# Table of Contents




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

# 2 architectural overview

# EJB Architecture


**EJB is about infrastructure.** Your components are the building blocks. With EJB, you can build big applications. The kind of applications that could run everything from the Victoria's Secret back-end to document-handling systems at CERN. But an architecture with this much flexibility, power, and scalability isn't simple. It all begins with a distributed programming model, where clients, servers, and even different pieces of the same application are running who-knows-where on the network. But how does the client *find* a bean? How does the client call methods it? Why are there different bean types? Will Ben marry J-Lo?

this is a new chapter 61


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

***no** objectives*


## *Background*

We're in this chapter for background, not because of an exam objective. Although, you could say that every objective in the exam *depends* on your understanding what's in *this* chapter.

But don't worry, we'll have plenty of objectives beginning with Chapter 3. By Chapter 6, you will look back longingly on this chapter and remember what it was like not to have any objectives. You'll miss this chapter when it's gone, so savor the moments you have with it.






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

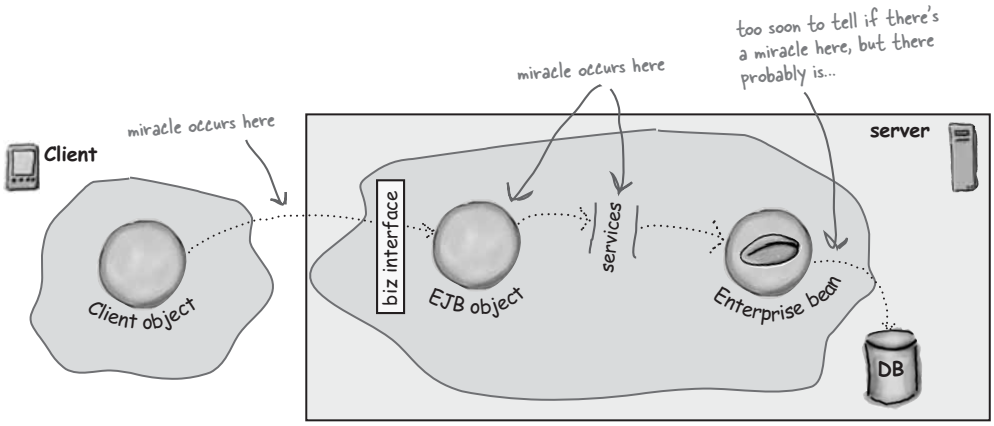
## You remember this picture...

But it was too high-level to get us anywhere. Think about how much is missing from this picture. Like, how does the client get a reference to something running on a different machine? How does the client actually communicate with the bean? How is it that the server can step into the middle of a client-to-bean method call?

Beneath EJB, there's Java's distributed technology for Remote Method Invocation (RMI). Although EJB hides some of the complexities of RMI from the bean developer, it's still there, and unless you truly understand it, some pieces will never make sense.

So, we start our descent from a high-level view to the blood and guts of EJB with a lesson on RMI. If you're one of the fortunate who've already worked a lot with RMI, you can skip this and go straight to the part where we talk about the ways in which EJB uses RMI. But you should still at least *skim* it, even if you're an experienced EJB developer, if for no other reason than to get comfortable with the terminology and pictures we'll use throughout the rest of the book.

OK, back to where we started—what's missing from this picture? Start by looking at the places where a miracle occurs...




A ridiculously high-level view of EJB architecture




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

## Making a remote method call

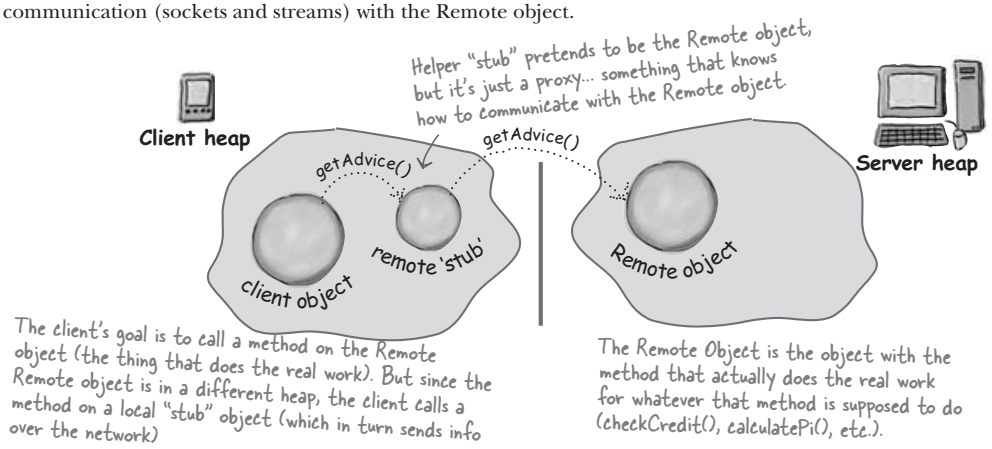
When you write a client to access a bean, the client is either *local* or *remote*. A local client means the client is running in the same JVM as the bean. In other words, both the bean and the client live in the same heap. We'll talk about that much more in the Client View chapter, but for now, remember that *local* means *same heap/ JVM*. Chances are, you'll use local clients only with entity beans, and only under very special circumstances.

You'll use a *remote* client when you want a bean to be used by the outside world. Most enterprise applications have a remote client, even if some of the beans *used* in the application talk to one another as local clients. (We'll explore every gory detail about this before the book is over.)

So how does an object in one heap/JVM directly call a method on a reference to an object running in another heap/JVM? Technically, it's not possible! Java references hold bits that don't mean anything outside the currently running JVM. If you're an object and you have a reference to another object, *that object must be in the same heap with you.*

Java RMI (Remote Method Invocation) solves this problem by giving the client a proxy (called a *stub*) object that acts as the go-between for the client and Remote object. The client calls a method on the *stub*, and the *stub* takes care of the low-level communication (sockets and streams) with the Remote object.

With RMI, your client object gets to act like it's making a remote method call. But what it's really doing is calling a method on a "proxy" object running in the same heap with the client. The proxy is called a "stub", and it handles all the low-level networking sockets and streams.







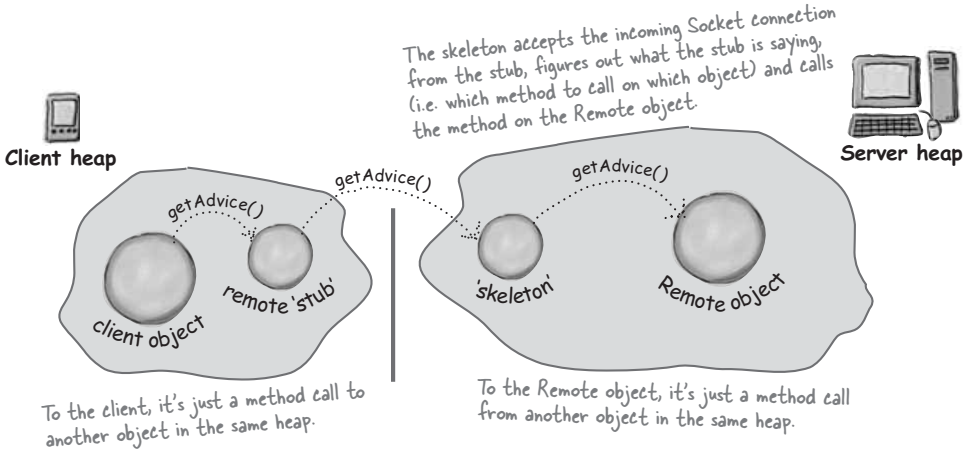
Head First EJB™ By Bert Bates, Kathy Sierra ISBN: 0596005717 Publisher: O'Reilly
Print Publication Date: 2003/10/01

## There's a "helper" on the server as well...

The Remote object has the method the client wants to call. But when the stub makes a network connection to the server, *something* on the server has to take the information in the incoming stream and turn it into a method call on the Remote object. You *could* put the networking code into your Remote object, but that defeats the whole point of RMI—to make it as easy for your client to call a method on an object across the network as it is to call a method on an object in the same heap. The goal of RMI is to promote *network transparency*. In other words, the fact that the objects are in different machines should be nearly transparent to the developer. Which means to you... less code—simpler code.

So, with that as the goal, RMI takes care of the server-side of the method call as well. The thing on the server-side that accepts the socket connection is called a *skeleton*. It's the counterpart to the client stub. In the early versions of RMI, for every stub there was a matching skeleton object. Today, though, that's not always true. The *functionality* of the skeleton has to happen somehow on the server-side, but an actual skeleton *object* is optional. We won't go into any of the details because it doesn't make any difference to us with EJB. How the container chooses to implement its skeleton behavior is up to the vendor.

All we care about is that *something* is on the server that the stub knows how to talk to, and that *something* knows how to interpret the message from the stub and invoke a method on the Remote object.







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

## there are no Dumb Questions

**Q: How can you have "network transparency"? What happens if the network or the server is down when the client calls the remote method? It seems like there's a LOT more that can go wrong than if the client object is just making a plain old method call to another object in the heap.**

A: Yes, yes. You obviously understand that "network transparency" is not only a myth, it's a bad idea. Of course the remote method call can fail in ways a local method call would not, and the client needs to be prepared for that!

That's why, in Java RMI, *all* remote methods must declare a java.rmi.RemoteException, which is a checked exception. That means the client has to handle or declare the exception. In other words, the client can't *really* pretend the method call isn't remote.

But wait, there's more—the client has to do something special to even *get* the reference to the Remote object in the first place. And what exactly *is* that reference? It's really a reference to the Remote object's *proxy*—the stub.

So no, RMI does *not* give you true network transparency. The designers of RMI want the client to acknowledge that *things can go horribly wrong with a remote method invocation*.

Still, when you look at everything that needs to happen to make a remote method call (networking Socket connection, streams, packaging up arguments, etc.), the client has to do only a couple of things: use a special lookup process to get the reference to the remote object, and wrap remote method calls in a try/catch. That's pretty trivial when you consider what it would take if the client had to manage the whole process.

(And there's even a way to make it easier for the client, using an EJB design pattern we'll see in the last chapter).

**Q: Am I responsible for building the stub and the skeleton? How does the stub *know* what methods my Remote object has? For that matter, how does the *client* know what methods my Remote object has?**

A: No, you don't need to make the stubs and skeletons. With plain old RMI, you use the RMI compiler (rmic) to generate them. But for the other two questions... we'll let you think about it for a minute before we look at the details.


In Java, what's the best way to tell the client what methods she can call? In other words, how do you expose your public methods to others?

Think about the relationship between the stub and the actual Remote object. What must they both have in common?

(We'll see the answer several pages from now)




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

## What about arguments and return values?

Remote method calls are just like local method calls, except for the RemoteExceptions. And what good would a method call be if you couldn't pass arguments or get a return value? You might as well be doing RPC*, the way your parents did.

This brings us to one of the key jobs for the stub and the skeleton (or whatever is doing the skeletonish things)—packing and unpacking values shipped over the wire.

Remember, the client is really calling a method on the stub, and the stub is local to the client (i.e. in the same heap). So, from the *client's* perspective, there's nothing special about sending arguments with the method. It's the *stub* that does all the dirty work. The stub has to package up the arguments (through a process known as marshalling) and send them in the output stream, through the Socket connection with the server.

The skeleton-thing on the server has to process the stream from the stub, unpack the arguments, figure out what to do with everything (for instance, which method to call on which object), and then invoke the method (a *local* call) on the Remote object, with the arguments.

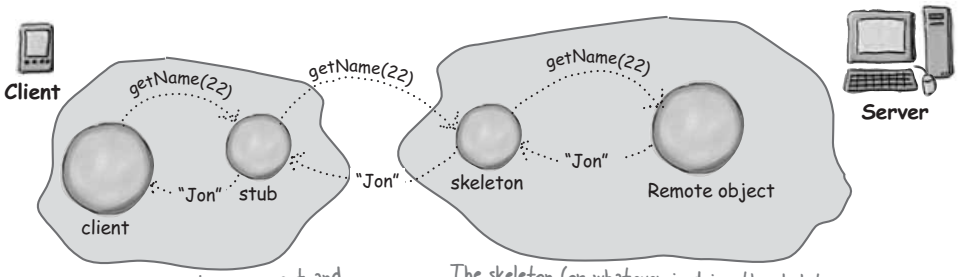
Then it all happens in reverse! The skeleton packages up the return values and ships them to the stub, who unpacks them and gives them to the client as plain old garden-variety return values. But in order to send arguments and return values, they must be primitives, Serializable objects, an array or collection of primitives or Serializable objects, or a Remote object.

**The stub and skeleton are in it for the whole round trip. They're both responsible for packing and unpacking the values shipped over the wire.**

**But it won't work if the arguments and return values aren't *shippable.***

***Shippable* values must be one of these:**

- **Primitives**
- **Serializable objects**
- **An array or collection of primitives or Serializable objects**
- **A Remote object**




The stub packages up the argument and sends it with the method call information, then unpacks the return value for the client.

The skeleton (or whatever is doing the skeleton behavior on the server) unpacks the method arguments for the Remote object method call, then packs and ships the return value.

*RPC stands for Remote Procedure Call. Boring.

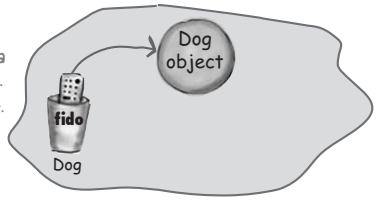



Head First EJB™ By Bert Bates, Kathy Sierra ISBN: 0596005717 Publisher: O'Reilly
Print Publication Date: 2003/10/01


**In ordinary local method calls, Java passes an object reference by value. In other words, by copying the bits in the reference variable.**

**The object itself is never passed.**

We'll think of a local Java reference as a remote control to an object on the heap. Something we can use to push buttons (i.e. call methods) on the object.

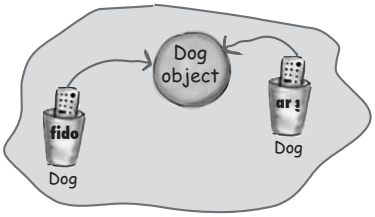



```
void go() {
  Dog fido = new Dog();
  this.trainPet(fido);
}
```

What are we really passing here??

A copy of the reference (the remote control), not the Dog object

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



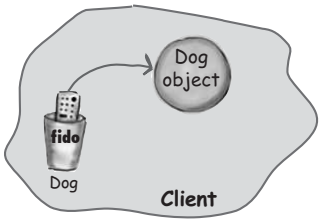

Now the "arg" parameter and the "fido" variable are identical copies. Both reference the same Dog object.

We know you know all this, obviously, but just so 'we're all on the same page' (which is an odd thing to say, since we clearly ARE all on the same page) but you get the idea.




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

## What really gets passed when you pass an object to a remote method?




**Imagine this CLIENT code:**

```
try {
  Dog fido = new Dog();
  remoteStub.trainPet(fido);
} catch(RemoteException ex){
    ex.printStackTrace();
}
```

NOW what are we really passing here??

Dog object

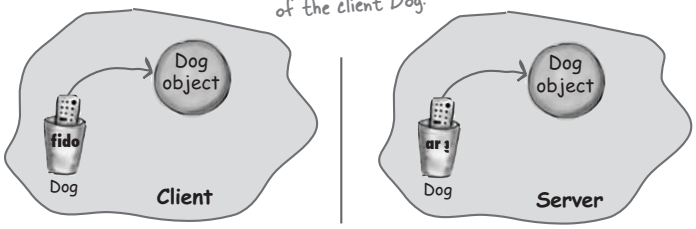
NOT the reference value!

It's a serialized copy of the actual Dog object.

**And this REMOTE method:**

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

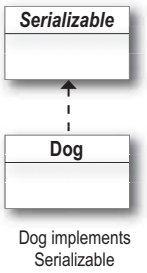
The server Dog is a copy of the client Dog.




**If your remote method has an argument that's an object type, the argument is passed as a full copy of the object itself!**

**For remote calls, Java passes objects by *object* copy, not *reference* copy.**

**A serialized copy of the object is shipped to the Remote object.**



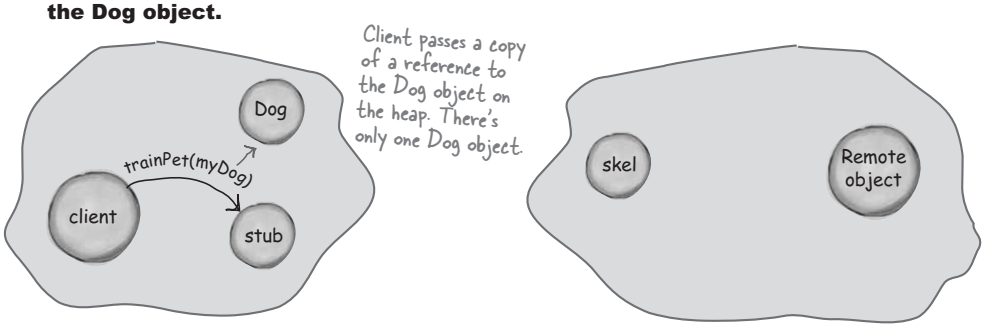




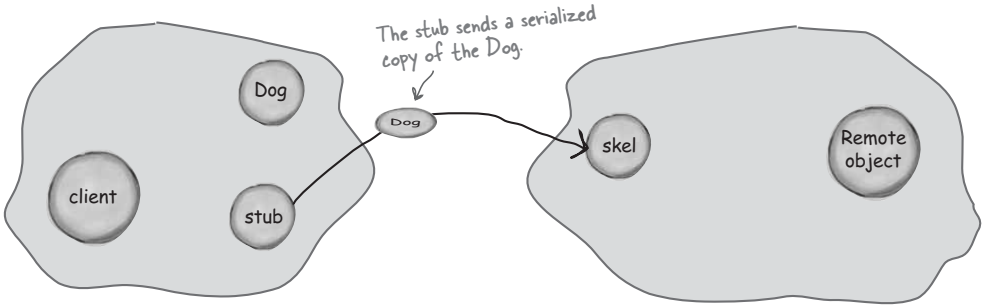
Head First EJB™ By Bert Bates, Kathy Sierra ISBN: 0596005717 Publisher: O'Reilly
Print Publication Date: 2003/10/01

## Getting the object argument from the client to the server

**1 Client invokes trainPet(myDog) on the stub, passing a copy of the reference to the Dog object.**




**2 The stub makes a serialized copy of the object and sends that copy over the wire to the skeleton.**



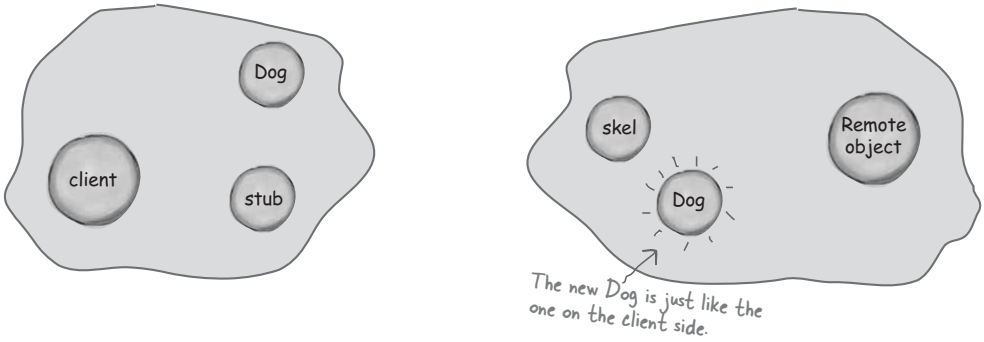




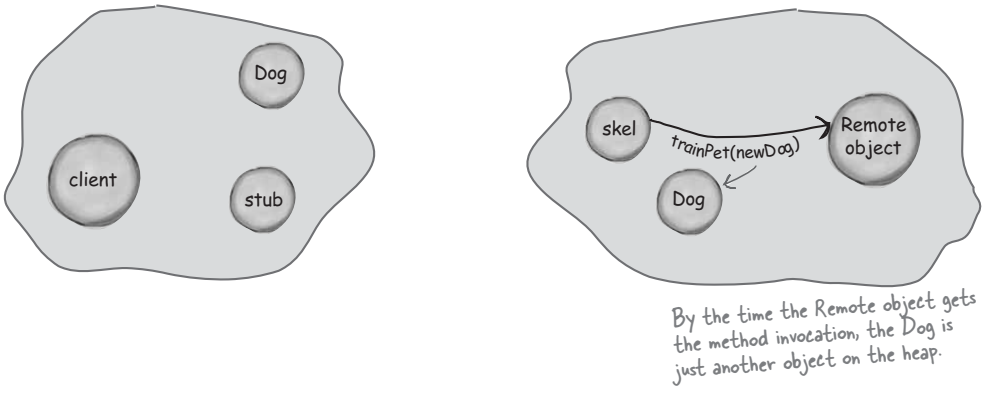
Head First EJB™ By Bert Bates, Kathy Sierra ISBN: 0596005717 Publisher: O'Reilly
Print Publication Date: 2003/10/01

## Unpacking (deserializing) the object on the server

**3 The skeleton deserializes the passed argument, creating a new Dog object in the Remote object's heap.**




**4 The skeleton invokes the method on the Remote object, passing a plain old Java reference to the new Dog object.**







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

## there are no Dumb Questions

**Q: I'm sitting here with a HashMap full of Serializable Customer objects with String keys. Do I have to worry about whether the HashMap *itself* is Serializable?**

A: OK, this is kind of a tricky one. All of the Collection implementations in the J2SE API are Serializable. So you don't have to worry about a Hashmap—as long as what you put *in* the Hashmap is Serializable, you're fine.

But... there *is* one place where things can fail. Chances are, you'll never see this, but it's still worth mentioning. You probably already know that the Map classes like HashMap and Hashtable have a values() method that returns a collection of just the values, without the keys. In other words, if you called it on your HashMap, you'd get a Collection of Customer objects.

But what *type* of collection? And there's the problem. You don't know. All you know is that it's something that implements the Collection interface. But that isn't enough to tell you whether the Collection returned by values() is Serializable! In other words, you might get back a Collection that—even when filled with Serializable objects—is not itself Serializable!

The bottom line: don't rely on the Collection returned from a map's values() method to be Serializable. Put your values into something you can trust to be Serializable, like ArrayList, before trying to ship them as part of a remote method call.

**Q: If something you're passing to a remote method isn't Serializable, is this a compile-time or runtime failure?**

A: Runtime! Usually, anyway. Remember that the *declared* argument or return type is not necessarily the same as the *actual runtime type*.

The only case where it can fail at *compile* time is if the remote method actually uses **Serializable** as the *declared* type of the argument or return value:

```
public void takeIt(Serializable s);
```

In that circumstance, the compiler can use normal Java type-checking to see if the declared type of the thing being passed implements Serializable.

Most of the time, the declared argument or return type is something *other* than Serializable (like Dog, ArrayList, String, etc.), so Java won't know whether the runtime object is Serializable until it actually tries to do the serialization (and then it throws an exception).

And with collections and arrays, if *any* of the objects inside aren't Serializable, the whole serialization fails!

**Q: Does my class have to explicitly implement Serializable, or can I inherit Serializableness from my superclass?**

A: Remember Java's IS-A rule: if your *parent* (superclass) is something, then so are *you*. If Dog extends Animal, and Animal implements Serializable, then Dog is Serializable whether the Dog class explicitly declares it or not.

However, it *is* considered good practice to explicitly declare your class as Serializable even if your superclass does, just so that others looking at your class API don't have to hunt through your class' inheritance tree to see if *somebody* up there is Serializable.




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

**Remember, arguments and return values for a remote method must be one of these:**

- **Primitives**
- **Serializable objects**
- **An array or collection of primitives or Serializable objects**
- **A Remote object**





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

## Passing a Remote object through a remote method call

It does *not* make sense to send a Remote object in a remote method call. After all, the whole point of a Remote object is to stay remote. To be accessed by clients who live... somewhere else. In another heap.

But what if you want to pass a remote client a reference to another Remote object? What if, rather than handing your client a full-blown copy of a Customer, you send him a stub to a Remote Customer?

Think about it. *Before* you read the next page.


What are the implications of passing a stub to a Remote Customer object as opposed to passing a non-remote Customer object?

What are the benefits of passing a stub instead of the real Customer object?

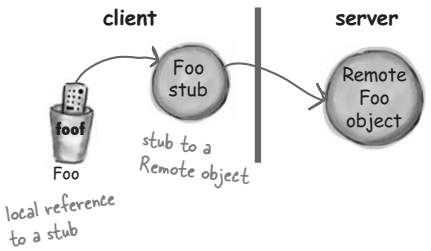
What are the drawbacks?

(Note: whether to pass serialized objects or stubs to remote objects is a crucial design decision. We'll explore this when we look at performance and patterns.)

When you pass a Remote object to or from a remote method, Java actually sends the Remote object's stub.

In other words, at runtime, the Remote object stays right where it is, and its stub is sent over the wire instead.

A remote reference is a stub to a remote object. If the client has a remote reference, he has a local reference to a stub, and the stub can talk to the Remote object.







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

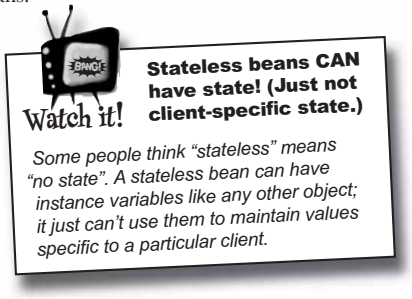
## Session beans can be stateless or stateful

We'll go over all this in detail in the Session Bean chapter. For now, you need to know that session beans can be marked (at deployment time) as either state*less* or state*ful*.

A stateful bean can remember conversational state between method calls, while a stateless bean won't remember anything about the client between method invocations.

The phrase "conversational state" really means "client-specific state", and a typical example is a shopping cart. It wouldn't be fun if you (the shopper) got a cart, put something in, but then when you go to put the second thing in, the first thing vanishes from the cart. Not too user-friendly. So, a good shopping cart will keep the client shopper state (i.e. the items in the cart) for as long as the shopping session is *alive*. (We'll explain what we mean by *alive* in the Session Bean chapter.)

Stateless beans simply forget about the client once the method call completes. So, stateless beans are for services that don't require a continued conversation between the client and the service. That doesn't mean the client won't keep calling methods on the stateless bean, but it does mean that the client can't depend on the bean remembering anything about the previous method calls.




there are no Dumb Questions

**Q: I've heard that only state*less* session beans are scalable, and that nobody should ever use state*ful* session beans. Is that true?**

A: No, not completely. It *is* true that stateless session beans are generally *more* scalable than stateful session beans because of the way stateless beans are managed by the container. You'll see the reasons for this in the Session Bean chapter.

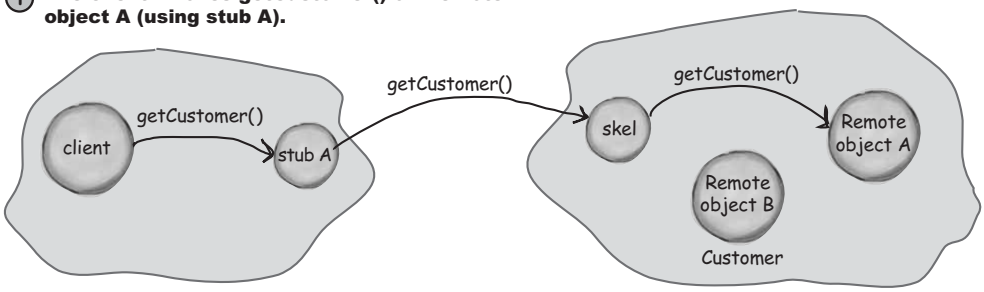
But... that doesn't mean you should never use stateful beans. You *should* consider stateful beans when you need conversational state, and when the alternatives for saving that state (like using the client to store state, or using a servlet to store state, or using a database to store state between each method call from the client) are more of a performance hit than the less-scalable nature of stateful session beans.




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

## When the return value is a Remote object...

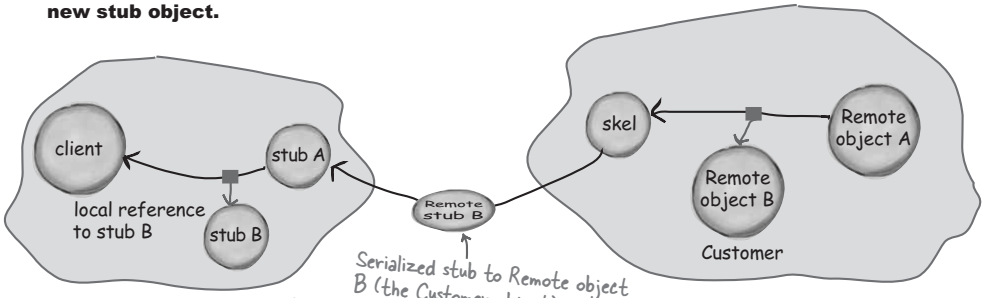
**1 The client invokes getCustomer() on Remote object A (using stub A).**




**2 Remote object A returns a reference to a Customer object (Remote object B). The skeleton substitutes (and serializes) the Remote object's stub, and sends it back to the client.**

**The Customer stub (B) is deserialized on the client, and the client gets a local reference to the new stub object.**

The Remote object (A) simply returns a local reference to the Customer object (Remote object B), but it's actually the stub that gets sent back.




Serialized stub to Remote object B (the Customer object).

Now the client gets back a local reference to the Customer stub object (B).




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

## there are no Dumb Questions

**Q: What happens if the client object and the Remote object are running in different JVMs, but on the same physical machine? In other words, they're both running in Java programs on the same server?**

A: Doesn't make any difference. All that matters is whether the two objects live in different heaps, and JVMs do *not* share heaps with one another, thank you very much, no matter *how* intimate they are (cohabiting the same server).

In fact, you can (and with EJB often *must)* use RMI even when the objects *are* in the same heap.

**Q: Why in the world would you ever want to do use RMI if you don't need to? Isn't there enough overhead with remote calls as it is?**

A: We'll go into this in more detail later, but the main reason is because if you don't use RMI for method calls, you're locking down your design in such a way that you can't distribute your objects in different places in your network (or even on the same server). In other words, without RMI, you must have both objects in the same heap.

For a distributed programming model, that's a pretty permanent decision, with no chance to change your mind later, without rewriting code. On the other hand, if you *do* use RMI, you can decide later to split your program up into different nodes on your system, with little or usually *no* code changes.

So the tradeoff is flexibility for performance, but for most distributed enterprise apps, the overhead of a remote call is not your biggest problem. It's usually your bandwidth and/or concurrency that hurts the most. Overall, you probably have bigger performance fish to fry in areas *other* than whether your calls are remote or local. But the story isn't always that simple, so we'll explore this again later in the book.

## BULLET POINTS

- EJB uses Java RMI (Remote Method Invocation) so that your beans can be accessed by remote clients.
- A remote client, in this context, is an object running in a different JVM, which also means a different heap.
- A Remote object stays in its own heap, while clients invoke methods on the Remote object's proxy, called a *stub*.
- The stub object handles all the low-level networking details in communicating with the Remote object.
- When the client wants to call a method on a Remote object, the client calls the same method on the stub. The stub lives in the same heap as the client.
- To the client, a remote method call is identical to a local method call, except a remote method can throw a RemoteException (a checked exception).
- The stub packages up the method arguments and sends information about the call to a *skeleton* on the server. The skeleton object itself is optional, but the skeleton's work must be done by something on the server. We don't have to care who—or what—is actually doing the skeleton's work.
- Arguments and return values must be one of the following: a primitive, a Serializable object, an array or collection of primitives or Serializable objects, or a Remote object. If the value isn't one of these, you'll get a runtime exception.
- If an object is passed as an argument or return value, the object is sent as a serialized *copy*, then deserialized on the Remote object's local heap.
- If a Remote object is passed as an argument or return value, the object's stub is sent instead.




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





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

## What must the Remote object and the stub have in common?

How does the client know which methods to call?

How does the stub know which methods the Remote object has?

Remember, if the stub is pretending to be the Remote object, the stub must have the same methods as the Remote object.

Of course you know the answer to this.

An *interface*.

The way all methods in a distributed environment should be exposed to a client.

We call this the ***business interface*** because it has the business method(s) the client wants to call. Technically, the business interface for a Remote object must be, surprisingly, a Remote interface.

It all begins with a Remote **interface**. Both the Remote object and the stub implement the same interface... the one with the methods the client wants to call.

A Remote interface must extend **java.rmi.Remote** and every method must declare a **RemoteException**.

To be Remote, an interface must follow three rules:

- **it must extend java.rmi.Remote**
- **each method must declare a java.rmi.RemoteException**
- **arguments and return types must be *shippable* (Serializable, primitive, etc.)**

RemoteException and Remote interface are in java.rmi package

A Remote interface MUST extend java.rmi.Remote (which doesn't have any methods).

All of your methods must declare a RemoteException.

```
import java.rmi.*;

public interface DiceRoller extends Remote {

    public int rollDice() throws RemoteException;
}
```

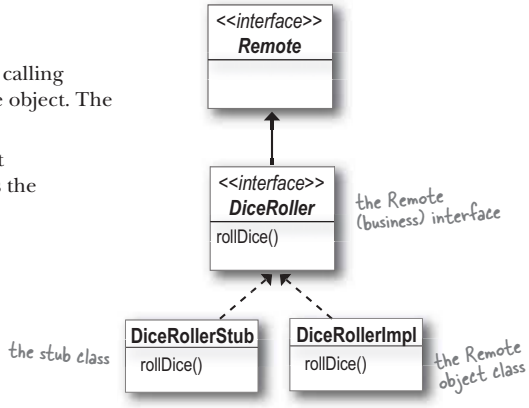


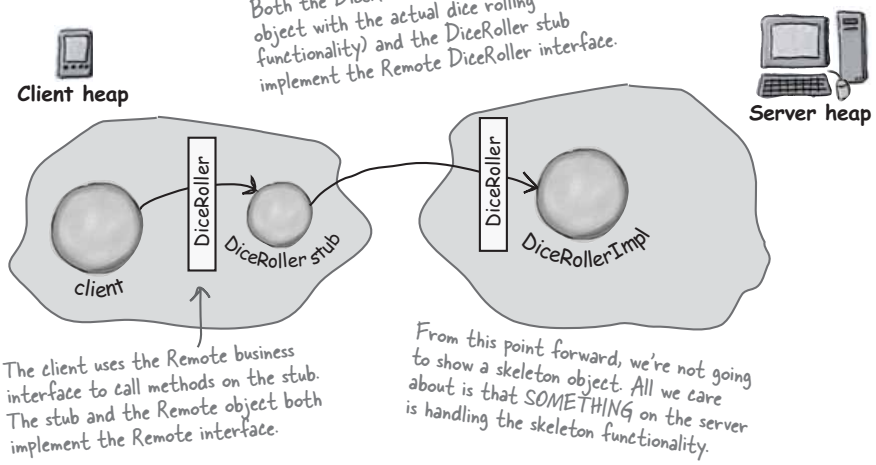

Head First EJB™ By Bert Bates, Kathy Sierra ISBN: 0596005717 Publisher: O'Reilly
Print Publication Date: 2003/10/01

## The client calls business methods on the stub through the Remote business interface

Remember, as far as the client's concerned, he's calling methods on The Real Thing. The actual Remote object. The thing that has the methods he wants to call.

The only thing reminding the client that he isn't calling methods directly on the Remote object is the RemoteExceptions he has to deal with.





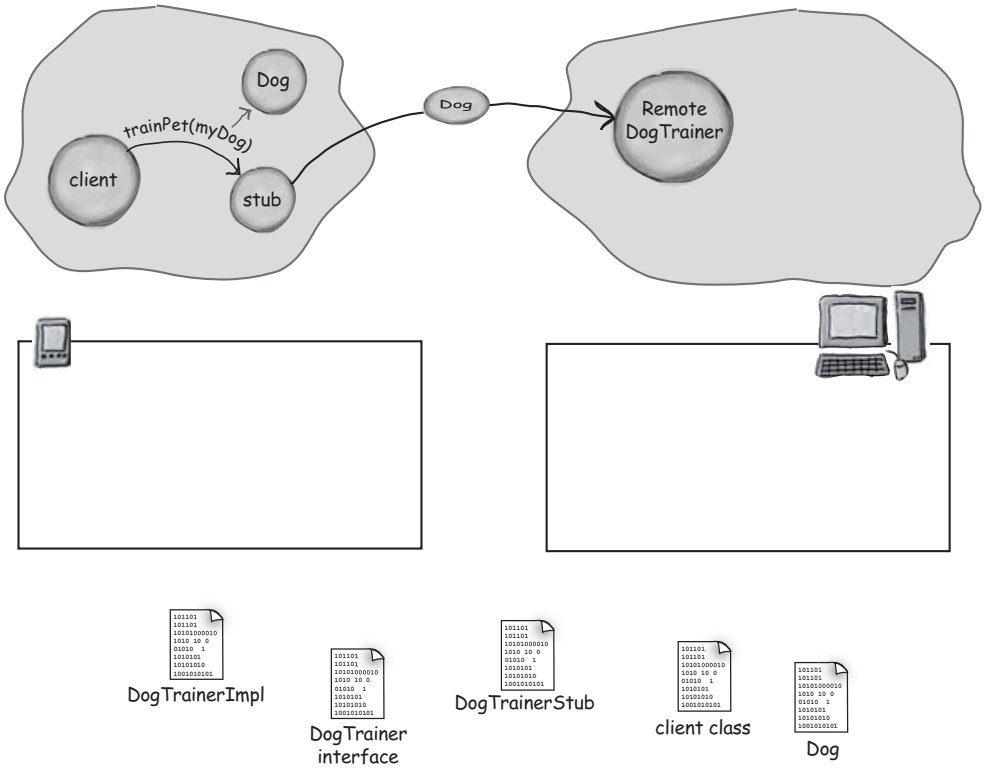


Head First EJB™ By Bert Bates, Kathy Sierra ISBN: 0596005717 Publisher: O'Reilly
Print Publication Date: 2003/10/01

## Sharpen your pencil

This sharpen gets you on the one most common mistake EJB developers make. So don't skip it!

Based on this scenario, draw the classes below into the appropriate slot for whether they must be on the client, server, or both (you can reuse a class). The picture is simplified, so you aren't seeing all of the players involved.







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

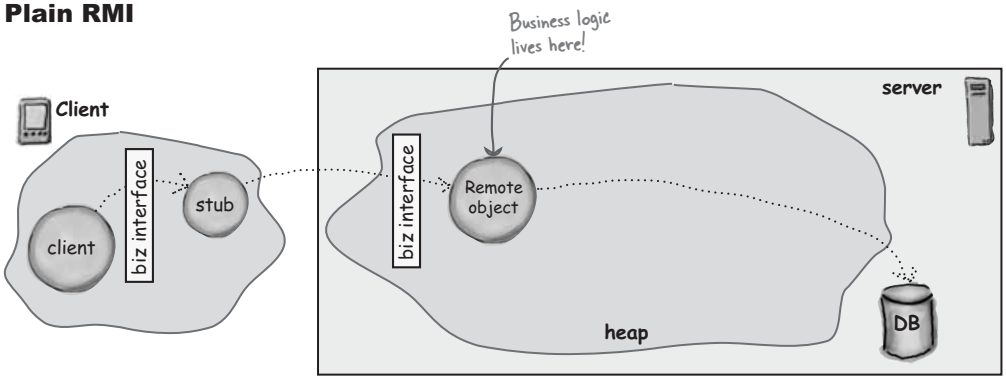
*architectural* overview

# How EJB uses RMI

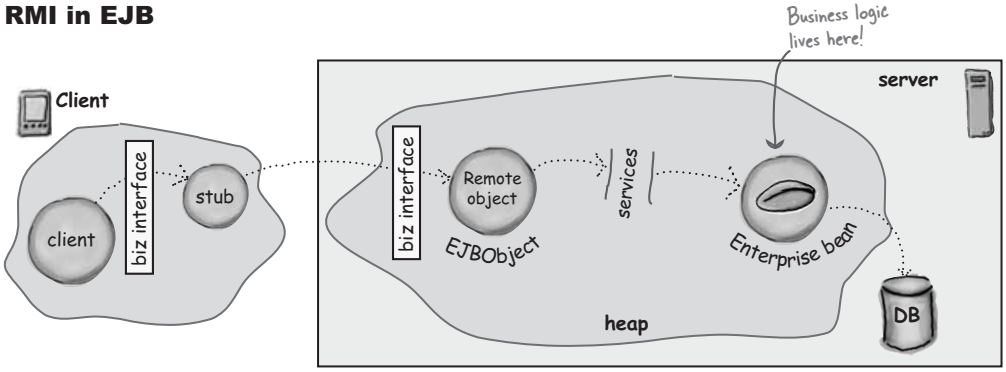
In EJB, a client's entry point into the enterprise application is nearly always through a reference (stub) to a Remote object. Yes, it is possible, and sometimes necessary, to use a *local* client (i.e. a client in the same heap as the bean, and which doesn't use RMI to invoke business methods), but this is for only a few very special cases.

So RMI lies at the heart of most client-to-bean communication. But as you saw a hint of in the first chapter, the EJB architecture is a little more complex than a simple client-to-stub-to-Remote-object scenario. In EJB, the bean—the thing on which the client wants to call business methods—is not Remote!

**Plain RMI**




**RMI in EJB**







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

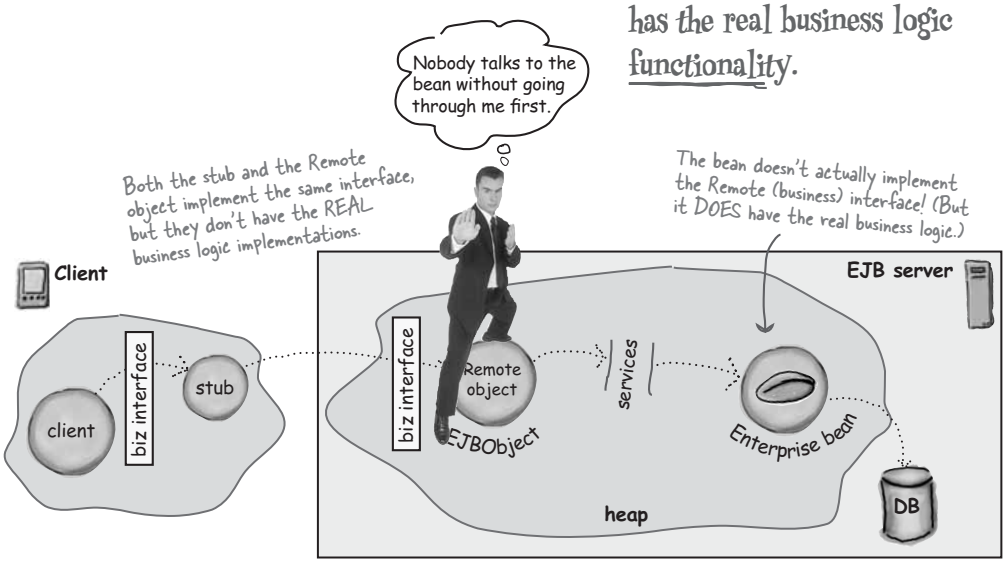
## The Remote object is not the *bean*, it's the bean's *bodyguard*—the EJBObject

In EJB, remember, the Remote object (EJBObject) is the bean's bodyguard. The bean sits back, protected from all client invocations, while the EJBObject implements the Remote interface and takes the remote calls. Once the call gets to the EJBObject, the server jumps in with all the services, like security (is this client authorized to call this method?), transactions (is this call part of an existing transaction, or should we start another transaction?), and persistence (does the bean need to load info from the database *before* running the client's method?).

The EJBObject implements the Remote business interface, so the remote calls from the client come to the EJBObject. But it's still the bean that has the real business logic, even though the bean doesn't implement the Remote interface in the technical Java way.

Both the Remote object and the stub implement the same interface–the business interface (called a Component interface)–but without the real business logic behavior.

The bean class does NOT implement the business interface (in the formal Java way), but the bean has the real business logic functionality.







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

# The Component interface

In EJB, the business interface is called the *component* interface. This is where you expose your business methods to the client. The main difference between an RMI interface and a remote component interface is that with EJB you extend javax.ejb.EJBObject instead of java.rmi.Remote.

### Key points:

1. Any interface with java.rmi.Remote in its inheritance tree is a Remote interface.

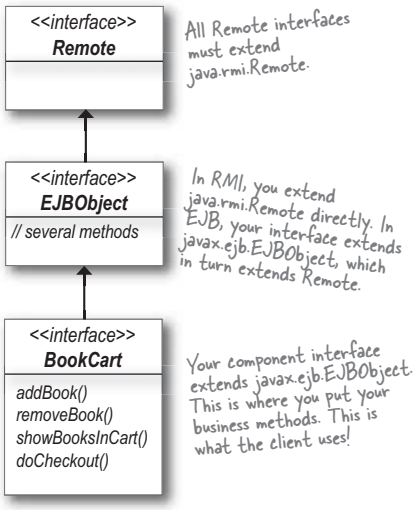
2. The EJBObject interface extends Remote, so EJBObject is a Remote interface.

3. Your remote component interface must extend the EJBObject interface.

   (You *can* have a local component interface, and the rules are different, but we'll look at that in the chapter on Client View.)

4. You expose your business methods to the client through the component interface.

5. The EJBObject interface adds additional methods for the client to use. (We'll see those later.)

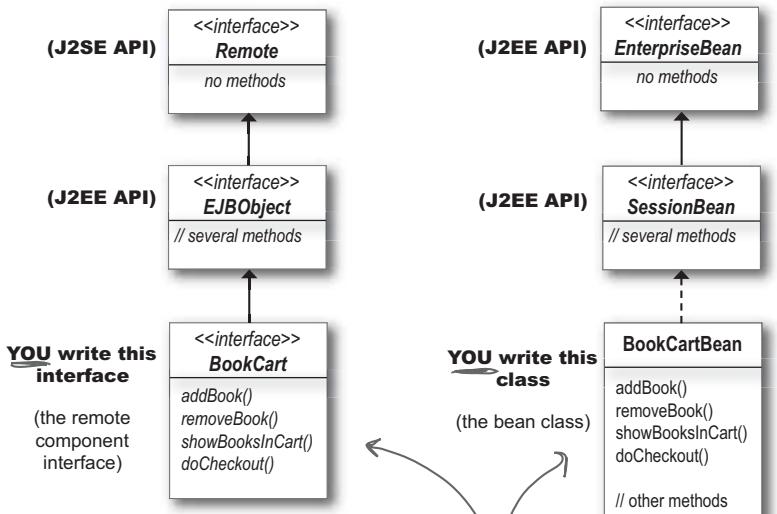



***Whoever implements the BookCart interface must implement* all *the methods from both BookCart and EJBObject. The EJBObject interface adds the methods that all EJB clients might need.***




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

## How the bean class fits in




Same methods, but no direct Java connection. BookCartBean does not implement the BookCart interface!

(but it sure looks like it does...)

The bean developer must make sure that the bean class has methods that match the methods in the component interface, exactly as if the bean class DID implement the component interface.


**You REALLY have to know the interfaces!**

*The exam expects you to know exactly where each method lives. You need to know which methods are in EJBObject, and which are in SessionBean, for example. And you have to know the exact signature for those methods. We'll get into these interfaces in the next couple of chapters, and when we do, pay attention!*




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

## there are no Dumb Questions

**Q: I just want to be sure I'm clear about this... interfaces can EXTEND other interfaces?**

A: Yes, interfaces have their own inheritance tree. In fact, with interfaces, you can do something you can't ever do with a class—***an interface can extend more than one interface!***

```
interface Cart extends EJBObject,CartBusiness
```

**Q: But what does that really mean when an interface extends another interface? Extending means inheritance, but what is the interface inheriting?**

A: When one interface extends another, it inherits everything from that interface. Whoever implements an interface must implement not just the methods from that interface, but also the methods from every interface that interface extends... all the way up the interface inheritance tree.

So, in this example, whoever implements BookCart must also implement the methods of EJBObject.

**When you implement an interface, you must implement *all* the methods that interface inherits from its super-interfaces.**

**So whoever implements BookCart must implement the methods from both BookCart *and* EJBObject.**

**Q: Why doesn't the bean implement the Remote/business interface? Isn't the whole point of an interface implementation to use the compiler to keep you honest, and to support Java type-checking?**

A: You asked this question before. But, hey, we all forget things, so we'll remind you again. The bean doesn't implement the Remote interface because the bean is never supposed to be a Remote object (in the Java RMI sense). In other words, you never ever want anyone to have a stub to the actual bean! If you were to somehow sneak a remote reference (i.e. a stub) out to the world, you'd be defeating the whole purpose of EJB! If you let a client talk directly to the bean, then the server wouldn't be able to apply its services, and if you don't need the services... you see where we're going here.

Technically, it *is* legal to have the bean implement the Remote interface, but it's a really bad idea, since you could make mistakes that wouldn't be caught at compile time (but which would blow up later). But you don't need to do it, since virtually all bean development tools (including nearly every bean-aware IDE) understand the relationship between the bean, the EJBObject, and the Remote interface, and they guarantee that the component interface and the bean class have matching methods.

**Q: But what if I just find this too disturbing? This whole idea violates my Java sensibilities, the very principles upon which I code. Surely there must be something I can do?**

A: You could trust us on the whole "you're almost certainly using tools, so it really won't matter. Really" thing, but no... so if you insist, yes there is something you can do that'll probably help you feel better. It's on the next page, but you can probably figure it out yourself anyway.

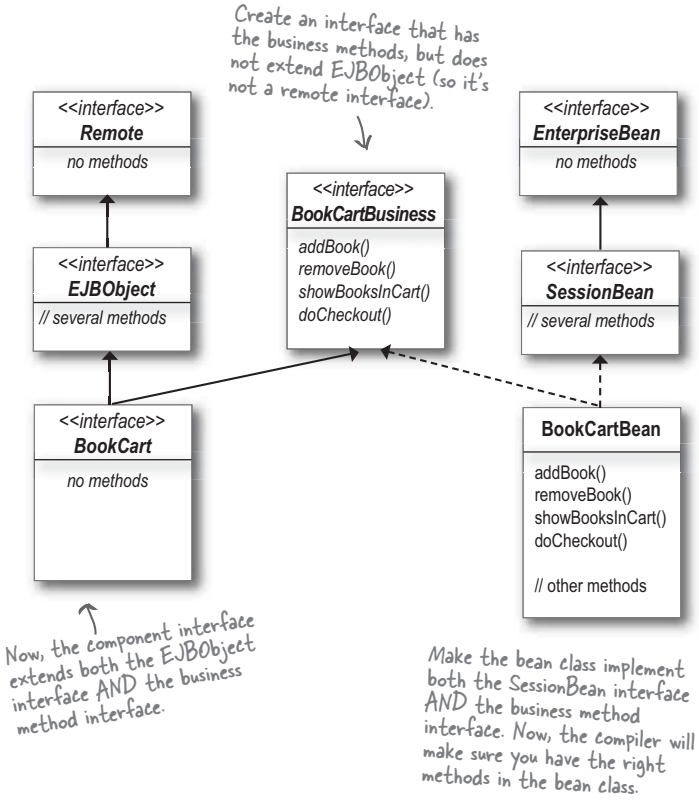
Nonetheless, we're sticking by our story that most developers won't need to do this.




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

***interface** implementation*

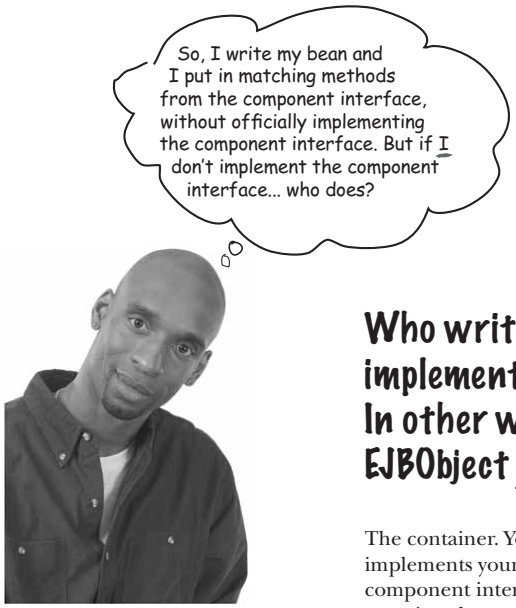

## A design for those who feel squeamish that the bean doesn't implement the business method interface...







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




## Who writes the class that really DOES implement the component interface? In other words, who makes the EJBObject class?

The container. You declare the methods, but the container implements your component interface. Remember, your component interface is the one that extends EJBObject, so the container has to implement not just your business methods, but also the methods of EJBObject (which we haven't yet looked at).

**Q: But how does the container know what to put in those methods? I'm the one who declared those methods...**

A: Remember, the container isn't implementing the real business logic. The true functionality for those business methods lives in your bean class—the class that you implement. The class implementing the component interface is going to be the EJBObject. The bodyguard. The Remote object. And remember that the EJBObject is only pretending to be the bean. It can respond to the remote method calls coming from the client (via the stub), but the EJBObject's only job is to capture the incoming client calls to the bean. After that, it's up to the container/server to take over.

We don't really know how the EJBObject is implemented—it's completely up to the vendor. But we don't really care. All you need to know is that an EJB container is required by the spec to generate the code for the EJBObject (and its corresponding stub).




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

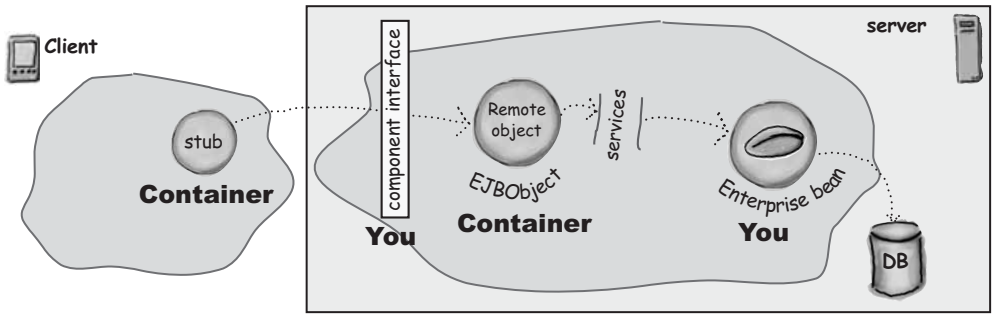
*who creates* which class

## Who creates what?

For a bean with a remote client view (in other words, a bean that can be accessed by remote clients), you know that you have to write the Component interface and the Bean class. But *somebody* has to write the class that implements your Component interface (to make the Remote EJBObject), and *somebody* has to make the stub that goes with that EJBObject. That *somebody* is the Container. And, though we haven't yet talked about the Home, we've listed the relevant pieces here for completeness.

| You | the Container |
|---|---|
| 1 **The Component interface** (extends javax.ejb.EJBObject) | 1 **The EJBObject class** (implements your Component interface) |
| 2 **The Bean class** (implements javax.ejb.SessionBean or javax.ejb.EntityBean) | 2 **The EJBObject stub class** (implements your Component interface and knows how to talk to the EJBObject) |
| 3 **The Home interface** (extends javax.ejb.EJBHome; we'll talk about this on the next page) | 3 **The Home class** (implements your Home interface) |
| | 4 **The Home stub class** (implements your Home interface and knows how to talk to the Home) |

We haven't talked about the Home yet, so we don't show it on this picture.







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


**EJBObject:** Hey Beanie...don't you ever get tired of always having me in the middle of everything? Don't you ever just want to have a direct conversation with someone?

**Bean:** No. I'm too important. I'm too valuable. And I'm sure as hell not gonna start screening my own calls. That's what you people are for.

**EJBObject:** You people?

**Bean:** Yeah, you people who work for the container. You, the Home, the stubs, all of you. My job is to handle the complex business logic. The critical functions that mean the difference between success and failure in an enterprise environment.

**EJBObject:** (Geez, sounds like a marketing speech). OK, so you have some important methods, but I still don't see why you need to have me in every call.

**Bean:** Look, my work is too important to be interrupted by clients who have no business calling me in the first place. Do you honestly think that I am going to check security clearances for every caller? Like I don't have better things to do?

**EJBObject:** OK, so it's really just a security thing, but I don't see why *you* can't just have the code to check the security access of the caller. That would save a lot of overhead (namely, ME).

**Bean:** First of all, security is just ONE of the reasons you have to take my calls. I'll get to those other reasons in a minute. But as for putting in code to do my own checks, actually I CAN do that, if the programmer wants me to. But it's usually not the way to handle security.

**EJBObject:** What's wrong with you handling the security checks in your own code?

**Bean:** You really don't know, do you? [rolls eyes] First, if the security checks are coded into me, then I'm not as reusable. The whole point of EJB is to configure and customize beans at runtime, without rewriting the code. If I've got security programming in my Java code, then it can't be changed without touching the source. And who wants that?

**EJBObject:** I guess that makes sense. You put the security checks in the XML deployment descriptor, and then when I get the call, the server can check to see if the client has the right authorization. But what if security isn't even an issue for you? What if whoever deploys you doesn't care who calls the methods?

**Bean:** Not too bright, are you? But at least you can lift heavy objects... THINK about all the other things that matter, like transactions and persistence.

**EJBObject:** Oh, I forgot about transactions. OK, that makes sense too. The server has to figure out if there's a transaction context before it calls your method so that your method can run in a transaction. Either your own, or the caller's...

**Bean:** Duh.

**EJBObject:** But where does persistence come in?

**Bean:** Well, think about entity beans for a minute. If I'm an entity bean,that means I'm representing some entity in the underlying persistent store and—

**EJBObject:** —wait—by persistent store, don't you just mean DATABASE? Why don't you just say *database*?

**Bean:** Uh, newsflash, the word phrase "persistent store" is not a synonym for "database". A database is just one example of a persistent store. But if it makes you feel better to think about it that way, go ahead for now. But as I was saying, if I represent an entity, say, a customer named Tom Duff, then what happens if the client calls my getAddress() method? The server can't just hand me the call!

**EJBObject:** Because...

**Bean:** Because I have to load in Tom Duff's information from the database first!

**EJBObject:** Because...

**Bean:** Because I'd return an address that might not even be valid! Unless I'm still in a previous transaction with this client, then the server has to tell me to load myself with Tom Duff's database info BEFORE the server tells me to run the getAddress() method. Otherwise, who knows what I'd return? OK, well, that's last call, so we'll have to continue this some other time.




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

## there are no Dumb Questions

**Q: How and when does the container create the EJBObject and the Home and the stubs?**

A: When you deploy a bean, the container looks at the DD and takes it from there. Remember, the DD gives the fully qualified name of your Remote Component (EJBObject) interface and your Remote Home interface. So, once the container gets those interfaces, it generates code for the two classes implementing those interfaces. And because they're Remote, the container also creates the client stubs that know how to communicate back to the Remote objects.

**Q: Are these always plain old RMI stubs? I see that when we deployed using the RI, it printed out a message that it was running rmic...**

A: The container can do whatever it wants to create the stubs; the requirement is that the stubs be RMI-IIOP compliant. In fact, a server can use something called dynamic proxies to implement the stub functionality, but we don't care. When we say "stub" we mean something with stub behavior. Whether it's an RMI stub or something else, is an implementation detail for the server. We'll look at that in more detail in the next chapter. The bottom line is that you really don't know what the stub class code looks like. For that matter, you really don't know how the EJBObject and the Home are implemented. You're not supposed to know.

You *might* have an EJB container that lets you view (possibly even *hack*) the generated source code, but don't count on it. And if we were you, we just wouldn't go there, even if we could.

**Q: So these classes are up to the vendor's implementation?**

A: Yes! The vendor has all sorts of choices for implementing these classes and might use the stubs and/or Home and EJBObjects to get some performance advantages. But again, that's not up for you to mess with, or even know about.

The only requirement in the spec that you're guaranteed (and required to adhere to) is that Remote objects must follow the rules for RMI-IIOP, which means Java's Remote Method Invocation using the IIOP (CORBA standard) wire protocol.

**Q: You brought it up: how is RMI-IIOP different from regular RMI?**

A: Plain old RMI uses JRMP as its wire protocol. But IIOP lets Remote objects interoperate through CORBA (we won't be saying much at all about CORBA in this book; it's definitely out of scope for the exam and the book, except in a few tiny cases we'll see scattered throughout the chapters).

That gives your objects a chance to be accessed, for example, by even non-Java clients. One thing IIOP specifies is the way information for transactions and security can be propagated along with the method call, and your container might be taking advantage of that.

For the most part, you'll barely notice the difference between plain old RMI and RMI-IIOP. But... there are a couple of places where it's different, and one of these differences is definitely on the exam—the need to narrow a stub. We'll cover narrowing in detail in the next chapter on Client View. For now, just know that it's something a client must do with an EJB stub that they don't have to do with a plain Java stub because the EJB spec tells you to assume that the stub is using IIOP and thus might be a different *kind* of stub.

**Q: When are we going to talk about the Home?**

A: Next page.

**Q: Why did you take so long? Isn't the Home important?**

A: As crucial as the Home is, it's usually just the way you *get* a reference to something that implements the Component interface. In other words, you use the Home to get an EJBObject stub (for Remote clients, which is all we've talked about so far).

For Entity beans, the Home can have a more important role, and we'll see that, but even with Entity beans, the Home's primary use is still to get EJBObject stubs. After that, most of the communication between the client and the bean comes through the EJBObject and not the Home. Most of the time, in fact, clients use the Home just to get the EJBObject reference, and then the Home reference is tossed out, not needed.




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

## The bean Home

Every Session and Entity bean has a Home.

Message-driven beans don't have homes because message-driven beans don't have a client view (in other words, client's can't get a reference to a Message-driven bean).

The Home has one main job: to hand out references to a bean's Component interface. For a Session bean, that's just about all you'll do with the bean's Home. For Entity beans, though, the Home plays a much bigger role.

Each deployed bean has its own Home, and that Home is responsible for all bean instances of that type. For example, if you deploy a ShoppingCart Session bean, the container will create one ShoppingCart bean Home. That ShoppingCart Home takes care of all the instances of ShoppingCart beans. In other words, if 2,000 clients each want their own ShoppingCart bean reference (which, remember, means a reference to the ShoppingCart bean's Component interface), the one and only ShoppingCart Home will hand out all 2,000 references.

If you deploy three beans as part of an application, say, a ShoppingCart, Customer, and Product, there will be three Homes in the server representing each of those deployed beans. It makes no difference how many EJBObjects and stubs the Home objects hand out, there will still be only three Homes.

So does that mean that for each Home there is only a single instance of the class that implements the Home interface for that bean type? Not necessarily, but that's exactly how we're supposed to think about it. We'll actually refer to the Home as *the Home object*, and we'll assume that there's always just one per deployed bean type. The spec guarantees that you can think about it that way, regardless of what your vendor actually does with its implementation for the Home.


Each deployed Session and Entity bean has a Home. For example, the AdviceBean has an AdviceBean Home. No matter how many clients get an AdviceBean, there's only one AdviceBean Home.

The Home's job is to hand out references to that bean's Component interface.

(Technically there's a little more to the story because a bean might have both a local Home and a Remote Home, but that's really unlikely. Even then, there would be only one of each Home type (Remote or local) no matter how many beans of that type are alive.)

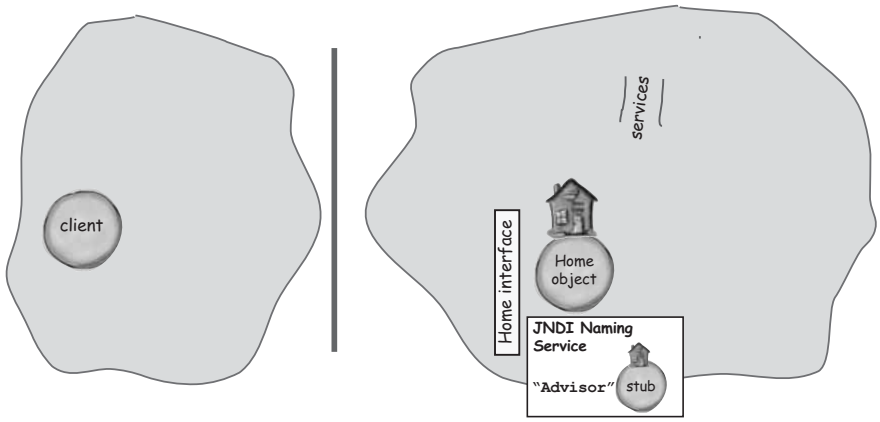



Head First EJB™ By Bert Bates, Kathy Sierra ISBN: 0596005717 Publisher: O'Reilly
Print Publication Date: 2003/10/01

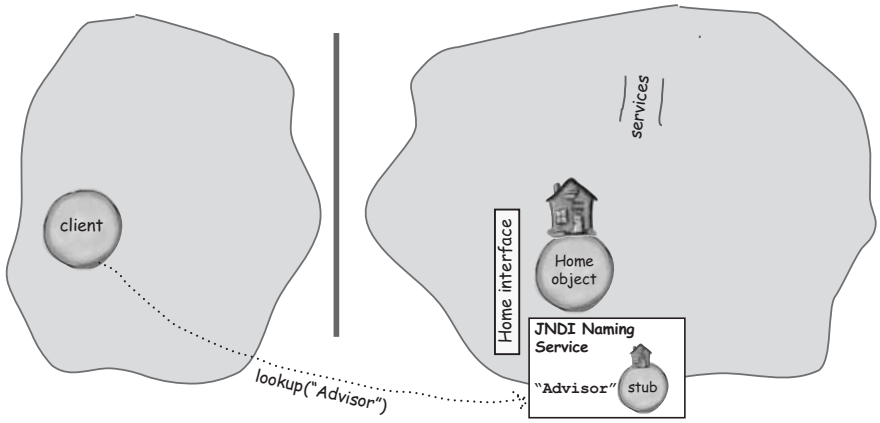
## Getting and using a Home for the AdviceBean

(This scenario assumes AdviceBean is a stateful Session bean.)

**1 The AdviceBean is deployed, and the server instantiates the AdviceBean Home object and registers it with JNDI.**




**2 The client does a JNDI lookup on the Home, using the registered name "Advisor".**

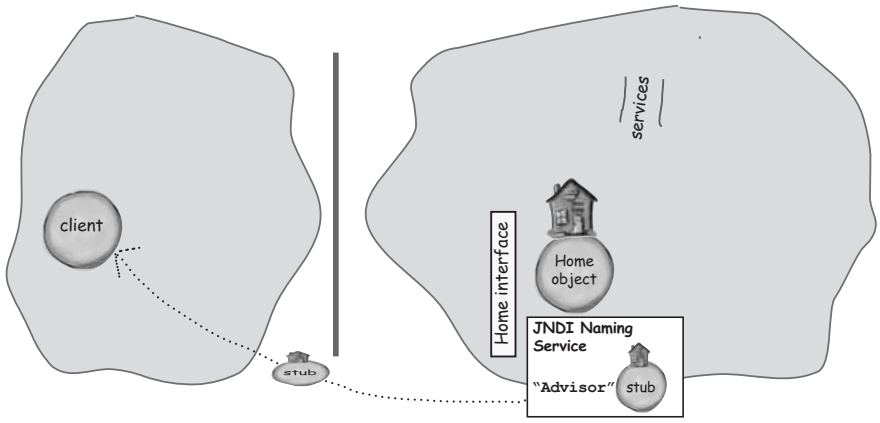






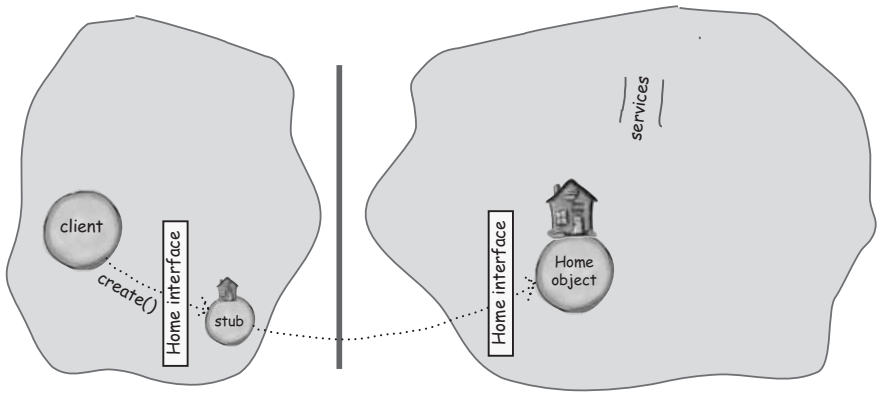
Head First EJB™ By Bert Bates, Kathy Sierra ISBN: 0596005717 Publisher: O'Reilly
Print Publication Date: 2003/10/01

**3 JNDI sends back a stub to the Remote Home object.**




**4 The client asks the Home for a reference to the Component interface, by calling create()**

**(In other words, the client wants to "create" a bean and get the stub back to the bean's EJBObject)**



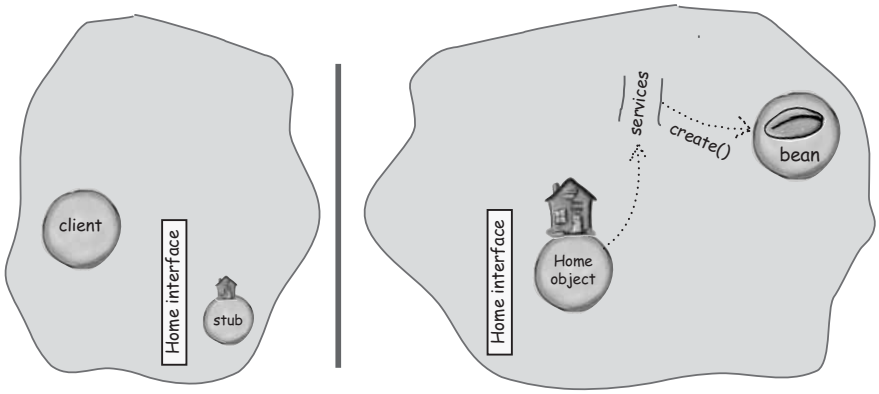




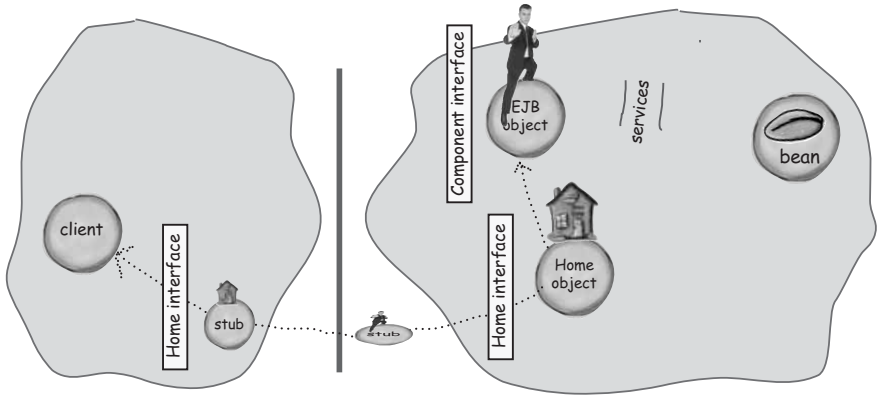
Head First EJB™ By Bert Bates, Kathy Sierra ISBN: 0596005717 Publisher: O'Reilly
Print Publication Date: 2003/10/01

## The Home object makes the EJBObject and sends back the stub

**5 Now the "services" kick in, and the bean is created**




**6 The EJBObject is made (the bodyguard for this newly created bean) and its stub is returned to the client.**

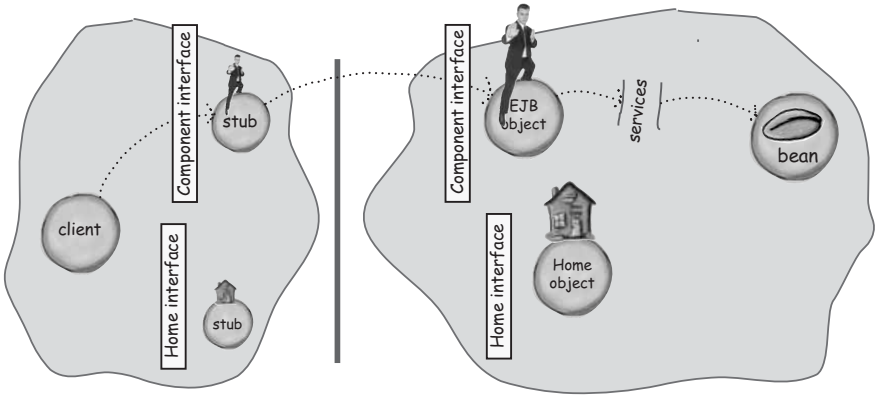




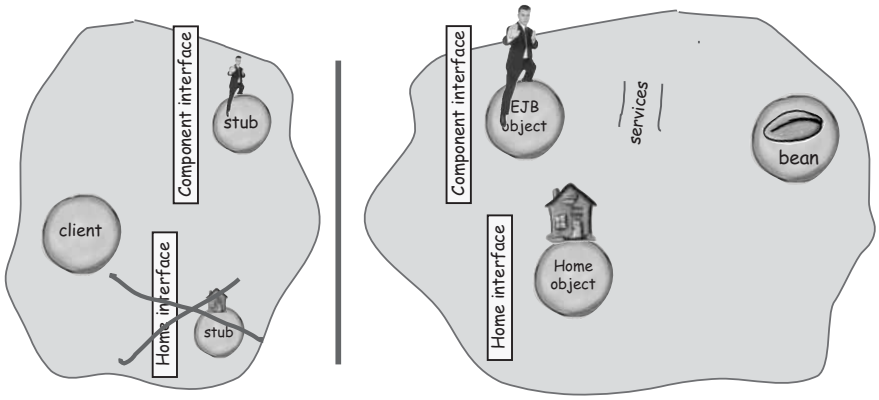


Head First EJB™ By Bert Bates, Kathy Sierra ISBN: 0596005717 Publisher: O'Reilly
Print Publication Date: 2003/10/01

**7** **Now, the client can (finally) do what he REALLY wants to do—call a business method on the bean! (Which of course, has to go through the Component interface.)**




**8** **The client can get rid of his Home stub if he doesn't want access to more beans of this type (AdviceBean), but he can still keep calling methods on the Component interface.**







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

## Sharpen your pencil

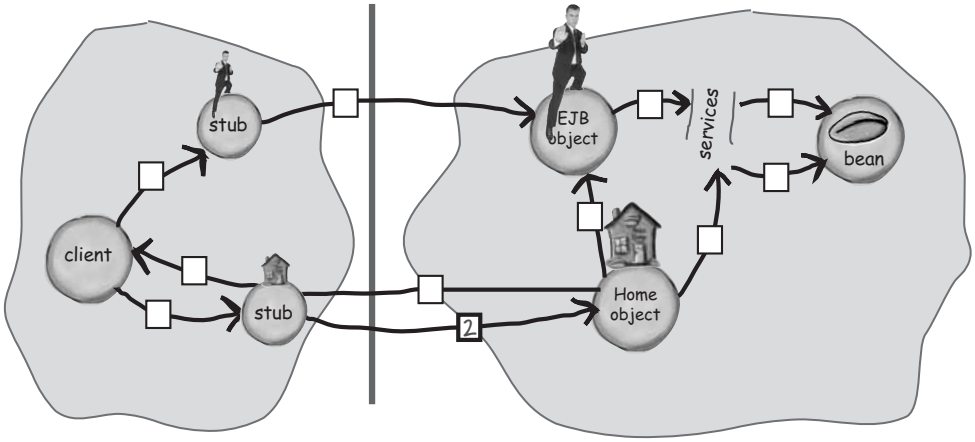
**EJB Lifecycle:**

**Client has only a Home stub but wants to invoke a business method on a bean.**

In the scenario below, assume the client has previously done a JNDI lookup and gotten back a stub to the Remote Home object.
Everything in the picture is what happens AFTER the client has the Home stub and now wants to get a reference to an EJBObject and ultimately call a business method on the bean.

Number the arrows (using the boxes over the arrows) in the order in which they occur. These arrows aren't necessarily direct method calls (although they might be), but rather arrows pointing to the next THING that happens. Tell a story for what happens at each arrow. There might be more than one right answer, depending on how you tell the story. Some arrows are missing; you can add them if you want, or, just assume some things are happening that you don't have arrows for.

Relax and take your time.
If you get stuck, flip back through the previous pages and study the diagrams.




**1.**

**2.** The stub tells the Home that the client wants to "create" a bean.

**3.**

**4.**

**5**

**6.**

**7.**

**8.**

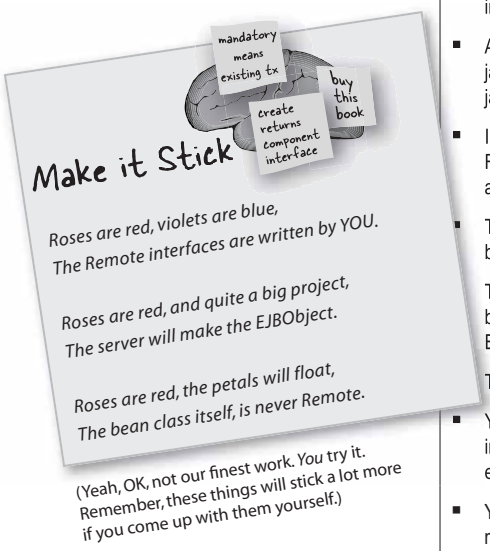
**9.**

**10.**

**11.**




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




(Yeah, OK, not our finest work. You try it. Remember, these things will stick a lot more if you come up with them yourself.)

## BULLET POINTS

- Beans that are exposed to remote clients have two Remote interfaces one for the EJBHome and one for the EJBObject.
- A Remote interface must extend (directly or indirectly) java.rmi.Remote, and all methods must declare a java.rmi.RemoteException.
- In EJB, the interface that extends EJBObject is called the Remote Component interface. It is where the business methods are declared.
- The client never calls methods on the bean itself because the bean is NOT a Remote object.

  The container implements the Remote Component interface by building a class that implements it. This class is used to make the EJBObject for the bean. (The bean's bodyguard.)

  The container also creates the stub to the EJBObject.
- You create the Remote Component interface by writing an interface that extends javax.ejb.EJBObject (an interface that extends java.rmi.Remote).
- You also create the bean class where the actual business methods are implemented (despite the fact that the bean class technically doesn't implement the Remote Component interface).
- The Home is the factory for the bean. Its main job is to hand the client a reference to the bean. But remember, the client can never truly get a reference to the *bean*—the best the client can do is to get a reference to the bean's EJBObject.
- You create the Home interface by writing an interface that extends javax.ejb.EJBHome (an interface that extends java.rmi.Remote).
- The container is responsible for implementing the Home interface by building a class that implements it, and the container also generates the stub for the Home.
- There is only one Home per deployed bean. For example, a ShoppingCart bean would have a single ShoppingCart Home, regardless of how many ShoppingCart beans have been created.

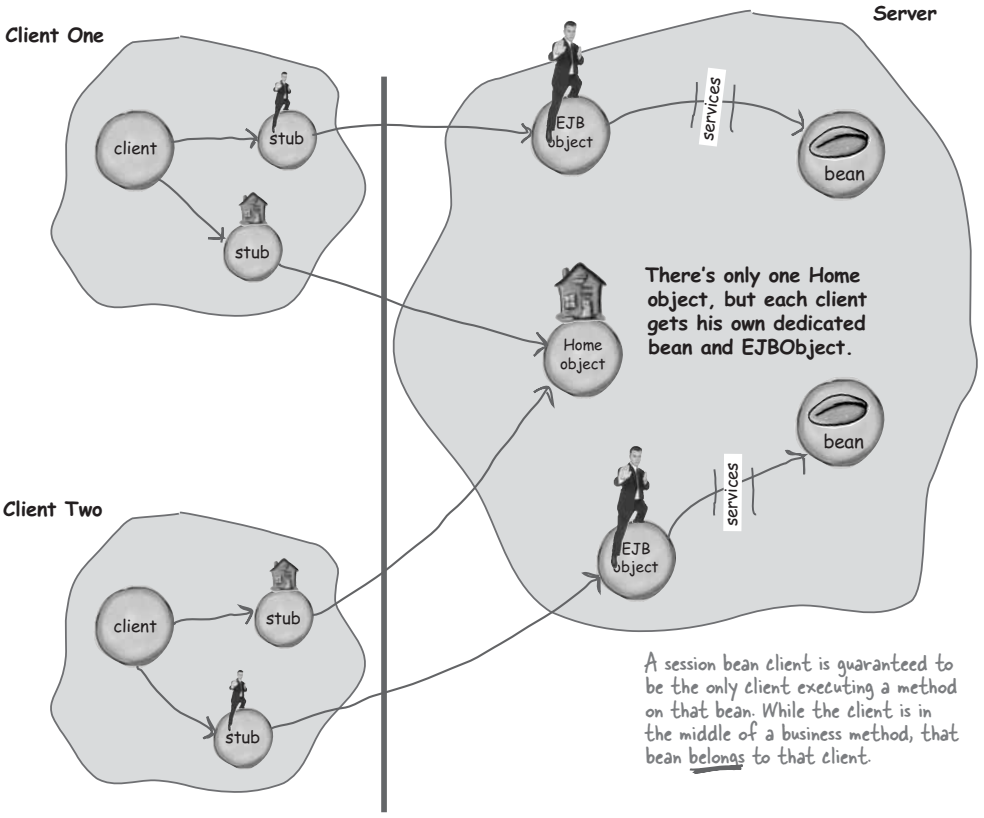



Head First EJB™ By Bert Bates, Kathy Sierra ISBN: 0596005717 Publisher: O'Reilly
Print Publication Date: 2003/10/01

# Architectural overview: Session beans

## Clients share the Home, but never the bean.

Each client gets his own EJBObject reference and his own bean. The client never shares a bean with another client, although the meaning of "shares" depends on whether the bean is stateful or stateless. (We'll see that in the next chapter.) However, there's only one Home object for this particular bean type (say, AdviceBean), so both clients have a stub to the one and only Advice Home. Both clients ask the same Advice Home for a reference to an Advice bean. (Of course, the client never gets the reference to the *bean instance*, but instead gets a reference to the bean's EJBObject. And since EJBObject is Remote, the clients gets a *stub.)*





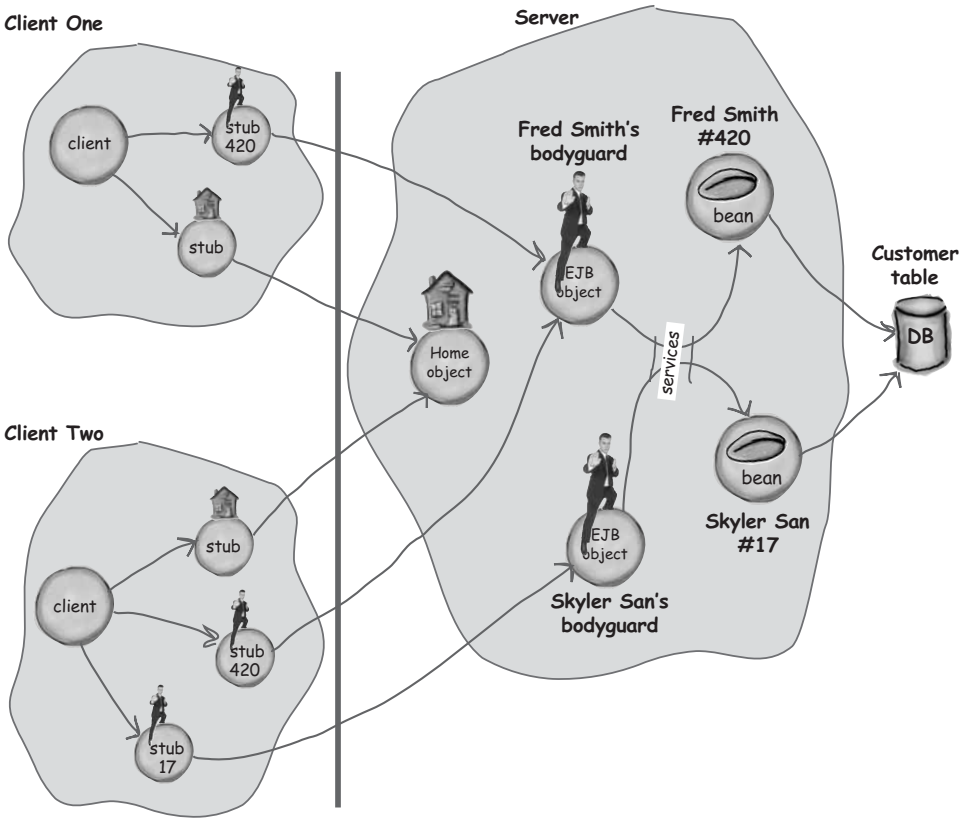


Head First EJB™ By Bert Bates, Kathy Sierra ISBN: 0596005717 Publisher: O'Reilly
Print Publication Date: 2003/10/01

# Architectural overview: Entity beans

### Clients share the Home, and may share the bean.

Each client has his own reference to the one and only Home for this bean (say, CustomerBean). But, if two clients are trying to access the same Customer (Fred Smith #420), then both clients have a reference to the same EJBObject. The EJBObject for #420. In other words, the EJBObject is the bodyguard for a particular Customer (like Fred Smith). If all the clients are trying to access Fred Smith #420, they will each have their own stub, of course, but all stubs will communicate with the same Remote EJBObject. And there will be only one bean representing Fred Smith #420. If a client wants to access two different customers, though, the client will have two stubs, and those stubs will be for two different EJBObjects, one for each customer. And that also means two different *beans*.



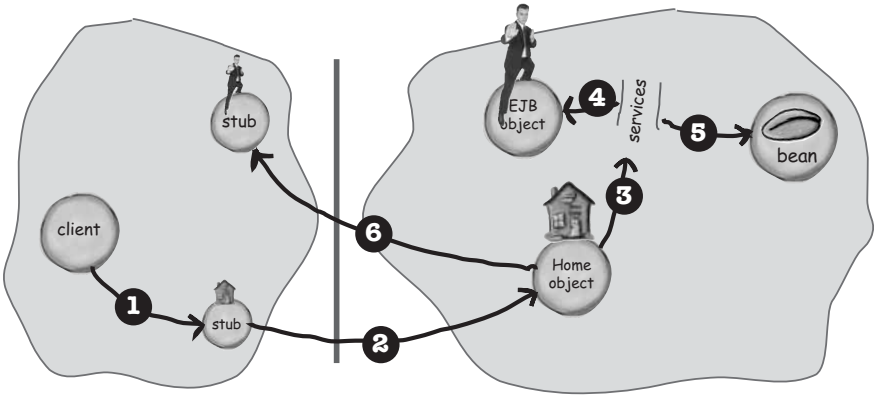




Head First EJB™ By Bert Bates, Kathy Sierra ISBN: 0596005717 Publisher: O'Reilly
Print Publication Date: 2003/10/01

# Architectural overview: Creating a Stateful Session bean

After getting a Home stub, the client calls "create" on the Home. The Home creates the bean and the EJBObject for the bean and hands back the EJBObject stub.



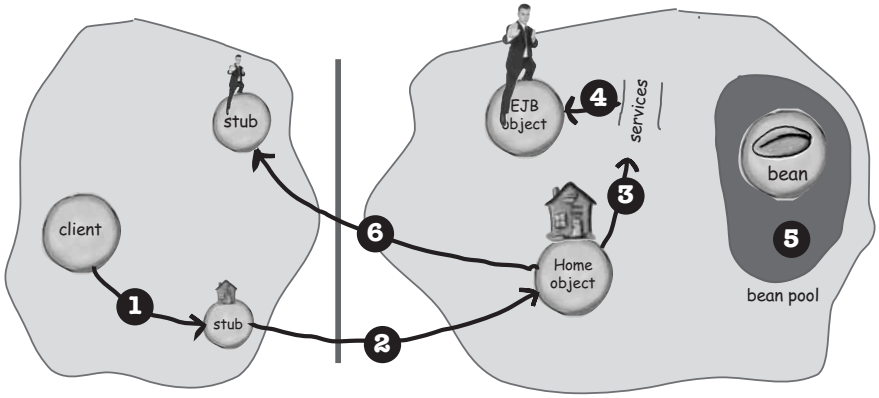

1. The client calls create() on the Home stub (create() is a method in the Home interface).
2. The stub sends the create() call to the Remote Home object.
3. The Home object steps in and adds its services.
4. The EJBObject is created/instantiated for the bean.
5. The bean itself is instantiated.
6. The EJBObject stub is returned to the client, so the client can call business methods on the Component interface.




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

# Architectural overview: Creating a Stateless Session bean

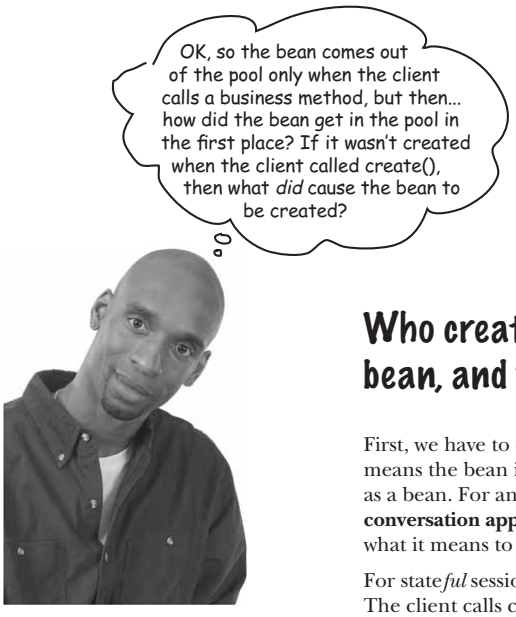
After getting a Home stub, the client calls "create" on the Home. The Home gives the client a stub to an existing EJBObject but does not associate a bean with this EJBObject! Instead, the bean stays in a pool until the client uses the EJBObject stub to call a business method.




1. The client calls create() on the Home stub (create() is a method in the Home interface).
2. The stub sends the create() call to the Remote Home object.
3. The Home container steps in and adds its services.
4. An EJBObject is created for this client.
5. The bean stays in the bean pool! It comes out only to service an actual business method, if the client invokes one on the EJBObject stub.
6. The EJBObject stub is returned to the client, so the client can call business methods on the Component interface.




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




## Who creates the stateless session bean, and when?

First, we have to define what *create* means. For a session bean, it means the bean instance is physically instantiated and initialized as a bean. For an entity bean, it's completely different, so **this conversation applies just to session beans**. Later we'll get into what it means to create an entity bean.

For state*ful* session beans, the create is triggered by the client. The client calls create on a Home stub, and everything happens at that point—an EJBObject is instantiated for this new about-to-be-created-bean, and then the bean itself is created and linked to the EJBObject (the bean's bodyguard).

But for state*less* session beans, the *client* create and the actual creation of the *bean* are decoupled. In other words, just because the client calls create on a Home stub doesn't mean a bean will be created at that point.

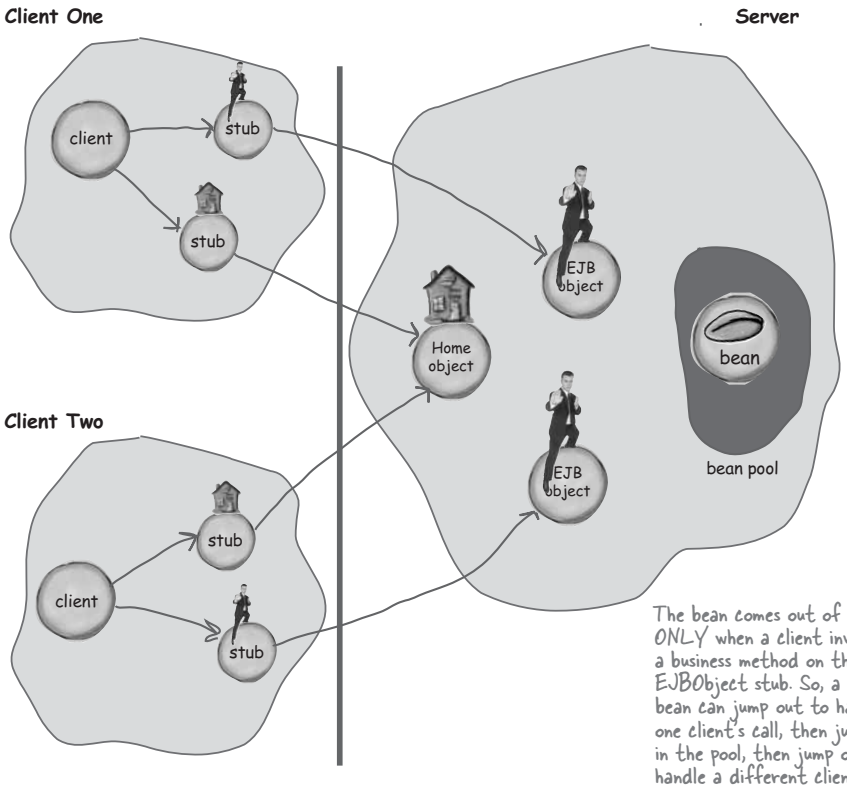
State*less* session beans aren't created until the container decides it needs one, and that's really up to the container. It might, for example, make a bunch of bean instances (i.e. create some beans) and plop them in a pool before even a single client has asked for one (by calling create on a Home stub). Or, the container might make just-in-time beans, and wait until the client invokes a business method before going to the trouble of physically creating the bean.




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

## Stateless session beans are more scalable

**Clients don't share EJBObjects, but the same bean can service *multiple* EJBObjects. Just not at the same time. A single bean can handle multiple clients, as long as only one client at a time is in the middle of a business method invocation.**




The bean comes out of the pool ONLY when a client invokes a business method on the EJBObject stub. So, a single bean can jump out to handle one client's call, then jump back in the pool, then jump out to handle a different client's call, and then jump back in the pool...




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


Why does the "pool" architecture work to make stateless session beans more scalable, but not stateful session beans? Why can't stateful session beans use the bean pool?

## there are no Dumb Questions

**Q: Here's something that is REALLY starting to annoy me... why do you have a Component interface and a Home interface when the Remote objects are called the Home and the EJBObject? Why isn't it just the EJBObject interface and the HomeObject interface? Or the Home and the EJB interfaces? Why the inconsistency?**

A: Actually, that really pisses us off too. But you'll get used to it. Be thankful, it was even worse if you learned EJB prior to version 2.0 when it was called the Home and the Remote. Now that was a real problem because, first of all, *both* the Home and the Remote were Remote in the java.rmi sense. Second, as of EJB 2.0 you can have a Home and a Remote that aren't... actually... Remote. So, they had to change the name to Component interface instead of calling it THE Remote interface, since it might, in fact, not be Remote. If you just remember that the Component interface is where the business methods are, and the Home is where the, um, *Home* methods are, you should be fine. Just remember, Component = business = EJBObject (or EJB*Local*Object, but we won't go there until the next chapter).


**T F** A stateful session bean can be shared between multiple clients.

**T F** An entity bean can be shared between multiple clients, as long as the entity being shared is the same.

**T F** Stateless session beans are created when the client invokes create on the Home.

**T F** Stateful session beans are created when the client invokes create on the Home.

**T F** There must be one stateless session bean per client, for as long as the client holds a reference to an EJBObject.

**T F** There must be one stateless session bean per client, for as long as the client is in the middle of a business method invocation on the bean.

**T F** Each entity bean must have its EJBObject.

Answers: F T F T F T T




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

## there are no Dumb Questions

**Q: How does this work? Is there one bean pool for all beans, or one bean pool for a particular type of bean?**

A: In reality, we don't know what the container implementation is, but conceptually there's one pool for every bean type. So, if you deploy an AdviceBean and a WeatherBean, and both are stateless session beans, they'll each get their own pool.

**Q: Does each stateless session bean have its own EJBObject?**

A: Sort of. A stateless session bean doesn't need a bodyguard until he's actually involved in a method invocation. So, the client gets a reference to an EJBObject, but the bean isn't associated with that EJBObject until the client calls a business method. At that point, the bean slides out of the pool to service the method. So, the EJBObject the client has is for a *kind* of bean (AdviceBean, WeatherBean, etc.) but not a specific *instance* of a bean.

**Q: Why don't stateful session beans have a pool?**

A: Have you already thought about this in the Brain Power on the previous page? Because if you haven't, don't read any further until you've come up with your own ideas. If you made it this far, we assume that you already know the answer and we're just confirming it...

A stateful bean, remember, holds client conversational state. That means the bean has to save client-specific state (in other words, it has to remember things about this client) across multiple method invocations from the client.

Think of a shopping cart again—a stateful bean needs to remember what's in the client's cart each time the client calls addItemToCart(). A stateless bean, on the other hand, doesn't have to remember anything on the client's behalf, so to a client, one stateless bean (of a particular type) is as good as any other bean of that type.

If the AdviceBean simply returns a piece of advice not connected in any way to anything it told the client before (or anything the client told *it*), then there's no need for the AdviceBean to be stateful. In that case, each time a client calls getAdvice() on the Component interface (i.e. the EJBObject), any AdviceBean is just as capable of running the method as any other.

On the other hand, if the AdviceBean were modified to return, say, random but non-repeating advice, then the AdviceBean would have to be stateful, so that it could keep track of previous advice and never repeat it.

**Q: How long does a stateful bean keep client-specific state?**

A: Only for the life of the session. A session continues until the client tells the bean he's done (by calling remove() on the bean's Component interface), or the server crashes, or the bean times out (we'll cover that in the Session lifecycle chapter).

**Q: So stateful session beans aren't scalable?**

A: No, they are. Just not *as* scalable as state*less* beans.

**Q: How can a stateful bean be scalable if you always need one bean for every client?**

A: You do need a separate bean allocated for each client, but not every bean has to be actively consuming resources. If the stateful session bean client is taking a long time between method calls, the stateful bean can be temporarily taken down and put in a state called passivation. This state preserves the client-specific state, of course, but reduces the number of beans currently alive in the server. The bean comes out of passivation and back into active duty (activation) when the client calls a business method.

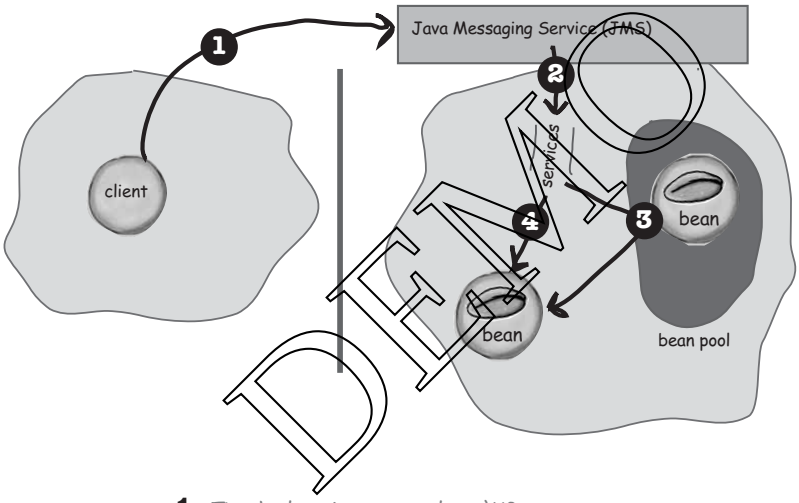



Head First EJB™ By Bert Bates, Kathy Sierra ISBN: 0596005717 Publisher: O'Reilly
Print Publication Date: 2003/10/01

*message-driven bean* overview

## Architectural overview: Message-driven beans

Message-driven beans don't have a client view. That means they don't have interfaces (Remote or local) that expose methods to the client. In other words, message-driven beans don't have a Home or EJBObject. They don't have a Home interface or a Component interface.




1. The client sends a message to a JMS messaging service.
2. The messaging service delivers the message to the container.
3. The container gets a message-driven bean out of the pool.
4. The container delivers the message to the bean (by calling the bean's onMessage() MessageListener interface method).




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


## What goes where?

Place the objects and classes in the appropriate spot on either the client, the server, or in both places (yes, you can reuse an object). Note: not all the pieces are here, so when you're done, if you can think of other things that should go into the picture (classes or objects) draw them in!

client

server

EJB object

bean

stub

DB

stub

Home object

client class

bean class

home stub class

EJBObject interface

home class





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

*bean table*


## Organize your beans

Finish the table by putting in a checkmark (even better if you add notes) in the boxes corresponding to the labels that apply to that bean type. We've done one of the boxes for you. If you get stuck, go back through the previous two chapters. You might have to make your best guess on a few things. That's OK—you'll have it all worked out way before the end of the book. We believe in you. You can do it. [cue theme song from "Rocky"]

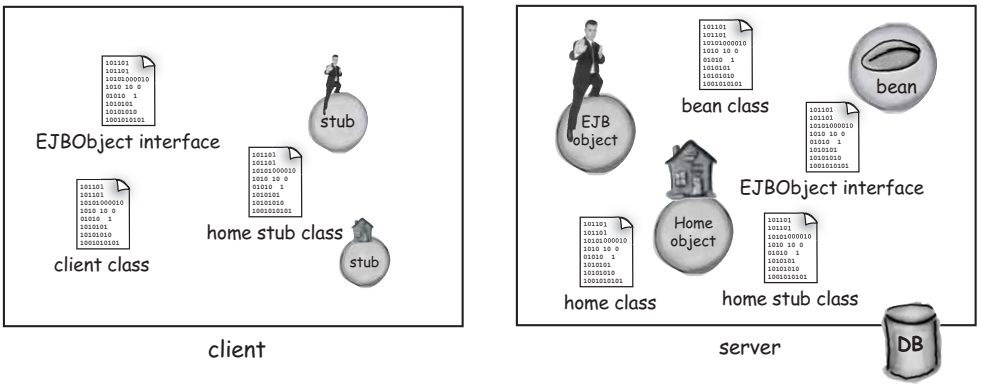
| | Stateful Session Beans | Stateful Session Beans | Entity Beans | Message-driven Beans |
|---|---|---|---|---|
| Uses a pool | Yes. Since they don't keep any client-specific data, you don't need one per each client. | | | |
| Multiple clients can have a reference to the same bean | | | | |
| Guaranteed to survive a container crash | | | | |
| Has a client view | | | | |
| Allows asynchronous communication | | | | |
| Represents a process | | | | |
| Represents a "thing" in an underlying persistent store (like a database) | | | | |




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

Exercise Solutions

## What goes where?




NOTE: you won't find a finished solution for the Organize Your Beans table. We want YOU to fill this out. It's yet another special Learning Opportunity for which you'll always remember us with fondness.




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


You're in luck. This chapter is all background knowledge, so there aren't any objectives or exam questions.

Appreciate the moment—Chapter 3 is just a page turn away...




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