean. And that means you couldn't call create().

To get a stub that's castable to the home interface, you have to first narrow() the object you get from the JNDI lookup on the bean home.

(But only when the home interface is Remote.)

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


With a Remote home stub from JNDI, an ordinary cast isn't good enough. You need something more exotic... you need to narrow it.

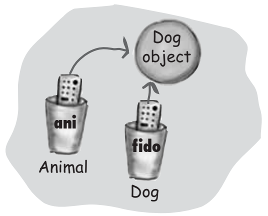
Think of narrowing as "Exotic Casting"

Narrowing is not the same as casting, but you can think of it as a form of "exotic casting".

C*asting* is about polymorphism. With a cast, the *object* doesn't change, but the way you *refer* to that object does. With *narrowing*, you might actually get a *different* object!

Cast

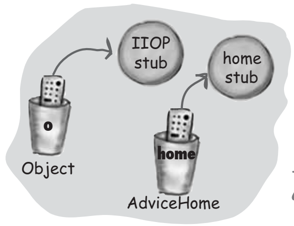
```
Animal ani = new Dog();
Dog fido = (Dog) ani;
```



Casting lets you see ONE object in multiple ways. The reference type determines what methods you can call, but the object itself always knows it's a Dog.

Narrow

```
narrow(o, AdviceHome.class);
```



The narrow method might return a completely different object (or it might not). But regardless, you'll get a stub that really DOES implement the interface, so you can then cast it.

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

Now that we (finally) have the home stub, let's use it to get what we REALLY want...

1 Call create() on the home interface to get the EJB object stub

```
Advice advisor = home.create();
```

The create method returns a reference to the component interface, Advice. In other words, it returns a stub to the EJB object (which implements Advice, the Remote component interface for this bean.) You don't need to cast and narrow the EJB object stub.

2 Call a business method on the component interface (EJB object stub)

```
System.out.println(advisor.getAdvice());
```

Nothing special here. It's just a plain old method call on the reference to the Advice interface.

Well, not quite. Remember, every Remote method call declares a RemoteException! And that's a checked exception, so you MUST handle or declare it.

```
try {
  System.out.println(advisor.getAdvice());
} catch(RemoteException rex) {
     rex.printStackTrace();
}
```

Wait a minute... how come we didn't have to cast and narrow the EJB object stub, but we had to for the home stub?

C
V
N
c

there are no Dumb Questions

Q: OK, I know, I know, I don't need to learn the details of IIOP, but I still want to understand WHY they use it.

A: OK, a little more. IIOP, the wire protocol for CORBA, can represent more information than plain RMI. For example, IIOP can propagate both transaction and security information... important things that you can't send with a non-IIOP remote method call.

So IIOP lets a container at least have the potential for interoperating with other servers, including (possibly) one that isn't Java-based.

Remember, CORBA is a standard that (among other things) can give two objects, written in two different languages, a chance to invoke each other's methods.

This does not mean that your server is necessarily using IIOP. The spec says that YOU—the developer—have to assume the server is using IIOP, which means you have to be sure your beans are IIOP-compliant (we'll talk about IIOP compliance a little later in this chapter).

Q: If my server doesn't use IIOP, do I still have to do the whole narrowing thing?

A: Yes and no. Your code might work just fine with nothing more than a cast. But—and this is a really huge but—your client code won't be vendor-independent! In other words, you won't have a portable app if you don't use narrow, because redeploying the bean on a server that does use IIOP will break the clients.

Q: Is there any downside to using narrow? Especially if the server is not using IIOP?

A: No downside (well, whatever overhead there is wouldn't be worth the portability tradeoff). If your server isn't using IIOP, narrow() is most likely a no-op (i.e. do-nothing) method. The spec says to always narrow, and it won't hurt you if it isn't needed.

The declared return type of create() is the component interface, not Object.

So the EJB object comes back from create() already knowing what it is (an implementation of your component interface).

```
public Advice create()
```

The return type of the home interface create() method is ALWAYS the component interface

So the EJB object stub doesn't need a cast or a narrow..

Writing the Remote home interface for a session bean

Now that you've seen the lookup and create process from the client's point of view, we'll see what you have to do to write a home interface for your bean. For session beans, the process is very easy. In fact, for *stateless* session beans, it's ludicrously easy—you just declare a single, no-arg create() method.

```
package headfirst;

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


public interface AdviceHome extends EJBHome {

   public Advice create() throws CreateException, RemoteException;

}
```

Rules for the home interface

1. Import javax.ejb.* and java.rmi.RemoteException.
2. Extend EJBHome.
3. Declare a create() method that returns the component interface and declares a CreateException and RemoteException
   - For stateless session beans, there can be only one create(), and it must NOT have arguments.
   - Stateful session beans can have multiple, overloaded create() methods, and do NOT need to have a no-arg create().
   - All create() methods must declare a CreateException and RemoteException, but they can also declare other application (checked) exceptions.
   - The name of create methods in stateful beans must begin with "create" (createAccount(), createBigDog(), createFashionAdvisor(), etc.).
   - For stateful session beans, arguments must be RMI-IIOP compatible (you know, Serializable, primtive, Remote, or arrays or collections of any of those).

Remote home interface examples for session beans

The examples on this page are all legal examples of Remote home interfaces. You'll see some that could be both stateless and stateful, and some that could be only stateful (because they have a create method with arguments). We've dropped the package and import statements to put more on the page.

1
```
public interface CartHome extends EJBHome {
   public Cart create(String storeID) throws CreateException, RemoteException;
   public Cart create() throws CreateException, RemoteException;
}
```

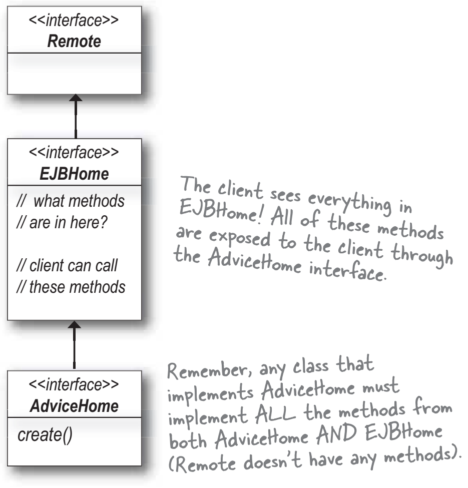
2
```
public interface MatcherHome extends EJBHome {
   public Matcher create(String customerID) throws CreateException, RemoteException;
   public Matcher createNewCustomer(String name, String login)
                                   throws CreateException, RemoteException;
}
```

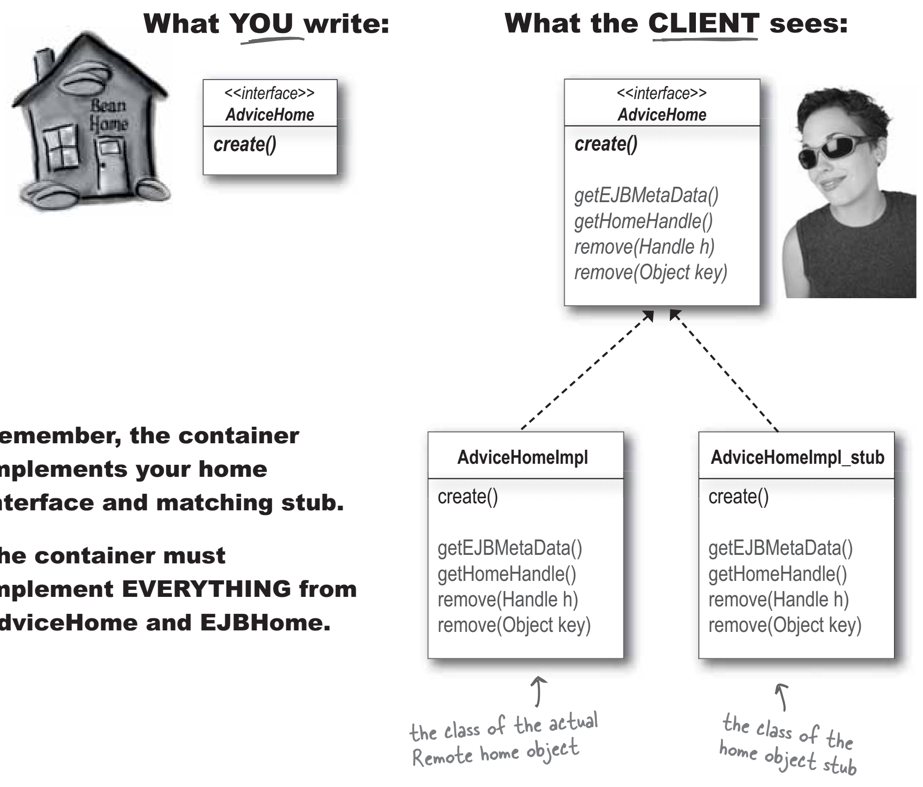
3
```
public interface TicketsHome extends EJBHome {
   public Tickets create() throws CreateException, RemoteException;
}
```

4
```
public interface ClubHome extends EJBHome {
   public Club createExisting(String clubID) throws CreateException, RemoteException;
   public Club createNewClub(String clubName) throws CreateException, RemoteException;
}
```

There's only one interface here that could be a stateless session bean's home—number 3. Notice, too, that number 4 has two create methods that both have the same argument—a String—but the methods are named differently to reflect what that particular create method is for.

I wonder what else I can do with the home interface. I know there's more than just the create method... what methods are in EJBHome?


<<interface>>
Remote
<<interface>>
EJBHome
// what methods
// are in here?
// client can call
// these methods
<<interface>>
AdviceHome
create()
The client sees everything in EJBHome! All of these methods are exposed to the client through the AdviceHome interface.
Remember, any class that implements AdviceHome must implement ALL the methods from both AdviceHome AND EJBHome (Remote doesn't have any methods).



Remember, the container implements your home interface and matching stub.

The container must implement EVERYTHING from AdviceHome and EJBHome.

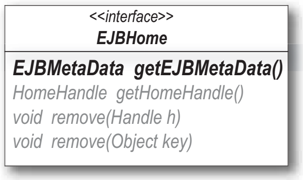
Note: these aren't necessarily the real names—the server generates these classes, and it can name them whatever it wants to.

If you're a client, and you want to...

Call this method:

1 **get reflection-like information about the bean.**

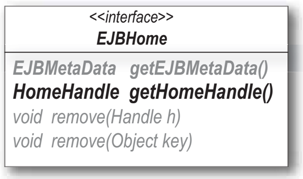
Unless you're a tool vendor, you'll probably never need to call this method. It returns the EJBMetaData interface—something you can use to get more specific class information about the bean. If you've got yourself an EJBMetaData reference (by calling getEJBMetaData), you can call getHomeInterfaceClass(), getPrimaryKeyClass(), isSession(), and more.



2 **serialize the home so that you can use the home again later, without having to go through JNDI.**

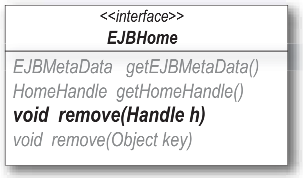
Imagine you're a client, and you've been working with a home—making a bunch of beans, calling home methods on entity beans, whatever. And now you have to reboot your machine. Or move to another machine. But you want to continue working with this home. What do you do?

You *could* go back through JNDI and do the whole lookup thing again. But if you ask the home for a handle, you'll get back a Serializable thing you can save and use later to get the home stub back without going through JNDI (more on handles a little later).



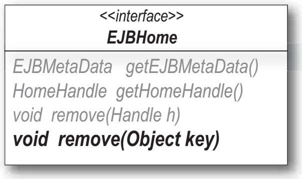
3 **tell the home you're done with a session bean.**

When you're done with a session bean, you can tell the home by calling remove() and passing the EJB object's handle. Yes that's right, the *EJB object.* Just as the home object can give you a handle (so that you can get the home stub back, later, without going through JNDI), the EJB object can give you a handle to itself. (We'll see more of that later in this chapter.) You *can* use this version of remove() for *entity* beans as well, but with entity beans, it's usually easier to call the *other* remove().



4 **tell the home to remove an entity bean.**

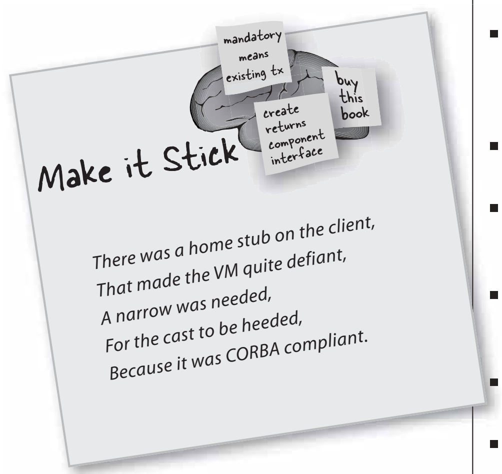
Notice we didn't say, "Tell the home you're done with an entity bean." That's because calling remove on an entity bean is drastically different from calling remove on a session bean. We'll get into the details in the entity bean chapters, but the short version is: when you remove an entity bean, you're not just telling the container that *you're* done with the bean, you're telling it that *everyone* is done with the bean. Forever. Because calling remove() on an entity bean means, "***Delete*** this entity from the persistent store." (Which usually means, "Delete this *row* from the *database*.")
This version of remove takes a primary key, which session beans don't have, so unlike the *other* remove(), *this* version can be used for entity beans *only.*



Hey, did I just see something about a primary key? In the home interface for a session bean? *That* can't be right...
I know, isn't that odd? But look... there's only one home interface, EJBHome, regardless of the bean type! There's no separate EJBSessionHome or EJBEntityHome--it's just EJBHome for everything!

Q: Does this mean the client has to know that she's using a session bean and not an entity bean? Isn't that something the client shouldn't have to know?

A: Yes, the client *does* have to know that when she's got the home interface for a session bean, he can't call the remove(Object primaryKey) method. If she does, she'll get an exception (javax.ejb.RemoveException). It does feel like more of an implementation detail than the client should have to know (i.e. that it's a session vs. entity bean), but in reality, you can't expect to write an EJB client without knowing whether you're communicating with a session or entity bean. For one thing, the way the client interacts with an entity bean home is completely different from the way a client uses a session bean home. You'll see dramatic differences when we get to the entity bean chapters.


Make it Stick
mandatory means existing tx
buy this book
create returns component interface

There was a home stub on the client,
That made the VM quite defiant,
A narrow was needed,
For the cast to be heeded,
Because it was CORBA compliant.

Make it Stick
mandatory means existing tx
buy this book
create returns component interface

Roses are red, blue is the sky,
you can't get a bean from JNDI.

It's only the home a client will spy,
when he does a lookup on JNDI.

BULLET POINTS

- The methods of the bean are exposed to the client through the component interface.
- The client can't directly get a reference to the bean; the client must go through the bean's EJB object, which implements the component interface.
- The client gets a reference to the bean's EJB object from the bean's home.
- To get the bean's home, the client does a lookup on JNDI, using the logical name under which the bean was deployed.
- To do a JNDI lookup, the client must first get an InitialContext, which is the entry point into the server's JNDI "virtual directory tree".
- For a Remote home interface, the stub returned from JNDI must be both cast and narrowed.
- Narrowing is the "exotic casting" needed for stub objects that come from a method that does not return the stub's client interface. Since the JNDI lookup returns type Object, the object returned from the lookup must be *narrowed* to the bean's home interface, and then *cast* to the bean's home interface.
- Narrowing is required for IIOP stubs (IIOP is the wire protocol for CORBA), because what's returned from the lookup might not be capable of implementing multiple interfaces, and thus would know only about the methods in type Object. Narrowing returns an object that implements the home interface.
- The home interface extends EJBHome, which has four additional methods the client can see: getEJBMetaData, getHomeHandle, remove(Handle h), remove(Object primaryKey). The remove(Object primaryKey) must not be called on a session bean.

Sharpen your pencil

Based on the rules for session bean home interfaces, which statements are true about this interface:

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

public interface CartHome extends EJBHome {

   public Cart create() throws CreateException, RemoteException;

}
```

- ☐ CartHome must not be the home of a stateful session bean.
- ☐ The interface is missing an import statement.
- ☐ The create method is missing an exception.
- ☐ Cart must be the class type of the bean.
- ☐ Cart must be the interface that extends EJBObject.
- ☐ The object returned from create() must be narrowed.
- ☐ The object returned from create() does not need a cast.

You MUST be able to look at code and infer a lot about it.

The exam expects you to look at client, interface, or bean code, and make inferences about things you don't see. You MUST know all of the rules for home and component interfaces. And there's more...

But enough about the home... let's talk about the EJB object. The component interface. The thing you REALLY want.

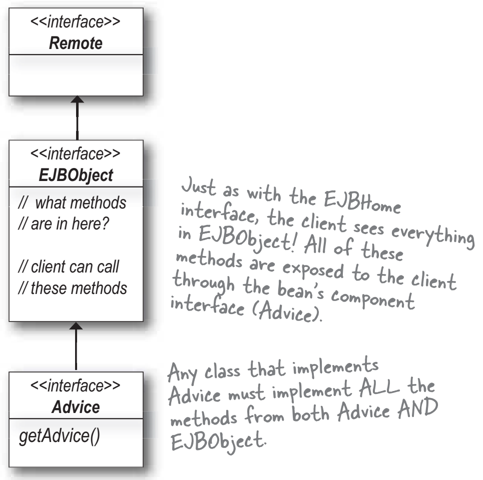
Remember, all that InitialContext-JNDI-lookup-cast-narrow-create business was just to get what you really wanted all along—something with the business methods. Something you can use to get the bean to do whatever it is that bean was created for. Number crunching, online shopping, advice.

```
package headfirst;

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


public interface Advice extends EJBObject {

   public String getAdvice() throws RemoteException;

}
```

Rules for the component interface

1. Import javax.ejb.* and java.rmi.RemoteException.
2. Extend 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).

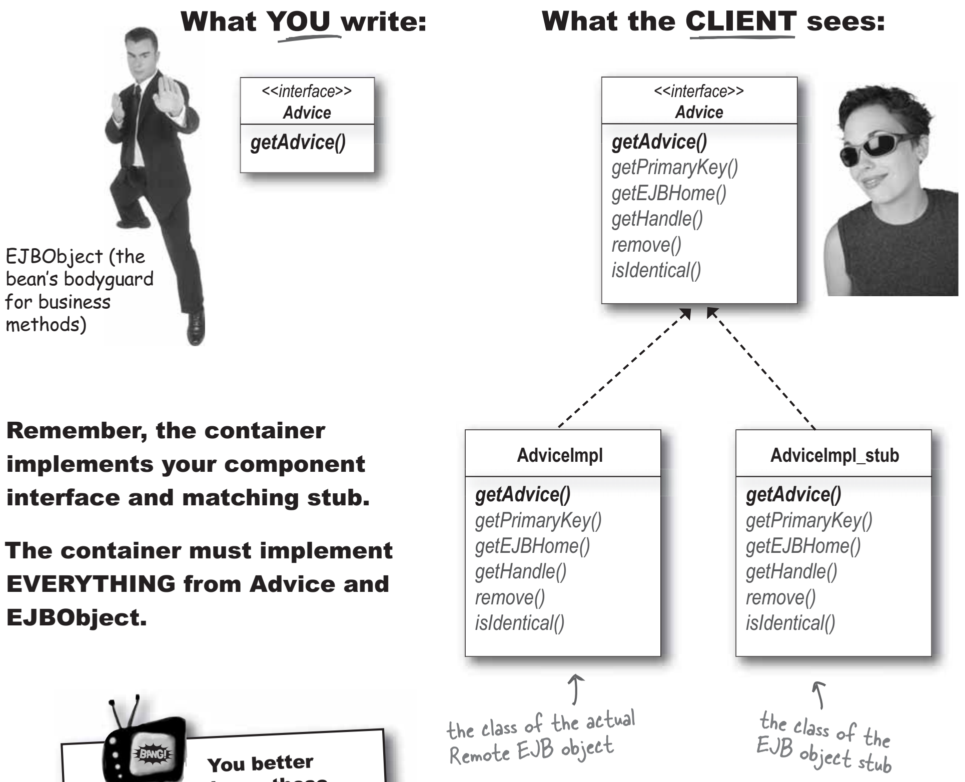
I wonder what else I can do with the component interface. I know there's more than just the business methods... what methods are in EJBObject?


<<interface>>
Remote
<<interface>>
EJBObject
// what methods
// are in here?
// client can call
// these methods
<<interface>>
Advice
getAdvice()
Just as with the EJBHome interface, the client sees everything in EJBObject! All of these methods are exposed to the client through the bean's component interface (Advice).
Any class that implements Advice must implement ALL the methods from both Advice AND EJBObject.

Imagine what else you might want to do with your EJB object reference...

You're a client. You have a reference to the AdviceBean's component interface. You know you can call getAdvice(). But now that you've gone to all the trouble of getting the stub, are there other things you might want to do?

I can think of some things... like, what if I have the bean, but I lost the reference to the home, and now I want to make more beans of that type? Surely the bean knows its own home, right? I can't believe they would have been stupid enough to make you go back through JNDI...



Remember, the container implements your component interface and matching stub.

The container must implement EVERYTHING from Advice and EJBObject.

You better know these other methods inside and out.

The exam expects you to know all five of the methods in EJBObject. You have to know that they're available to Remote clients, and exactly how they're used!

If you're a client, and you want to... call this method:

1 get the primary key of an entity bean

We won't go into this now, since we have 10 million pages on entity beans coming up. Just know that this does not apply to session beans, which don't have a unique identity exposed to the client. But if the client somehow gets a reference to an entity bean (maybe by searching on a customer's name) and wants the actual primary key, this is the method to call. Try it on a session bean, and you'll get a big fat RemoteException (or EJBException if the client is local).

<<interface>>
EJBObject

Object getPrimaryKey()
EJBHome getEJBHome()
Handle getHandle()
void remove()
boolean isIdentical(Object o)

2 get the bean's home

Imagine you've got a bean, but you don't have the bean's home. And now you want to make more beans of that type. What do you do? You could do a JNDI lookup and get the home in the usual way. But what if you don't have enough information to do the JNDI lookup? You can ask the bean to give you a reference to its home. Even if you are capable of doing a JNDI lookup on the home, calling getEJBHome() on the bean is more efficient.

<<interface>>
EJBObject

Object getPrimaryKey()
EJBHome getEJBHome()
Handle getHandle()
void remove()
boolean isIdentical(Object o)

3 save a reference to the EJBObject

You're shopping online, carefully putting items in your cart, after hours of painstaking research and decision-making on whether your girlfriend will prefer you in cornflower blue, or the Martha Stewart seafoam green. But before you can finish, you have to switch to another machine. No problem. You can ask the bean for a handle to the EJB object. You can use the handle to get back to your original EJB object and keep shopping.

<<interface>>
EJBObject

Object getPrimaryKey()
EJBHome getEJBHome()
Handle getHandle()
void remove()
boolean isIdentical(Object o)

4 tell the bean you're done with it

When you're finished with the bean, it's good manners to tell it you're done, so the container can free up any resources it might be keeping on your behalf. DANGER!! We're talking only about session beans here. Although you can call remove() on an entity bean, remember, it has a very different meaning (we'll see that in the entity bean chapters).

<<interface>>
EJBObject

Object getPrimaryKey()
EJBHome getEJBHome()
Handle getHandle()
void remove()
boolean isIdentical(Object o)

5 compare two EJB object references to see if they reference the same bean.

You've got two references to session bean EJB objects. Now you want to know if they're really references to the same bean. The isIdentical() method takes an EJB object reference and compares it to the EJB object on which you invoked isIdentical(), and returns true or false.

<<interface>>
EJBObject

Object getPrimaryKey()
EJBHome getEJBHome()
Handle getHandle()
void remove()
boolean isIdentical(Object o)

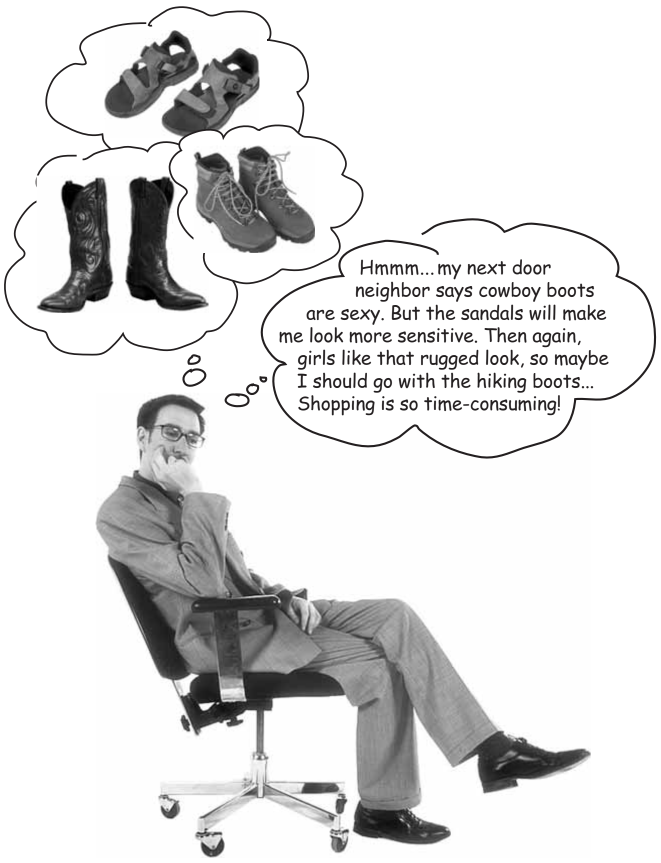
Online shopping should not be rushed...

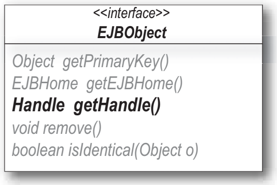
You're shopping. It's tough, because you can't decide whether you're a spring or a summer. You don't want to be rushed, but you've already got a bunch of stuff in your cart when you realize you're five minutes late for work.

You'd love to continue with your shopping once you get to work.

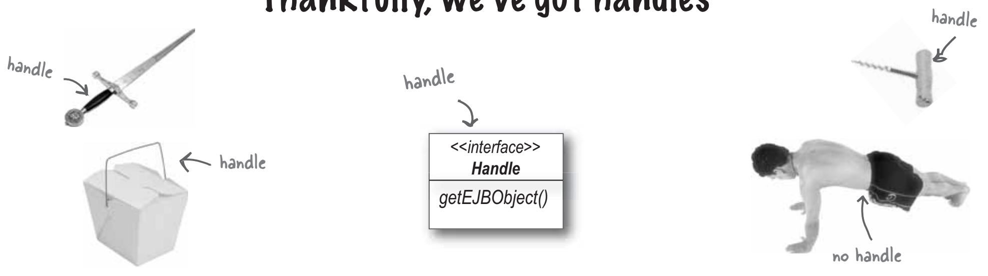
What do you do? If it were Amazon, your shopping cart would still be there when you log-in from the Web. But this is a proprietary Swing-based shopping client app you're using. How can you get your EJB object stub from your home machine to your work machine?

You could try serializing the stub. Yeah, that might work. Then again, it might not. The stub has a live network connection, and there's certainly no guarantee you can get that same connection, to the same EJB object again. And since that EJB object is the component interface for your own personal, temporary, shopping cart bean, you need a way to get back to your exact same EJB object again from work.


Hmmm... my next door neighbor says cowboy boots are sexy. But the sandals will make me look more sensitive. Then again, girls like that rugged look, so maybe I should go with the hiking boots... Shopping is so time-consuming!



Thankfully, we've got handles



A handle can rescue your shopping experience. Ask the bean (via the EJBObject interface) for a handle:

```
Handle myHandle = myCart.getHandle();
```

serialize it, email it to yourself, then deserialize on your work machine and you're back in business.

The handle is a Serializable thing that knows how to get back to the stub. It has a single method:

```
   public EJBObject getEJBObject()
```

So when you call it, you have to cast and narrow the stub that comes back! Remember, you always have to cast and narrow a stub unless the method that returns it has the actual Remote interface as its declared return type. Since the handle's method has no frickin' clue what your component interface is (say, ShoppingCart), you're faced with the same scenario you had with the home stub you got from the JNDI lookup() method. Cast and narrow. Cast and narrow. Cast and narrow.

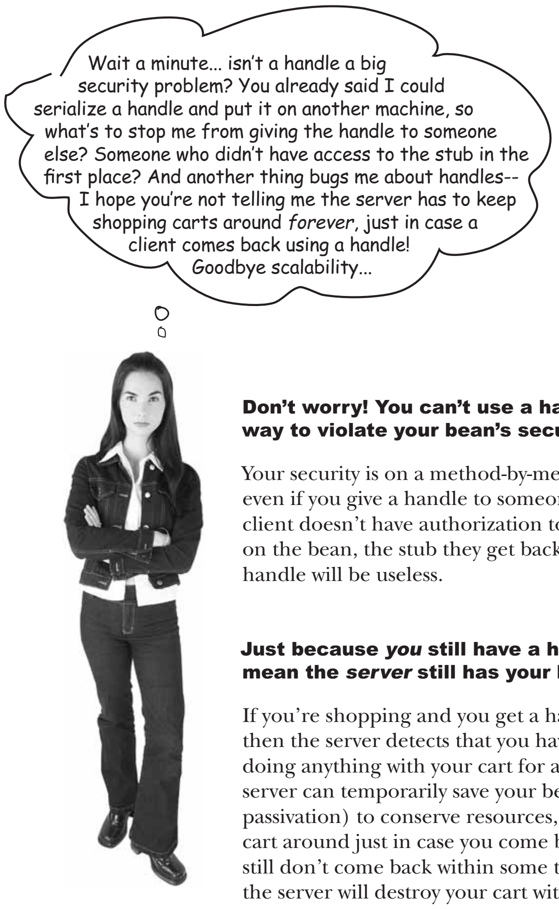
In your client code, you'll have something like:

```
// your code to get the serialized handle you saved earlier
Handle h = this.restoreTheHandle();

// now use it to get the EJBObject stub
Object o = h.getEJBObject();
Shopping cart = (Shopping) PortableRemoteObject.narrow(o, Shopping.class);
```

A handle is a Serializable object that knows how to get back to the original Remote EJB object. It's not a stub, but it can GET the stub.

It has just one method, getEJBObject(), that returns type Object.

So you have to narrow and cast the stub you get back!


Wait a minute... isn't a handle a big security problem? You already said I could serialize a handle and put it on another machine, so what's to stop me from giving the handle to someone else? Someone who didn't have access to the stub in the first place? And another thing bugs me about handles-- I hope you're not telling me the server has to keep shopping carts around *forever*, just in case a client comes back using a handle! Goodbye scalability...

Don't worry! You can't use a handle as a way to violate your bean's security.

Your security is on a method-by-method basis, so even if you give a handle to someone else, if that client doesn't have authorization to call methods on the bean, the stub they get back from the handle will be useless.

Just because *you* still have a handle, doesn't mean the *server* still has your bean.

If you're shopping and you get a handle, and then the server detects that you haven't been doing anything with your cart for a while, the server can temporarily save your bean (known as passivation) to conserve resources, but keep your cart around just in case you come back. But if you still don't come back within some time period, the server will destroy your cart with no hope of resurrecting it. That bean is history.

In that case, your cart won't be there when you call getEJBObject() on the handle, and you'll get a RemoteException.

isIdentical?

how to find out if two stubs refer to the same bean

These twins are identical if they're stateLESS, since they can each do the same thing and clients won't know the difference. But if they're stateFUL, then they are always distinct. Stateful twins cannot be identical, because they can hold information specific to their own unique client.

```
<<interface>>
EJBObject

Object  getPrimaryKey()
EJBHome  getEJBHome()
Handle  getHandle()
void remove()
boolean isIdentical(Object o)
```

If you've got two stubs, and you want to know if they refer to the same bean, you call isIdentical on one reference, passing in the reference you want to compare it against. Just like the way you use the equals() method.

The trick is, stateless session beans, stateful session beans, and entity beans each have different rules for what causes isIdentical() to return *true*.

Stateless session beans

True if both references came from the same home, even if the stubs are referring to two different Remote EJB objects! To the server, one stateless bean is as good as any other bean from the same home, because the client would never be able to tell the difference (since the bean can't hold any client-specific state).

Stateful session beans

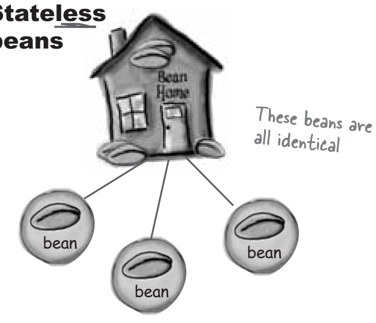
False no matter what, for any two unique stubs, even if from the same home. After all, my shopping cart isn't the same as yours!

Entity beans

True if the stubs refer to two entities with the same primary key.

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

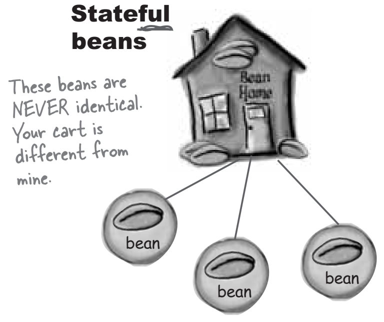
Stateless beans



isIdentical() always returns *true*, even for different beans.

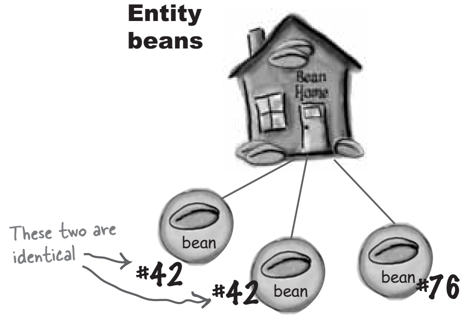
The isIdentical() method is like calling a Remote equals() method... except you're not asking if two objects on your heap are meaningfully equivalent, you're asking if two Remote objects are meaningfully equivalent--on the server!

Stateful beans


Bean Home
These beans are NEVER identical. Your cart is different from mine.
bean
bean
bean

isIdentical() always returns *false* for different beans, even if the beans are from the same home.

Entity beans



isIdentical() returns *true* for beans* that reference the same entity (in other words, the same primary key)

*We use the term "bean" here a little loosely because, conceptually, the server uses only one bean to represent a particular entity. So there would be only one bean with a primary key of #42, but clients may have multiple EJB object references to it.

there are no Dumb Questions

Q: Why *can't* you just use the equals() method instead of isIdentical()? Isn't that what equals() is for?

A: Remember, we're talking about Remote objects. The equals() method compares two objects on the same HEAP, where isIdentical() compares two Remote objects on the SERVER.

Q: I still don't see why they couldn't have just implemented the equals() method on the stub to do the same thing.

A: The equals() method is not a remote method, for one thing. You *can* always call equals() on a stub, because you can call it on any object on your heap. But it's not part of the remote interface, so it can't be a remote method (for example, it doesn't declare a RemoteException, etc.)

And remember, the equals() method is used to see if two objects on the *heap* are meaningfully equivalent. The vendor can implement the equals() method on the stub any way it likes, but that still doesn't tell you anything about what's going on back at the server end. Just because two stubs don't pass the equals() test, doesn't mean the server doesn't consider the two EJB objects to be identical (or referencing identical beans).

Your server may be using RMI stubs, for example, that have no logic for how their comparisons relate to meaningful comparisons of two EJB objects on the server. RMI stubs know about Remote objects, but they don't know what those Remote objects *represent*.

Q: How come there's a method in the EJBObject interface for getting the bean's home? If you don't HAVE the home, then how did you get the bean in the first place???

A: There are other ways to get a reference to an EJB object. It's true that you can't use JNDI to look up the EJB object; only the home is registered.

BUT... there's nothing to stop you from passing an EJB object reference as an argument or return value. You might have a business method in one bean, whose sole job is to hand you back a reference to an EJB object for a *different* bean.

Now suppose you have this EJB object reference, to a bean whose home you never had, and now you want to make more of those beans for yourself. You can do that by asking for the bean's home using getEJBHome().

And even if you *do* have enough information to do a lookup in JNDI for that bean's home, JNDI lookups are expensive. You'll save some overhead if you just get the home reference from the bean directly.

Q: Why *can't* you serialize the stub? Why do we need handles?

A: We didn't say the stub wasn't Serializable. But even if you can serialize it, that doesn't mean it's got enough information to get you back to the same (or meaningfully identical) EJB object. When the stub comes over from the server, it's already knowledgeable about how to contact a particular Remote object. When that stub is recreated, that exact Remote object might not even exist any longer.

Q: Then how would the handle be any better?

A: The handle has the 'smarts' to communicate with the server and get back something that is *just the same* as the EJB object you had before. In other words, it might *not* be the same EJB object, but the client will never be able to tell the difference.

mandatory means existing tx
buy this book
create returns component interface

Make it Stick

In your stateless home,
write one and no more,
but the client will see
an additional four.
[methods]

Roses are red, when they are alive,
From EJBObject, the client gets five.
[methods]

Watch it!

In EJB, there's can and there's can.

On the exam, some folks are confused by the word "can", as in "Can a session bean client call getPrimaryKey() on the EJB object stub?"

Depends on how you define "can". And for the exam, the answer would be NO! In other words, just because something is legally exposed to the caller, which means the caller can COMPILE with the method call, if the spec says you can't, then you can't. So just because you ***can*** *(compile) doesn't mean you* ***can*** *(according to the spec). Remember, there's compiler law, and then there's EJB spec law. On the exam, we're looking for EJB law. If the question DOES involve compilation, you'll know from the wording.*

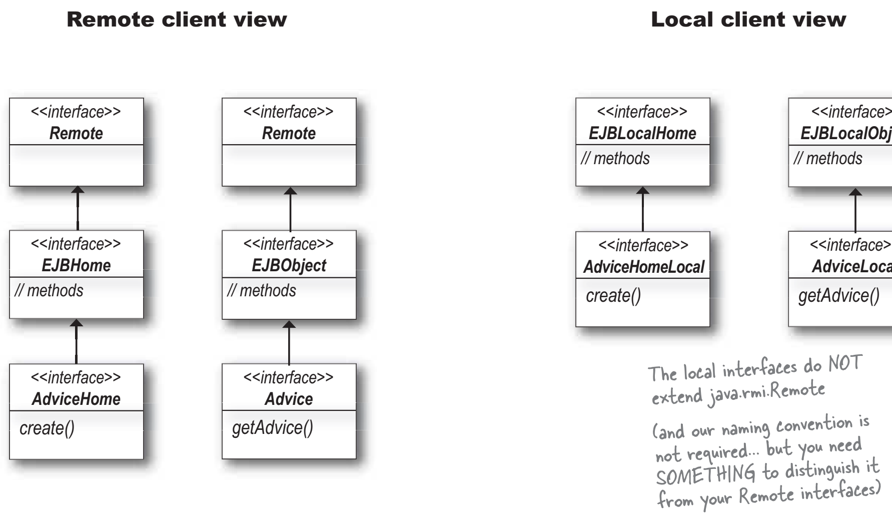
BULLET POINTS

- You expose your bean's business methods in the component interface.
- Remote component interfaces must extend javax.ejb.EJBObject.
- The client gets a reference to the bean's EJBObject by calling a method on the bean's home interface.
- References to both stateless and stateful session beans are retrieved from the home's create() methods.
- From the EJBObject interface, the client sees five additional methods: getEJBHome(), getHandle(), remove(), isIdentical(), and getPrimaryKey().
- Only entity bean clients are allowed to call getPrimaryKey() on the bean's component interface. Session bean clients will get a RemoteException.
- The getEJBHome() method returns a reference to the bean's home interface, so that the client doesn't have to go through a JNDI lookup, if they want to make more beans of that type.
- The getHandle() method returns a Serializable object that can be used later to reestablish contact with the server, and get back the stub to the component interface that the client used to get the handle.
- The handle has one method, getEJBObject(), that returns the Remote stub as type EJBObject. That means the stub must be cast and narrowed, just as you must do with the home stub that you get from a JNDI lookup.
- The isIdentical() method is kind of like doing an equals() method on the server. It returns true for two different stateless beans from the same home, false for two different stateful beans from the same home, and true for references to entities with the same primary key.

A bean's client interfaces can be local

We've looked at only the Remote client interfaces for a bean so far, but as of EJB 2.0, session and entity beans can expose a *local client view.* In other words, client interfaces that do not extend java.rmi.Remote.

What does this mean? That the home object and EJB object are not Remote objects! They're running in the same JVM as the client and the bean. In the entity bean CMR chapter, we'll look at why local interfaces were added to the spec. For now, think of them as a very special case.


Remote client view
Local client view
<<interface>> Remote
<<interface>> Remote
<<interface>> EJBHome
// methods
<<interface>> EJBObject
// methods
<<interface>> AdviceHome
create()
<<interface>> Advice
getAdvice()
<<interface>> EJBLocalHome
// methods
<<interface>> EJBLocalObject
// methods
<<interface>> AdviceHomeLocal
create()
<<interface>> AdviceLocal
getAdvice()
The local interfaces do NOT extend java.rmi.Remote
(and our naming convention is not required... but you need SOMETHING to distinguish it from your Remote interfaces)

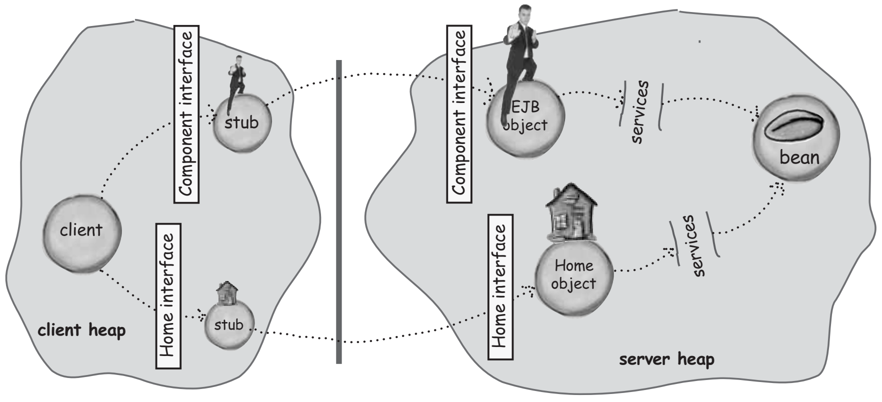
BRAIN POWER

The Remote interfaces EJBHome and EJBObject have more methods than the local interfaces EJBLocalHome and EJBLocalObject. Flip back through this chapter and look at the methods for EJBHome and EJBObject, and try to work out which methods in those Remote interfaces might be inappropriate or not needed in the local interfaces.

Think about the implications of having the interfaces local to the client...

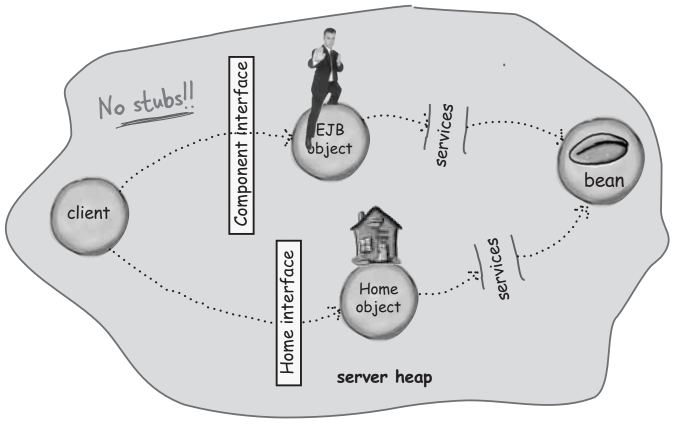
Remote vs. local client view

REMOTE client view



LOCAL client view

The client still can't get to the bean directly, because the server still needs a place to intercept the call to the the bean (so the server can add services). But this time, the client has a local reference to the home and component interface objects.



Sharpen your pencil

Given that the client has a plain old everyday Java reference to the home and component interfaces (i.e. the home object and the EJB object), which of the Remote interface methods do you think are appropriate for the local client view?

In other words, which methods of EJBHome are not in EJBLocalHome, and which methods of EJBObject are not in EJBLocalObject? ***Why?***

<<interface>>
EJBHome

- ***EJBMetaData getEJBMetaData()***
- ***HomeHandle getHomeHandle()***
- ***void remove(Handle h)***
- ***void remove(Object key)***

Of the four methods in EJBHome, EJBLocalHome has only one. *Which one?*

<<interface>>
EJBLocalHome

1 ____________

<<interface>>
EJBObject

- ***Object getPrimaryKey()***
- ***EJBHome getEJBHome()***
- ***Handle getHandle()***
- ***void remove()***
- ***boolean isIdentical(EJBObject o)***

Of the five methods in EJBObject, EJBLocalObject has only four (and one of the four is slightly different). *Which four?*

<<interface>>
EJBLocalObject

1 ____________
2 ____________
3 ____________
4 ____________

Which methods make sense for the local client interfaces?

You already know that the local client interfaces are missing some of the methods from their Remote counterparts. Let's figure out which ones are missing, and why.

Do we need handles with local interfaces?

Remember why handles exist in EJB—to give you a Serializable object that you can use to re-establish a stub to the EJB object you'd been working with. The handle is just an abstraction of a remote connection. So... does this make sense on a local client?

Do we need EJBMetaData with local interfaces?

Remember what EJBMetaData is used for—to get reflection-like info about a bean. If you call getEJBMetaData() on a bean's Remote home, you get back an object that implements EJBMetaData. That interface (EJBMetaData) has methods that let you interrogate the bean and learn more about the classes that make up the component. Would a local client ever need EJBMetaData?

Do we need isIdentical() with local interfaces?

Remember why isIdentical() exists—to let you compare two home or component interface references to see if they refer to "meaningfully equivalent" beans on the server. Would you need to use isIdentical() on a local client? (Big Hint: the server is free to implement .equals() any way it chooses...)

Do we need primary key information with local interfaces?

Remember why primary keys exist—to uniquely identify entity beans. Would you ever need to identify an entity bean on a local client?

Do we need remove methods with local interfaces?

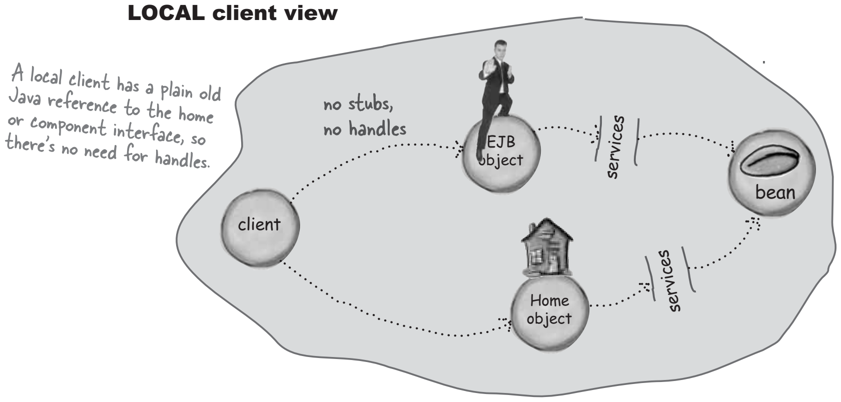
Remember what remove() is used for—to tell the container that you're done with a bean, for Session beans, or to tell the container to permanently delete the entity, for entity beans. Would a local client need to call remove() on a bean?

When you think handle, think Remote

Local clients don't need handles!

Local clients have no use for a handle, because handles are strictly for getting a savable (Serializable) object that knows how to reestablish communication with the Remote object.

LOCAL client view


A local client has a plain old Java reference to the home or component interface, so there's no need for handles.
no stubs, no handles
EJB object
services
bean
client
Home object
services

Who needs EJBMetaData when you've got reflection?

Local clients don't need EJBMetaData!

With the Java reflection API, you can interrogate an object to get all sorts of information about its class. With Remote objects, you don't have that option, because you can't get a reference to the class of the Remote object. The only thing you can interrogate on a Remote client are the stub objects, but they can't tell you anything about the *real* EJB object or Home object.

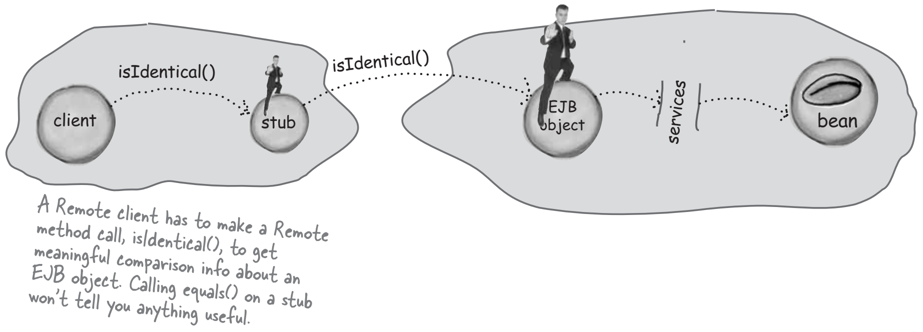
So while a Remote home client has to use EJBMetaData (the interface returned from the EJBHome getEJBMetaData() method) to get info, a local client will simply use the Java reflection methods (getClass(), etc.).

Do you need isIdentical() when there's equals()?

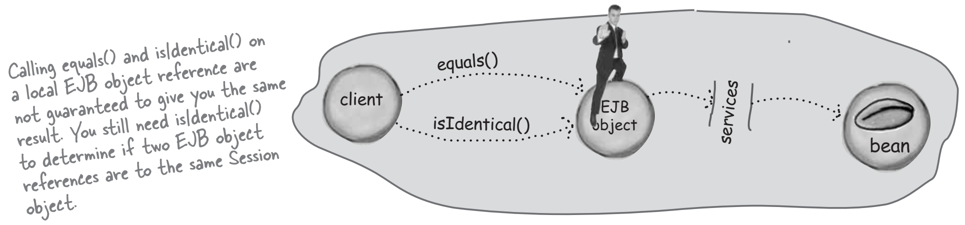
Local clients still need isIdentical()

Remember for a Remote client, the only local comparison you do is on two stub objects, using equals(). This doesn't work when you want to compare something back on the server, in this case the two EJB object references. That's what isIdentical() is for. But local clients have the real thing! They have the *real* reference to the EJB object, so they can use equals(), to see if two EJB object (local) references are meaningfully equivalent. But... that's still not what you want. There is no guarantee in the spec, for the results you'll get with .equals()! So while it *seems* like you could just use .equals() rather than isIdentical() with a local client, the spec does not guarantee that the results will be the same. Bottom line: if you want to know if two EJB object references are referencing the same session object, you have to use isIdentical() even when the EJB object is local.

REMOTE client view


isIdentical()
client
stub
isIdentical()
EJB object
services
bean
A Remote client has to make a Remote method call, isIdentical(), to get meaningful comparison info about an EJB object. Calling equals() on a stub won't tell you anything useful.

LOCAL client view


Calling equals() and isIdentical() on a local EJB object reference are not guaranteed to give you the same result. You still need isIdentical() to determine if two EJB object references are to the same Session object.
equals()
client
isIdentical()
EJB object
services
bean

Why so many remove methods?

For Remote clients, two in the home, plus one in the component interface

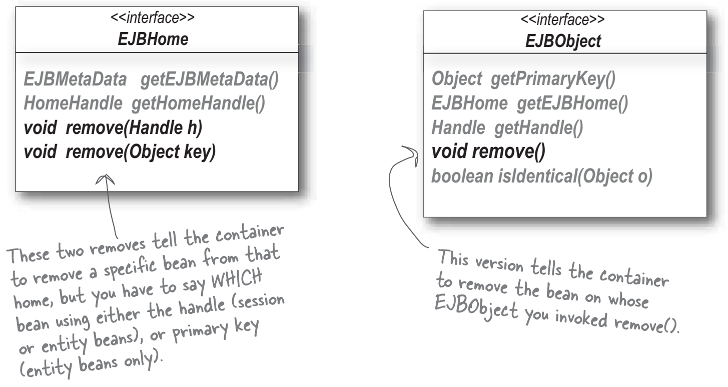
Remember, Remote clients have three remove() methods available, two in the home, and one in the component interface. The remove() that comes from EJBObject is simple; if you call it, you're saying you want to remove *that very bean*! In other words, the bean whose EJBObject you used to call remove(). And for session beans, remember, calling remove() simply tells the container that you're done with the bean. It's good manners, and it improves scalability since the server can stop keeping client-specific resources on your behalf, rather than waiting, say, for your shopping session to time out from inactivity.

But things aren't so simple when you call remove on a home. For one thing, you actually *can't* remove a home! The server keeps the bean home alive whether you're around or not, so there's no significant client-specific resources. There's no need to tell the server you're done with the home, because the server would simply say, "So what?"

Then what does it mean to call remove() on a home?

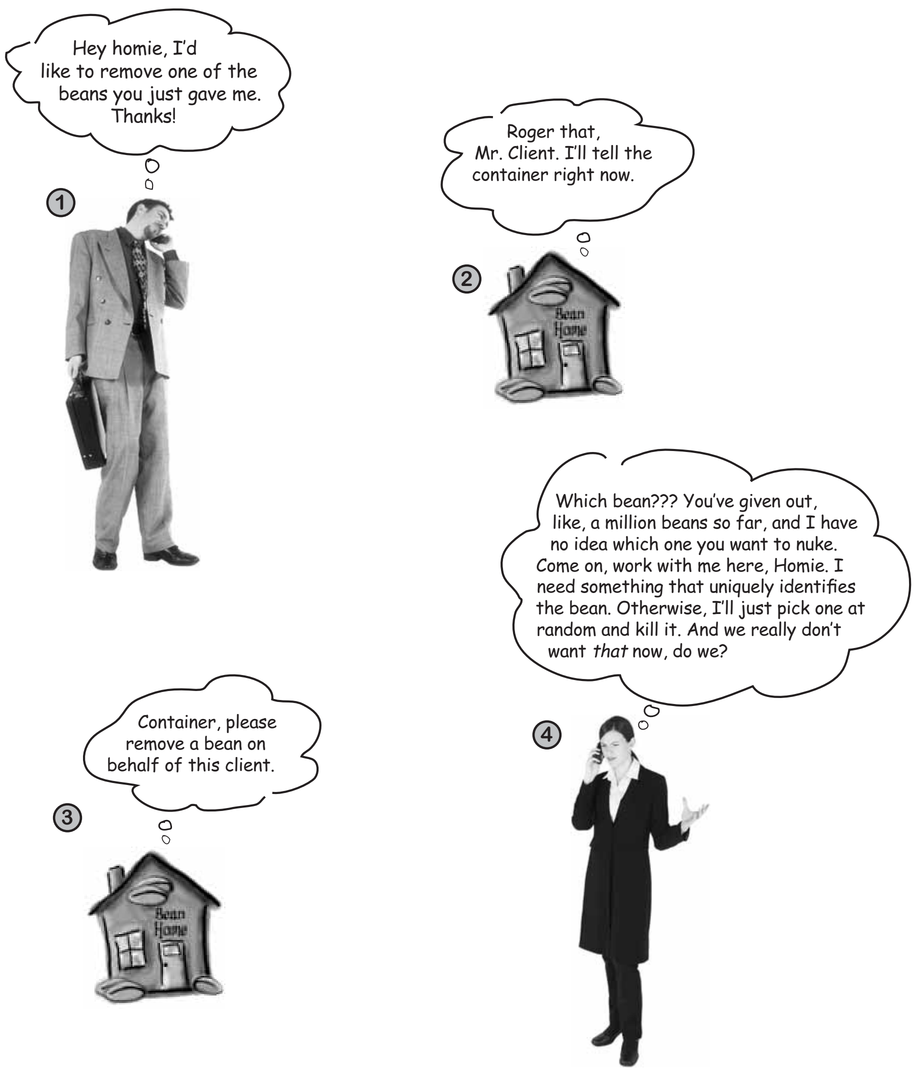
It means you're telling the home to remove one of the beans that *came* from that home. And that means you have to identify *which* bean you're talking about!

REMOTE client view



<<interface>>
EJBHome
EJBMetaData getEJBMetaData()
HomeHandle getHomeHandle()
void remove(Handle h)
void remove(Object key)
<<interface>>
EJBObject
Object getPrimaryKey()
EJBHome getEJBHome()
Handle getHandle()
void remove()
boolean isIdentical(Object o)
These two removes tell the container to remove a specific bean from that home, but you have to say WHICH bean using either the handle (session or entity beans), or primary key (entity beans only).
This version tells the container to remove the bean on whose EJBObject you invoked remove().

Why there's not a no-arg remove method in the home...


1
Hey homie, I'd like to remove one of the beans you just gave me. Thanks!
2
Roger that, Mr. Client. I'll tell the container right now.
Bean Home
3
Container, please remove a bean on behalf of this client.
4
Which bean??? You've given out, like, a million beans so far, and I have no idea which one you want to nuke. Come on, work with me here, Homie. I need something that uniquely identifies the bean. Otherwise, I'll just pick one at random and kill it. And we really don't want *that* now, do we?

How can you use a remove that takes a handle when you don't have a handle?

Local clients don't have handles, so local homes don't have a remove() that takes a handle.

If local home interfaces don't have handles, then there's no way you could have a remove method that takes a handle. Because in order to pass a handle to the home, that uniquely identifies the bean you're trying to remove, you'd have to first get the bean's handle. And since locally-exposed beans don't have handles... you see the problem.

You don't call remove on a home to remove the home.

You call remove on a home to tell the home to remove a bean.

A bean from that home type.

That means you must uniquely identify the bean you want removed, when you call a home remove method. For entity beans, use the primary key or a handle, and for session beans, use a handle.

But handles only work for Remote home clients...

there are no Dumb Questions

Q: What if you have a local home, but you have a Remote EJB object reference? Can't you pass the Remote bean's handle to the local home?

A: NO!! Because you can't mix local and Remote interfaces together. Only a local bean comes from a local home, and vice-versa. So it will NEVER be possible to have a Remote bean's handle, to give to that same Remote bean's home, unless that home is also Remote.

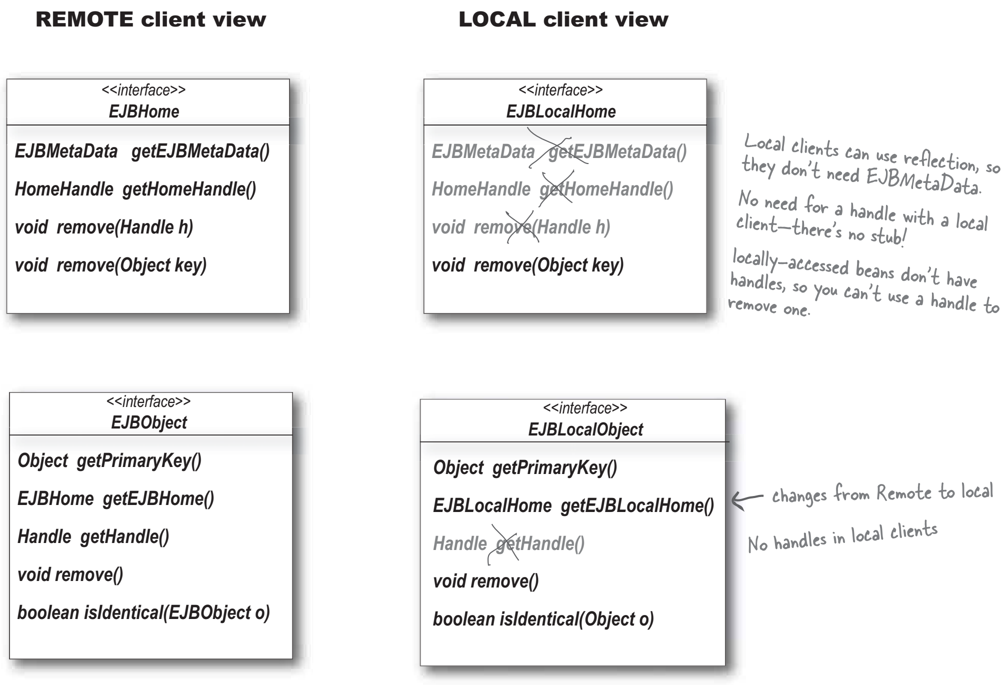
We didn't say that very well, did we... OK how about this—a *Remote* home will hand out only *Remote* references to the component (EJBObject) interface for that bean type, and a local home will only return local references to the component (EJBLocalObject) interface.

Q: Tell me again why you can't remove the home.

A: There's never a reason to remove the home (in other words, to tell the container you're done with it), because the container must keep the home around with or without your interest. So if you were able to say remove to the home, the container would say, "Gee...don't flatter yourself buddy. What I do here on the server is not ABOUT you. I could care less when you're done with your home reference."

Comparing Remote vs. Local interfaces

The EJBObject and EJBHome interfaces have more methods than the EJBLocalObject and EJBLocalHome interfaces because there are methods that don't make sense in a local context.


REMOTE client view
LOCAL client view
<<interface>> EJBHome
EJBMetaData getEJBMetaData()
HomeHandle getHomeHandle()
void remove(Handle h)
void remove(Object key)
<<interface>> EJBLocalHome
EJBMetaData ~~getEJBMetaData()~~
HomeHandle ~~getHomeHandle()~~
void ~~remove(Handle h)~~
void remove(Object key)
Local clients can use reflection, so they don't need EJBMetaData.
No need for a handle with a local client—there's no stub!
locally-accessed beans don't have handles, so you can't use a handle to remove one.
<<interface>> EJBObject
Object getPrimaryKey()
EJBHome getEJBHome()
Handle getHandle()
void remove()
boolean isIdentical(EJBObject o)
<<interface>> EJBLocalObject
Object getPrimaryKey()
EJBLocalHome getEJBLocalHome()
Handle ~~getHandle()~~
void remove()
boolean isIdentical(Object o)
changes from Remote to local
No handles in local clients

Sharpen your pencil

Based on what you now know about the difference between local and Remote client interfaces, decide if the following statements are true or false. You'll have to make some inferences and smart guesses for some of them.

Select all that are true:

- ☐ The only way to remove a local session bean is through the component interface
- ☐ Entity beans can be removed through a local home interface
- ☐ If you see an isIdentical() call, this must be a local bean
- ☐ If you see a getHandle() call, this must be a Remote bean
- ☐ If the client is catching a RemoteException on a home method, the bean's home interface must extend EJBLocalHome
- ☐ If the client is not handling a RemoteException on a business method, the bean's component interface must extend EJBObject.
- ☐ If you see a call to getEJBMetaData(), the bean's component interface must extend EJBLocalObject.
- ☐ If you do a JNDI lookup on a local home, you must narrow the object returned from JNDI
- ☐ There are three methods in the EJBLocalObject interface
- ☐ There are two methods in the EJBLocalHome interface

Writing the local client interfaces

Now that we've covered what the client sees in a local interface, let's look at your responsibility as a Bean Provider. In other words, what *you* have to do to write the local interfaces for your bean.

Component interface:

```
package headfirst;
import javax.ejb.*;

public interface AdviceLocal extends EJBLocalObject {

   public String getAdvice();
}
```

extend EJBLocalObject instead of EJBObject

No RemoteException!!

this return type doesn't need to be RMI-IIOP compliant (although it certainly can be, of course, like String)

Home interface:

```
package headfirst;
import javax.ejb.*;

public interface AdviceHomeLocal extends EJBLocalHome {

   public AdviceLocal create() throws CreateException;
}
```

extend EJBLocalHome instead of EJBHome

This MUST be the local component interface!

still needs a CreateException but no RemoteException

Rules for local interfaces

1. Import javax.ejb.* (or use fully-qualified names).
2. Extend EJBLocalObject (for the component interface) or EJBLocalHome (for the home interface).
3. Declare one or more business methods in the component interface.
4. All create methods in the local home must return the local component interface, and declare a CreateException.
5. Any method you declare in the home or component interface can declare your own application exceptions, which must be compiler-checked exceptions (i.e. not subclasses of RuntimeException)..
6. You must NOT declare a RemoteException for any methods

You can have both a Remote and local client view for a bean, but you probably won't.

We mentioned earlier that local interfaces are a very special case for a client view. They were introduced with version 2.0 of the EJB spec (which this book is based on) and the original intent was to support container-managed relationships in entity beans. But enough customers and vendors asked for the ability to have non-Remote interfaces for beans, so the J2EE team decided to make it available for session beans as well as entity beans.

But regardless of which you choose, it's *very* unlikely you'll have a design that requires both a local and Remote client view. If your bean is in a container-managed relationship with another entity bean (you'll learn all about this in the entity chapters), you have no choice. The bean must expose itself locally. And in that case, it's almost impossible to think of a reason to also have that same bean exposed to Remote clients for other purposes.

Just know that it *is* legal to have both.

But you can never, ever, ever mix and match.

A *Remote* home interface can give out only *Remote* component interface references (in other words, a stub to the Remote EJBObject). A *local* home can give out only *local* component interface references (in other words, a regular Java heap reference to the EJBLocalObject).

You can't mix local and Remote interfaces together for the same bean, even though a given bean can expose both a Remote and local client view.

But a Remote home will give out only Remote component interface references (stubs), and a local home will give out only local component interface references (regular Java references).

Sharpen your pencil

Change the AdviceClient from a Remote client to a local client, using the local interfaces for AdviceLocal and AdviceHomeLocal. Do NOT turn the next page!!

```
import javax.naming.*;
import java.rmi.*;
import javax.rmi.*;
import headfirst.*;
import javax.ejb.*;

public class AdviceClient {

   public static void main(String[] args) {
      new AdviceClient().go();
   }

   public void go() {
      try {
         Context ic = new InitialContext();
         Object o = ic.lookup("Advisor");

         AdviceHome home = (AdviceHome) PortableRemoteObject.narrow(o, AdviceHome.class);

         Advice advisor = home.create();
         System.out.println(advisor.getAdvice());
       } catch (Exception ex) {
           ex.printStackTrace();
       }
   }
}
```

What has to change? Write the changed line (or lines) below:

Exceptions in client interfaces: what the client might get

A Remote interface must have RemoteException declared on every method. That means the client using a Remote interface must deal with RemoteException for every Remote method call. But local interfaces don't have that restriction. The only methods in a local client interface that must declare exceptions are the create() and remove() methods (for session beans; entity beans also have a finder method that declares a FinderException).

We have a whole chapter devoted to exceptions in EJB, so we won't go into the details now, but the essence is this: if a bean (or the Container) generates a runtime exception, Remote clients see the exception as a *checked* RemoteException, but local clients see it as an *unchecked* EJBException.

In addition to whatever other checked exceptions (called application exceptions in EJB) the interface methods declare, all Remote interface methods can throw a RemoteException and local client interface methods can always throw an EJBException. So Remote clients must wrap all calls to a home or component interface method in a try/catch, while local clients use a try/catch *only* if the interface method declares an application exception (which includes CreateException, RemoveException, FinderException, and any other exceptions the Bean Provider declares in the methods of the bean's client interfaces).

✓ Indicates a compiler-checked exception (i.e. non-RuntimeException)

	REMOTE client view	**LOCAL client view**
ALL methods	✓`javax.ejb.RemoteException`	`javax.ejb.EJBException`
CREATE methods	✓`javax.ejb.RemoteException` ✓`javax.ejb.CreateException`	`javax.ejb.EJBException` ✓`javax.ejb.CreateException`
REMOVE methods	✓`javax.ejb.RemoteException` ✓`javax.ejb.RemoveException`	`javax.ejb.EJBException` ✓`javax.ejb.RemoveException`

Local client code

Compare this to the code modifications you made on the previous sharpen. To help show that the calls to the home and component interface are no longer Remote, we've made the exception handling more fine-grained. Notice that we're not catching a RemoteException.

```
import javax.naming.*;
import headfirst.*;
import javax.ejb.*;

public class AdviceLocalClient {

    public static void main(String[] args) {
        new AdviceLocalClient().go();
    }

    public void go() {
        Object o = null;

        try {

           Context ic = new InitialContext();
           o = ic.lookup("AdvisorLocal");

        } catch (NamingException nex) {
              nex.printStackTrace();
        }

        AdviceHomeLocal home = (AdviceHomeLocal) o;
        AdviceLocal advisor = null;

        try {

           advisor = home.create();

        } catch (CreateException cex) {
              cex.printStackTrace();
        }

        System.out.println(advisor.getAdvice());

    }
}
```

we got rid of javax.rmi and java.rmi imports

You still have to go through JNDI, and do the lookup the usual way, only this time you get a reference to a real Java object on the heap (an instance of EJBLocalHome), instead of a stub to a Remote EJBHome object..

Here's a big change! No narrowing! Just a plain old cast (we still have to cast because the return type of lookup is Object, but we don't have to narrow it since it isn't a stub.)

The create() method still declares a CreateException, but not a RemoteException

The business method call is no longer a Remote method call. Just a normal local method call, and since getAdvice() doesn't declare any exceptions, the business method call doesn't have to be wrapped in a try/catch.

BRAIN POWER

What has to change inside the bean class?

We've seen how the interfaces change, and how the client code has to change, when you go from a Remote to local client view. But what about the bean class itself? What do you think? Does the bean code need to change, if you're going to deploy it with a *local* client view instead of a *Remote* client view? What if you plan to deploy it with *both* a local and Remote client view?

For now, let's assume that the only method that matters is the bean's business method. Here's how it looks in the original bean class:

```
 public String getAdvice() {
    System.out.println("in get advice");
    int random = (int) (Math.random() * adviceStrings.length);
    return adviceStrings[random];
}
```

Do you see anything in that method that looks specific to a Remote client view? Would you need to do anything different with a local client?

No, don't think so.

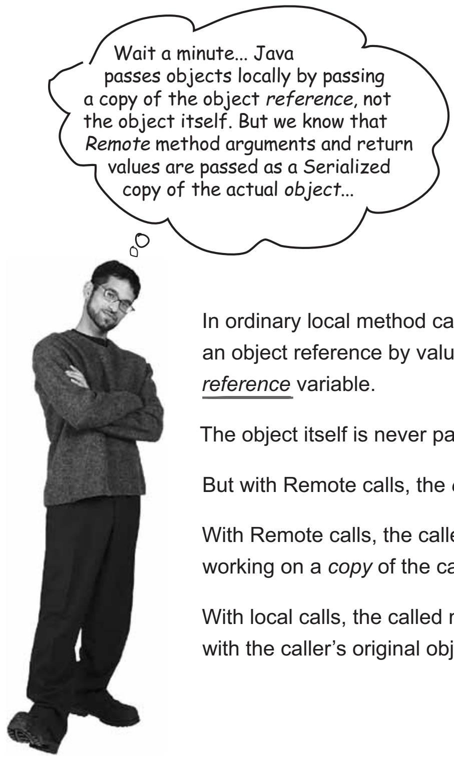
Anything that works as a return type or argument for a Remote method is guaranteed to work for a local method as well, so we're OK there. (Kind of a no-brainer when the return type is String, though.) OK, there is *one* exception—remember, according to Bean law you must not return a bean's Remote interface from a local interface method.

So it looks like (at least with this bean) we should never have to know or care. We should be able to deploy the bean as written, and the bean should be kept unaware of whether its clients are Remote or local.

Sounds good, doesn't it? Simple, clean, object-oriented.

But think about it some more. Imagine a bean with more complex logic. More business methods. Arguments to those methods. Arguments the method might even need to act on.

Hmmmmm... can you think of *anything* that the bean might want to treat differently, if it knew the client were local instead of Remote?


Wait a minute... Java passes objects locally by passing a copy of the object *reference*, not the object itself. But we know that *Remote* method arguments and return values are passed as a Serialized copy of the actual *object*...

In ordinary local method calls, Java passes an object reference by value, as a *copy* of the *reference* variable.

The object itself is never passed.

But with Remote calls, the *object* itself is copied.

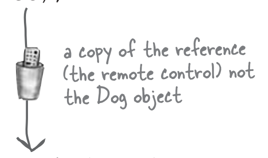
With Remote calls, the called method is always working on a *copy* of the caller's object.

With local calls, the called method is always working with the caller's original object—*not a copy!*

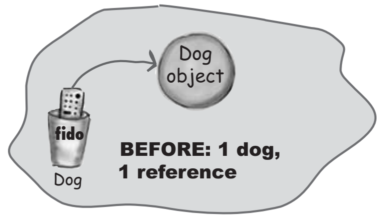
Arguments to Remote vs. local methods

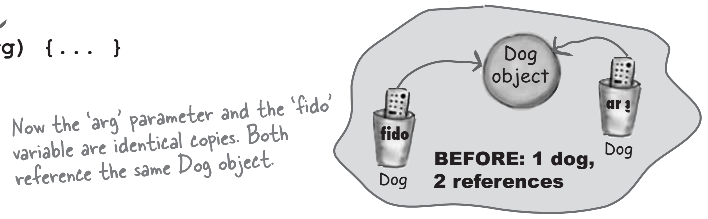
LOCAL method call

```
Dog fido = new Dog();
this.trainPet(fido);
```


a copy of the reference (the remote control) not the Dog object

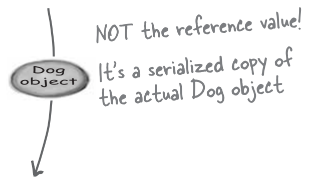
```
void trainPet(Dog arg) {... }
```

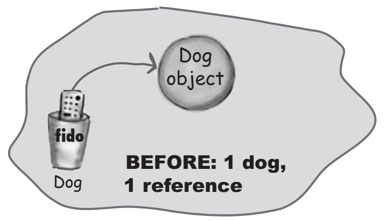



Now the 'arg' parameter and the 'fido' variable are identical copies. Both reference the same Dog object.
Dog object
fido
Dog
arg
Dog
BEFORE: 1 dog, 2 references

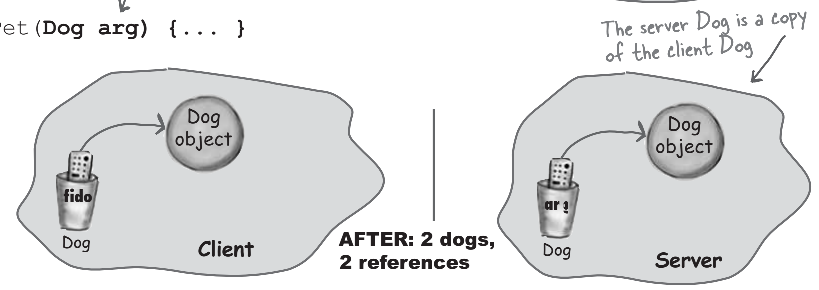
REMOTE method call

```
Dog fido = new Dog();
remoteStub.trainPet(fido);
```





```
void trainPet(Dog arg) {... }
```


The server Dog is a copy of the client Dog
Dog object
fido
Dog
Client
AFTER: 2 dogs, 2 references
Dog object
arg
Dog
Server

there are no Dumb Questions

Q: So, I'm still not clear if the bean MUST always know if the client is Remote or local.

A: It's not that the bean *must*, but rather that the bean *might* have to know. If it matters that the bean is working on a the caller's *object*, (via a copy of the caller's reference) as opposed to a *copy* of the caller's object, your bean code might have to change. And that goes for return values too. If it matters that the calling method gets back a copy of a reference vs. a copy of an object, the bean code might have to change.

Q: But I thought that choosing to deploy a bean with local vs. Remote client views was just a matter of switching a switch at deploy time.

A: NO! NO! NO! Let's imagine that you did write two sets of interfaces, one for a local client view and one for a Remote client view. It is true that at deployment you could decide which of the two views you wanted to expose (or both). But that works only if the bean code doesn't care where the client is. A bean method with no arguments or return values might be safe regardless of how the client is accessing it.

One solution might be to write the bean code assuming the bean is always getting a copy, and then if the client is local, have the client make a copy (clone) of the object before passing it. Or, for return values, you might always have the bean make a copy before handing it back. That way, the bean never has to worry that a local client might be modifying the bean's object.

Q: Then it's just about arguments and return values? Is there any other reason you couldn't deploy a bean and make the decision for Remote vs. local view at deploy time?

A: There is another reason. The client code! Even if the *bean* doesn't need to know how its client is accessing it, the *client* must know! A client written to access a bean locally wouldn't work if the bean's Remote client interfaces, and vice-versa.

Q: Why not?

A: The client must know in advance whether its accessing a bean's Remote or local client view, because the interfaces themselves are different. Polymorphically, you can't use the Remote and local views interchangeably, because the interfaces themselves are different.

There's no way the client can be kept blissfully ignorant*, because the behavior of the bean is different. Remember, a Remote client must handle RemoteExceptions, and narrow the Remote stub coming back from the lookup! And a Remote client is exposed to methods in the bean's client interfaces—methods that don't exist in the local interfaces. So a Remote client might, for example, try to call a getHandle() method on the local component interface, a method call that would never work.

And a local client won't have code to handle the RemoteExceptions or narrow the stubs.

The bottom line is that deploying a bean with a Remote vs. local client view is a Big Deal. It's a commitment. The client has to know in advance.

Q: Can you get around this by declaring your RemoteExceptions on your local interface? And could you make an interface that is both Remote and local... by having your component interface (like Advice) extend both Remote and EJBLocalObject? What's the harm if the client simply always handles RemoteExceptions, and always does the narrow()? That way the client shouldn't have to know.

A: Still won't work. For one thing, according to bean law, you're not allowed to declare RemoteExceptions on local methods! So there's no guarantee that your server would even *let* you deploy a bean with a local interface that declares RemoteExceptions. And there is no guarantee that the narrow() method would not cause problems. And then there are handles and all that other stuff... You need to just let this go.

BE the Container

Each of the code snippets on this page represents code from either an interface or a client. Your job is to play Container and decide whether each is legal according to both Java law and Bean law. In other words, even if the code compiles, it might still be WRONG to the Container, because it doesn't comply with the rules of the EJB spec. Assume that everything you do NOT see is legal and correct. Figure out if the problem is a compiler error or a problem to the Container, and figure out how to fix it.

A. In a local client:

```
public void go()  {
  Object o = null;
  try {

     Context ic = new InitialContext();
     o = ic.lookup("AdvisorLocal");

  } catch (NamingException nex) {
        nex.printStackTrace();
  }
  AdviceHomeLocal home = (AdviceHomeLocal) o;
  AdviceLocal advisor = null;
 // more stuff
}
```

B. In a Remote client:

```
public void go() {
  try {
   // look up the Advice bean, assign it
   // to advisor
  } catch (Exception ex) {
    ex.printStackTrace();
  }
  System.out.println(advisor.getAdvice());
}
```

C. In a bean's home interface

```
package headfirst;
import javax.ejb.*;

public interface AdviceHome extends EJBHome {

   public Advice create() throws CreateException;

}
```

D. In a bean's component interface

```
package headfirst;
import javax.ejb.*;

public interface AdviceLocal extends EJBLocalObject {

   public String getAdvice();

}
```

BULLET POINTS

- You can expose your bean to local clients using a local client view.
- Local component interfaces must extend javax.ejb.EJBLocalObject. Local home interfaces must extend javax.ejb.EJBLocalHome.
- Methods in local client interfaces do NOT declare RemoteException.
- Some of the interface methods exposed to Remote clients are not exposed to local clients.
- Local clients cannot get handles, since handles are used to re-establish a connection to the Remote object.
- EJBMetaData is not used with local clients, since a local client can use reflection to interrogate the EJB object and Home object.
- Local home interfaces have only one remove() method—the one that takes a primary key. The remove() that takes a Handle doesn't exist in the local home interface, since Handles aren't used with a local client view.
- Because the only remove() in the local home interface requires a primary key argument, local session bean clients cannot remove a bean using the bean's home; they can call remove() *only* on the bean's component interface.
- EJBLocalHome has only one method: remove() that takes a primary key, because the getHomeHandle(), getEJBMetaData(), and remove(Handle) methods that are in EJBHome don't apply to a local client view.
- The only method in EJBObject that is not also in EJBLocalObject is getHandle().
- Arguments and return values are passed by value when using a local client view. In other words, they're passed in the normal Java way (objects passed by a copy of the reference, primitives passed by a copy of the value).
- Local clients do not need to narrow the Home reference because it's a normal Java reference, not a stub to a Remote object.
- Local clients do not need to catch RemoteExceptions, since local interface methods don't declare RemoteExceptions.

Watch it!

You have to recognize a local vs. Remote client view!

Be prepared to look at bean code, client code, or interface code and know whether you're looking at a Remote or local client view.
And it might be subtle!
For example, you might see the client make a business method call, and the argument to the method is a non-Serializable object. Since you are to assume that the method is legal and correct, you KNOW that this must be a local interface—you can't pass a non-Serializable object as an argument or return value to or from a Remote method.
Some of other gotchas to tell you whether the bean is local or Remote include:
1. Client doesn't narrow the Home.
2. Client doesn't handle any checked exceptions on a business method call.
3. Client calls remove(Handle) on a bean's home.

Exercise Solutions

BE the Container

A. In a local client:

```
public void go()  {
  Object o = null;
  try {

     Context ic = new InitialContext();
     o = ic.lookup("AdvisorLocal");

  } catch (NamingException nex) {
       nex.printStackTrace();
  }
  AdviceHomeLocal home = (AdviceHomeLocal) o;
  AdviceLocal advisor = null;
 // more stuff
}
```

A is fine. Because it's local, it does not need to narrow the Home reference. Only a cast is needed.

B. In a Remote client:

```
public void go() {
  try {
   // look up the Advice bean, assign
   // it to advisor
  } catch (Exception ex) {
    ex.printStackTrace();
  }
  System.out.println(advisor.getAdvice());
}
```

Compiler error! getAdvice() is a remote method, but it isn't handling the RemoteException. If we moved the last line INSIDE the try block, it would work.

C. In a bean's home interface

Container error! EJBHome is a Remote interface, and the rule is that you must declare RemoteException on each method in the interface. The compiler doesn't care, but the Container will. At some point in the deploy process, this will fail.

```
package headfirst;
import javax.ejb.*;

public interface AdviceHome extends EJBHome {

   public Advice create() throws CreateException;

}
```

needs RemoteException

(look at the imports; no java.rmi.*)

D. In a bean's component interface

D works fine.. It's a local component interface, and it doesn't need to declare any exceptions on the business methods.

```
package headfirst;
import javax.ejb.*;

public interface AdviceLocal extends EJBLocalObject {

   public String getAdvice();

}
```

OK, needs no exceptions declared

COFFEE CRAM

Mock Exam

1 Which capabilities are provided by both remote and local home interfaces for session beans? (Choose all that apply.)

❏ A. Creating a session object.

❏ B. Removing a session object.

❏ C. Getting a session object's EJBMetaData.

❏ D. Getting a session object's handle.

2 When locating a session bean's remote home interface, which are steps that must occur to create a valid home interface reference? (Choose all that apply.)

❏ A. The session context must be narrowed, and the narrowed result cast.

❏ B. The result of the JNDI lookup must be narrowed, and the narrowed result cast.

❏ C. The initial context must be narrowed, and the narrowed result cast.

❏ D. The result of the JNDI lookup must be cast to an initial context, and then narrowed.

3 Given a remote client 'R', that has valid references to session beans 'A' and 'B', and given that A is a local client to B, which statements are true? (Choose all that apply.)

❏ A. R cannot pass his reference for A, to B.

❏ B. A cannot pass his reference for B, to R.

❏ C. A cannot invoke methods on B.

❏ D. B cannot invoke methods on R.

4 When comparing two session objects, what is true? (Choose all that apply.)

❑ A. Using the isIdentical() method, stateless session beans from the same home will always return true.

❑ B. Using the isIdentical() method, stateful session beans from the same home will always return true.

❑ C. The isIdentical() method can be used for only remote object references.

❑ D. Using the equals() method, stateless session beans from the same home are guaranteed to return true.

❑ E. Using the equals() method, stateful session beans from the same home are guaranteed to return true.

5 Which statement(s) about session beans are true? (Choose all that apply.)

❑ A. The bean provider must write the method public void remove() in both stateless and stateful session classes.

❑ B. Local clients can remove session beans by calling a method on the bean's home.

❑ C. The remove() method in the component interface can be used only by remote clients.

❑ D. To ask the EJBHome to remove a session bean, the client must provide the bean's handle.

COFFEE CRAM

Mock Exam Answers

1 Which capabilities are provided by both remote and local home interfaces for session beans? (Choose all that apply.)

[✓] A. Creating a session object.

❏ B. Removing a session object. – session beans can't be removed through a local home interface because there is only a remove() that takes a primary key

❏ C. Getting a session object's EJBMetaData. – not in a local home

❏ D. Getting a session object's handle. – not in a local home

2 When locating a session bean's remote home interface, which are steps that must occur to create a valid home interface reference? (Choose all that apply.) (spec: 57)

❏ A. The session context must be narrowed, and the narrowed result cast.

[✓] B. The result of the JNDI lookup must be narrowed, and the narrowed result cast.

❏ C. The initial context must be narrowed, and the narrowed result cast.

❏ D. The result of the JNDI lookup must be cast to an initial context, and then narrowed.

3 Given a remote client 'R', that has valid references to session beans 'A' and 'B', and given that A is a local client to B, which statements are true? (Choose all that apply.)

❏ A. R cannot pass his reference for A, to B.

[✓] B. A cannot pass his reference for B, to R.

❏ C. A cannot invoke methods on B.

[✓] D. B cannot invoke methods on R.

You can't give a remote client a local reference, A sees B through a local reference

4 When comparing two session objects, what is true? (Choose all that apply.) (spec: 65–66)

- [✓] A. Using the isIdentical() method, stateless session beans from the same home will always return true.
- ❑ B. Using the isIdentical() method, stateful session beans from the same home will always return true.
- ❑ C. The isIdentical() method can be used for only remote object references.
- ❑ D. Using the equals() method, stateless session beans from the same home are guaranteed to return true. The behavior of equals() is not specified
- ❑ E. Using the equals() method, stateful session beans from the same home are guaranteed to return true.

5 Which statement(s) about session beans are true? (Choose all that apply.)

- ❑ A. The bean provider must write the method public void remove() in both stateless and stateful session classes. – it's ejbRemove(), not remove()
- ❑ B. Local clients can remove session beans by calling a method on the bean's home. – local homes have only a remove() that takes a primary key
- ❑ C. The remove() method in the component interface can be used only by remote clients. – no, only for local
- [✓] D. To ask the EJBHome to remove a session bean, the client must provide the bean's handle.

– but not true for EJBLocalHome