Table of Contents

9 EJB transactions

The Atomic Age

It was a long transaction, but she finally committed. She had plenty of time to rollback, but I just kept catching all the exceptions, so it all worked out in the end.

Transactions protect you. With transactions, you can take a risk. You can try something BIG, knowing that if anything goes wrong along the way, you can just pretend the whole thing didn't happen. Everything goes back to the way it was *before*. The idea is simple—you either *commit* to *everything* in the transaction, or you *rollback*, so that nobody sees what you were trying (but failed) to do. Transactions in EJB are a thing of beauty—you can deploy a bean with customized transaction behavior *without* touching the bean's source code! But you can write transaction code, if you need to, so we'll learn that too.

OBJECTIVES

Enterprise Javabeans Transactions

Official:

11.1 Identify correct and incorrect statements or examples about EJB transactions, including bean-managed transaction demarcation, and container-managed transaction demarcation.

11.2 Identify correct and incorrect statements about the Application Assembler's responsibilities, including the use of deployment descriptor elements related to transactions, and the identification of the methods of a particular bean type for which a transaction attribute must be specified.

11.3 Given a list of transaction behaviors, match them with the appropriate transaction attribute.

11.4 Given a list of responsibilities, identify those which are the container's with respect to transactions, including the handling of getRollbackOnly, setRollbackOnly, getUserTransaction, SessionSynchronization callbacks, for both container and bean-managed transaction demarcation.

(Note: we cover the part of 11.4 that deals with SessionSynchronization in the Session Bean chapter.)

What it really means:

You need to know the rules, and implications of, bean-managed (BMT) vs. container-managed (CMT) transaction demarcation. For example, you must know that both message-driven and session beans can use CMT *or* BMT but entity beans can use only CMT. And message-driven beans and stateless session beans using BMT must end the transaction before the end of the method, but stateful session beans are allowed to keep a transaction open across multiple method invocations from a client. You need to know that propagation of transactions in BMT is a one-way street: a BMT transaction can propagate out with a bean's method calls (i.e. be used by the called method), but an existing transaction context can never be propagated into a BMT bean. In other words, a BMT bean will run only in transactions the bean itself has started.

You must be very clear about the effects of transaction attributes for CMT. For example, you must understand that message-driven beans can use only *two* of the six attributes (NotSupported and Required), because the others don't make sense for a message-driven bean (no calling transaction can ever be propagated into a message-driven bean because it is only the container that invokes the bean's onMessage() method. You also have to know the methods of each bean type (session, entity, or message-driven) for which transaction attributes must be specified. For example, an entity bean's create() method is transactional, but a session bean's create() method runs in "an unspecified transaction context." You must be able to specify transaction attributes in the deployment descriptor.

Finally, you need to know what your bean can count on from the container when it comes to transactions. For example, you must know that if you invoke getRollback-Only on a bean's context, the container must ***not*** commit

An EJB transaction is an atomic unit of work.

A transaction means you've wrapped some work (statements, method calls, whole methods, access to a database, etc.) into a single unit in such a way that either everything succeeds, or everything reverts to its previous state.

In other words, you either commit or rollback the whole atomic unit.

Either it all works, or we just forget the whole thing ever happened.

Shopping cart checkout: a quintessential EJB transaction example.

Imagine you have an online shopping cart system. When it comes time to checkout, what needs to happen? At the very least:

- Have user ***confirm*** order
- Validate and debit user's ***credit card***
- Remove purchased items from ***inventory***
- Create and submit a shipping ***order***

These must all happen as one unit— if ANY of this goes bad, ALL of it should rollback as though none of this ever happened...

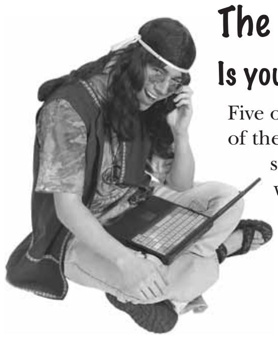
You don't want to debit the inventory if the credit card isn't valid. And you don't want to submit a shipping order if the items aren't yet in stock. And you don't want any of it to happen if the user doesn't confirm the order! If any of these things go wrong, you want your transaction to end with a rollback, rather than a commit. Think of the horror you'd go through if you couldn't do a transaction rollback. Imagine that you went through the first three out of the four steps only to find the user doesn't confirm the order. You would have to go back and add money to the user's credit card, cancel the order, and put the items back in inventory.

Relax

You don't need to know about JTS, XA, or any other transaction APIs except javax.transaction.UserTransaction.

You won't be tested on any of the lower-level transaction API details. For example, you don't need to know anything about HOW the server/container communicates with a transactional resource such as a database. And although EJB supports distributed transactions, you won't be asked about how it works. We know it's depressing that you won't get to show off your two-phase commit protocol prowess.

Head First EJB™ By Bert Bates, Kathy Sierra ISBN: 0596005717 Publisher: O'Reilly
Print Publication Date: 2003/10/01
Prepared for Linda Martin, Safari ID: doannn16@yahoo.com
User number: 896963 Copyright 2008, Safari Books Online, LLC.



The ACID test

Is your transaction safe?

Five out of five experts (plus pretty much the entire rest of the industry) agree on four characteristics of a good, safe transaction. (This is not just an EJB thing, by the way—the ACID test goes back long before Java was a gleam in Gosling's eye.) To put your transactions through the ACID test, make sure the transaction is:

Atomic

Either it all works, or it all fails (and rolls back).

A transaction isn't atomic if it's possible for *some* of it to commit while other parts don't.

Consistent

Whether it works (commits) or fails (rolls back), the data should stay consistent with the business logic reality. You'd have real trouble if, say, you could take items out of inventory without actually submitting an order. You'd end up with items that exist in the *real* inventory (i.e. in a warehouse somewhere), but that don't show up in anybody's computer records.

Isolated

Let's say you have two different transactions running, potentially hitting the same database. You don't want the effect of one transaction to corrupt the state of another transaction. In other words, the transactions should be protected (*isolated*) from one another. Isolation is very similar to thread synchronization—you don't want one transaction reading some data (with the intention of acting on it) if that data is smack in the middle of another transaction that hasn't finished committing its own changes to the data.

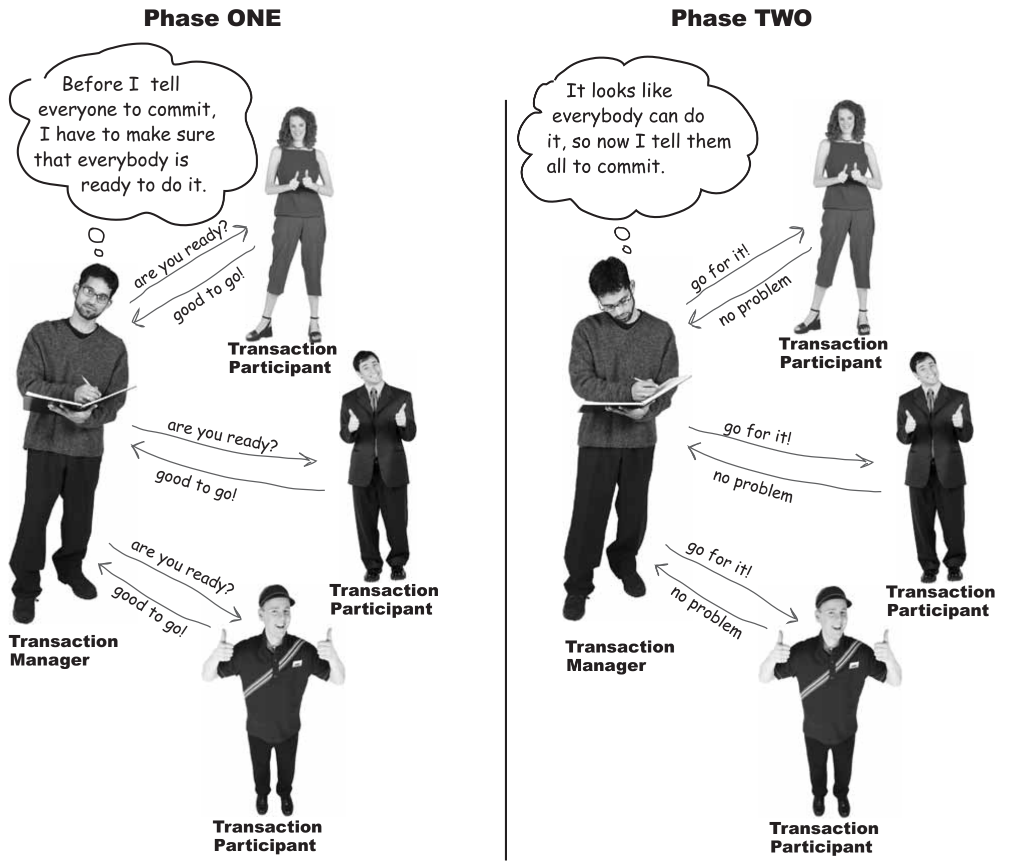
Durable

Once a transaction commits, the changes made by that transaction must become permanent! Even if the server goes down, it must come back up and finish what it started to commit.

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

Distributed transactions: two-phase commit

Most EJB containers support distributed transactions through a two-phase commit protocol. If you're a transaction manager (like a J2EE server), you might have multiple participants, including a database, another bean, and another server on the network. Once you've told everyone to commit, there's no good way to undo it, so before you give the signal to commit, you need to make *sure* that everyone can do what you're asking. As the transaction manager, your job is to find out if everyone is ready to perform (update the database, debit the account, etc.), and then, depending on the results, tell them all to do it (commit) or tell them all to forget it (rollback).



How it works in EJB

Transactions can propagate through method calls

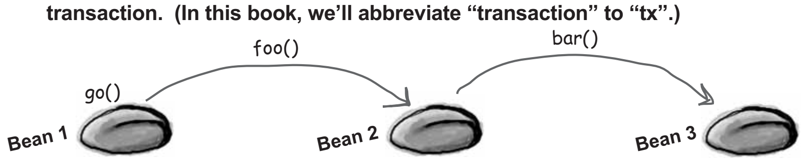
When a bean is running code in a transaction, and calls a method on another bean, three different scenarios are possible:

A) The called method runs ***in the caller's transaction.***

B) The called method runs ***without a transaction.***

C) The called method runs ***within its own new transaction.***

(A) The transaction started in the first method propagates to all other methods in the call stack. All called methods run within the same transaction. (In this book, we'll abbreviate "transaction" to "tx".)



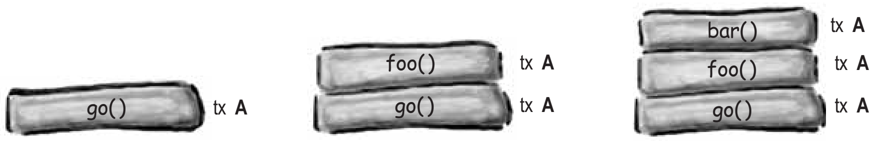
Bean 1's go() method starts a transaction (tx A).

Bean 1 calls method foo() on Bean 2, and the transaction is propagated into the method call on Bean 2.

Bean 2's foo() method runs in the transaction from Bean 1 (tx A) and calls method bar() on Bean 3. The transaction (tx A) is propagated into the method call on Bean 3.

Bean 3's bar() method runs in the transaction (tx A) propagated from Bean 2.

If a system exception (like EJBException) happens in any of the methods in tx A, the whole transaction rolls back.



So, what does it mean for multiple methods to run in the same transaction context? That depends on the bean type and what the beans do in their code. For example, imagine Bean A has a method with JDBC code that does an update to a database row. If any *other* method in the same transaction causes a rollback, Bean A's update won't commit, even if the database would have been more than happy to do it.

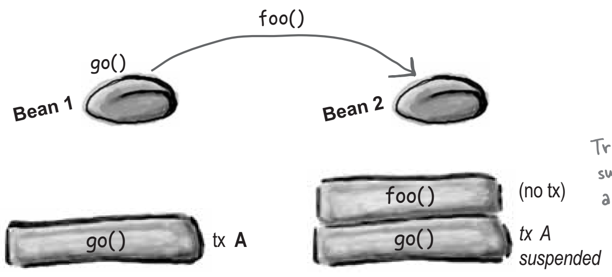
Head First EJB™ By Bert Bates, Kathy Sierra ISBN: 0596005717 Publisher: O'Reilly
Print Publication Date: 2003/10/01

Some transactions don't propagate

The caller's transaction might be suspended

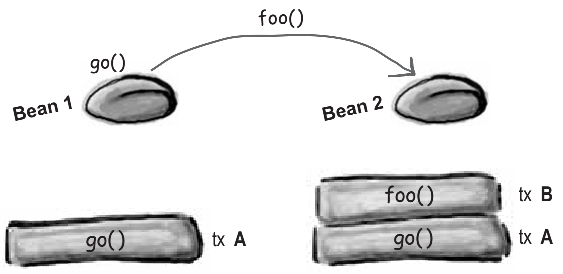
If a transactional bean method calls another method, the called method (whether it's in the same or a different bean) might not run in the same transaction. The called method, in that case, will run with either a brand new transaction, or with no transaction at all. (In a few minutes, we'll look at how the container decides whether to propagate the transaction, run without one, or start a new one.)

(B) The first transaction is suspended and the second method runs *without* a transaction.



Transaction A is temporarily suspended and foo() runs without a transaction.

(C) The first transaction is suspended and the second method runs within a *new* transaction.



Transaction A is temporarily suspended and foo() runs in its own new transaction (tx B).

How do I make (or get) a transaction?

Two ways: code it or declare it

The container manages your transactions, but you have to tell it how. You can either put transaction code in your bean class, or you can put transaction declarations in the DD. By far, the most common approach is to use the DD, because it's simpler, supports bean reuse, and is the *only* way you can do transactions for entity beans. By putting transaction information in the DD instead of in code, you can deploy the same bean multiple times and get different transaction behavior each time without ever touching the code!

***Bean*-managed transactions (BMT)**

① **Write transaction code in your bean.**

tell the DD that you're taking care of transactions

.java

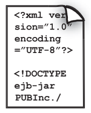
```
<transaction-type>Bean</transaction-type>

UserTransaction ut = context.getUserTransaction();
ut.begin();
// transactional code
ut.commit();
```

programmatically

OR

***Container*-managed transactions (CMT)**

② **Declare transactions in the DD.**


<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ejb-jar PUBInc./

.xml

```
<transaction-type>Container</transaction-type>

<method>
   <ejb-name>MyBean</ejb-name>
   <method-name>bar</method-name>
</method>
<trans-attribute>Required</trans-attribute>
```

declaratively

You can't use BOTH in the same bean! You can't mix BMT and CMT in one bean.

You CAN have a combination of BMT and CMT beans in the same ejb-jar, but each bean must have only one transaction type. On the exam, if you see transaction code in a bean, make sure that if you see the DD for that bean, it says: `<transaction-type>Bean</transaction-type>`

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

Transaction-related methods are in two interfaces

UserTransaction

begin()
commit()
getStatus()
rollback()
setRollbackOnly()
setTransactionTimeout()

Everything in the UserTransaction interface is for beans using bean-managed transactions (BMT).

Beans with container-managed transactions (CMT) aren't allowed to call ANYTHING in this interface.

javax.transaction.UserTransaction

EJBContext has methods for both BMT and CMT beans

EJBContext
getUserTransaction()

getCallerPrincipal()
getEJBHome()
getEJBLocalHome()
isCallerInRole()

setRollbackOnly()
getRollbackOnly()

For BMT beans only:
getUserTransaction()

For both BMT and CMT:
(all non-transaction-related methods)

For CMT beans only:
getRollbackOnly()
setRollbackOnly()

javax.ejb.EJBContext

(Super class of SessionContext, EntityContext, and MessageDrivenContext)

BRAIN POWER

Think about the differences between CMT and BMT.

If tied up and forced to pick one over the other, which would you pick?

☐ CMT ☐ BMT

Why? What are the pros and cons?

Making a BMT transaction

1. Get a **UserTransaction** reference from your EJBContext.
   `context.getUserTransaction()`
2. **Start** the transaction.
   `ut.begin()`
3. **End** the transaction (commit or rollback).
   `ut.commit()`
   `ut.rollback()`

(We'll look at the actual exceptions later when we get to the Exceptions chapter).

Get a reference to something that implements the UserTransaction interface by calling getUserTransaction() on the EJBContext.

```
public void checkOut() throws Exception {

    UserTransaction ut = context.getUserTransaction();


     ut.begin();


     anotherBean.validateCredit(customerNum);

     checkInventory();


     ut.commit();

     //or ut.rollback();

     doNonTxStuff();
}
```

this says, "Start a new transaction now."

call some other methods; we assume these methods will run under this new transaction (they might not, but we'll get to that later.)

now we're ending the transaction by telling the Container, "Go ahead and commit everything that happened in this transaction."

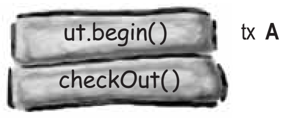
(or, we could end it with a rollback)

this method is called without a transaction. We can't tell from this code if doNonTxStuff() starts its own transaction. We'd have to see that method to know for sure.

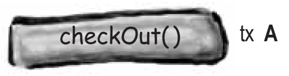
Head First EJB™ By Bert Bates, Kathy Sierra ISBN: 0596005717 Publisher: O'Reilly
Print Publication Date: 2003/10/01

Call stack of the checkOut() method

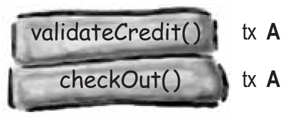
1. ut.begin() is called, and the transaction starts.



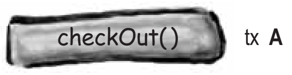
2. ut.begin() completes, and now the checkout() method is running in a transaction (tx A).



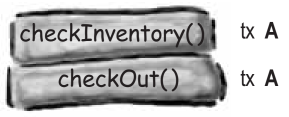
3. validateCredit() is called and runs in the same transaction as checkOut().



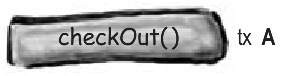
4. validateCredit() completes (pops off the stack) and the checkOut() method is still running in tx A.



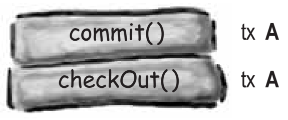
5. checkInventory() is called and runs in the same transaction (tx A).



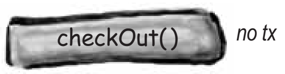
6. checkInventory() completes and pops off the stack. The transaction (tx A) is still open.



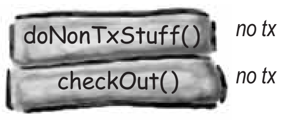
7. commit() is called, which ends the transaction.



8. commit() completes, and now checkOut() is running without a transaction.



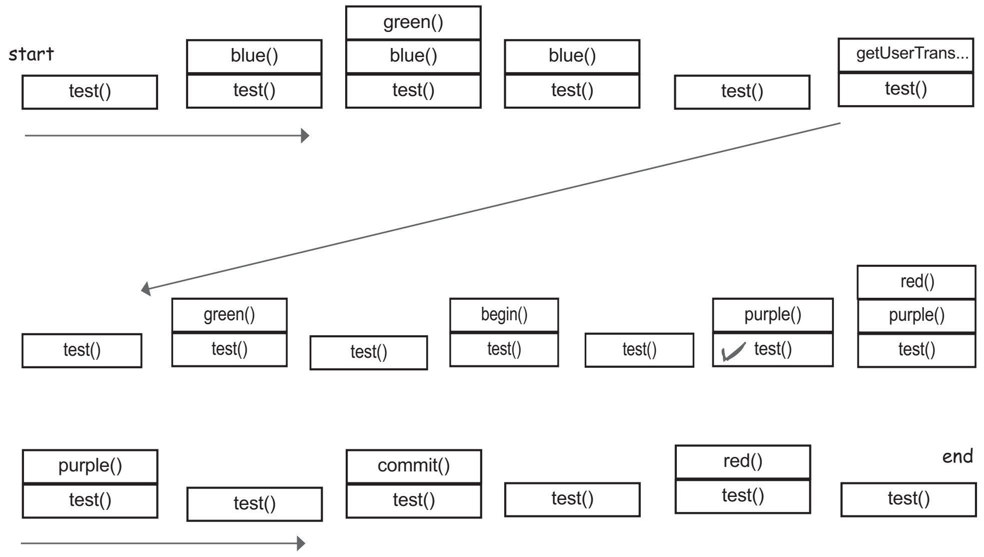
9. doNonTxStuff() is called without a transaction.



Sharpen your pencil

Using this code listing, mark the matching call stack frames with a checkmark if that frame is currently in a transaction. We did one in the middle for you.

```
public void test() throws Exception {
   blue();
   UserTransaction ut = ctx.getUserTransaction();
   green();
   ut.begin();
   purple();
   ut.commit();
   red();
}

void blue() { green(); }
void green() { }
void purple() { red(); }
void red() { }
```



Things you must NOT do with BMT

1 **A BMT bean must NOT start a transaction before ending the current transaction.**

```
public void go() {
    UserTransaction ut = context.getUserTransaction();
    ut.begin();
    doStuff();
    ut.begin();
    // more
}
```

NO! NO! NO! Can't start a transaction without completing (through a commit or rollback) the previous one. If you COULD do that, it would mean you could have "nested transactions".

BRAIN POWER

Imagine the implications of starting a new transaction before ending the current one. What might happen if you were allowed to do this?

Nested transactions are not allowed in EJB!

You're expected to know what the term "nested transaction" means, and what it might look like in code.

2 **A BMT stateless session or message-driven bean must NOT complete a transactional method without ending the transaction.**

```
public void go() {
    UserTransaction ut = context.getUserTransaction();
    ut.begin();
    doMore();
}
```

This is a problem! We're ending the method without ending the transaction we started. In other words, there's no commit() or rollback().

BRAIN POWER

Why are stateful session beans allowed to end a method without ending the transaction?

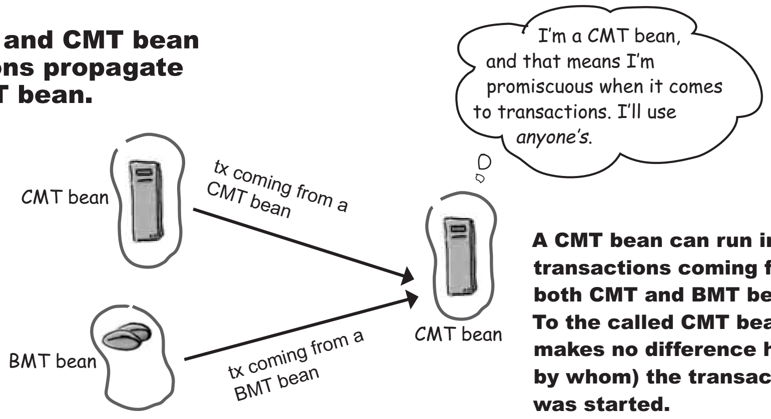
For a stateful bean, can you think of a scenario where you might want to do this (leave the transaction open)?

What might go wrong if you do this?

Only STATEFUL session beans can leave a transaction open at the end of a method.

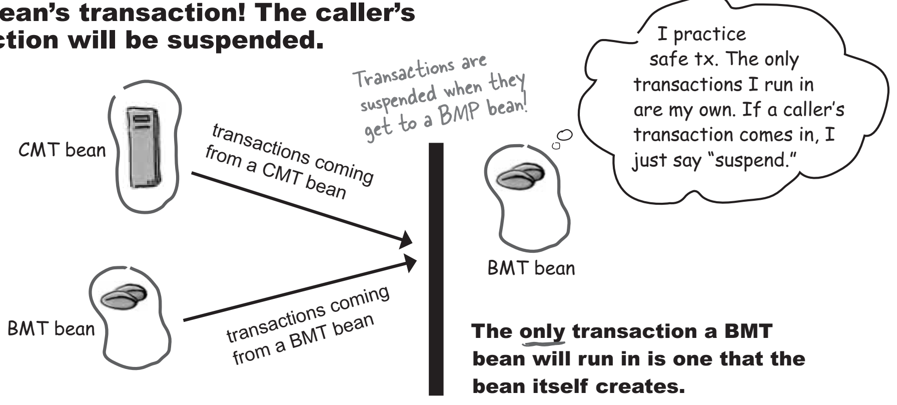
BMT transactions are one way: they can propagate out to a CMT bean, but no other transaction can propagate in to a BMT bean

Both BMT and CMT bean transactions propagate into a CMT bean.


I'm a CMT bean, and that means I'm promiscuous when it comes to transactions. I'll use *anyone's*.
CMT bean
tx coming from a CMT bean
BMT bean
tx coming from a BMT bean
CMT bean

A CMT bean can run in transactions coming from both CMT and BMT beans. To the called CMT bean, it makes no difference how (or by whom) the transaction was started.

A BMT bean will NEVER use any other bean's transaction! The caller's transaction will be suspended.


Transactions are suspended when they get to a BMP bean!
I practice safe tx. The only transactions I run in are my own. If a caller's transaction comes in, I just say "suspend."
CMT bean
transactions coming from a CMT bean
BMT bean
transactions coming from a BMT bean
BMT bean

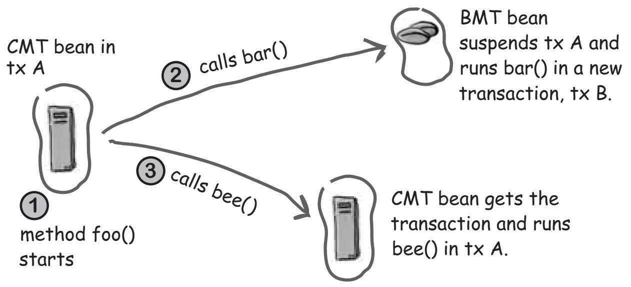
The only transaction a BMT bean will run in is one that the bean itself creates.

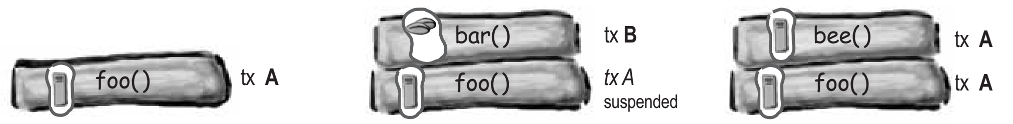
What does it mean to suspend a transaction?

If a transaction is in progress when a method on a BMT bean is called, the transaction is suspended. ***Temporarily.*** The transaction just sits there waiting for the BMT bean to complete its work. Work that's not part of the caller's original transaction. Once the BMT method finishes and is popped off the stack, the original transaction kicks back in, right where it left off.

Imagine this scenario: a CMT bean, bean one, is running a method foo() in a transaction (tx A) when it calls a method bar() on bean two (a BMT bean). Once bar() completes and pops off the stack, method foo() invokes another method, bee(), but this time the called bean is another CMT bean (bean three).

When a transaction is suspended, it waits until it can pick up where it left off. But this means that the things that happen while the transaction is suspended are NOT part of the same atomic unit. In other words, the things that happen while the transaction is suspended won't be rolled back if the suspended transaction (after it comes back to life) fails to commit.


CMT bean in tx A
1 method foo() starts
2 calls bar()
BMT bean suspends tx A and runs bar() in a new transaction, tx B.
3 calls bee()
CMT bean gets the transaction and runs bee() in tx A.



1. Method foo() of a CMT bean is running in transaction A. Method foo() then invokes method bar() on a BMT bean.
2. Method bar() suspends transaction A, and starts a new transaction, B. Method bar() runs in the new transaction B, then completes.
3. When method bar() completes, foo() resumes and picks up transaction A again. It then calls method bee() on bean three (a CMT bean). Method bee() runs in foo()'s existing transaction (A).

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

The UserTransaction interface

(javax.transaction.UserTransaction)

Sharpen your pencil

The UserTransaction interface has six methods for the things a BMT bean might want to do. Try to figure out the method names, based on the description of what you want to do. (The answers are at the bottom, upside down, so don't look down there.)

1. **Begin** a transaction

   `ut.________________________`

2. **End** a transaction

   ```
   ut.________________________
   // or
   ut.________________________
   ```

3. Mark a transaction for **rollback**

   `ut.________________________`

4. Find out the **status** of the transaction

   `ut.________________________`

5. Set the transaction **timeout**

   `ut.________________________`

UserTransaction is for BMT beans only! A CMT bean is never supposed to get (or try to use) a reference to a UserTransaction.

UserTransaction is NOT for entity beans. Since entity beans MUST be CMT, if you see UserTransaction code in an entity bean, you know the code's not legal.

Relax ***There's nothing about transaction timeout on the exam.***

All transactions have some default timeout value, but you can change that with: setTransactionTimeout(anIntValue). As a bean developer, you'll probably never use anything but the default timeout value, so we don't test for it on the exam.

1. begin(), 2. commit() or rollback(), 3. setRollbackOnly(),
4. getStatus(), 5. setTransactionTimeout()

setRollbackOnly()

The sound of a transaction's death

When a bean calls setRollbackOnly() it means that transaction is going down. Once you invoke setRollbackOnly(), ***the transaction is doomed to never, ever commit.***

So, what does it mean to sentence a transaction to death? It means the transaction definitely won't commit. *Duh.* But it also means that any participant in the transaction (i.e. any bean) can check to see if the transaction is already marked for death.

Remember, a transaction started by a BMT bean might propagate to method calls that the BMT bean makes on CMT beans. A CMT bean in a BMT-started transaction might want to sentence the transaction to death, or at least *find out* whether it's already doomed.

In a CMT bean, setRollbackOnly() is how you tell the container that it ***must not commit the transaction.*** If you can figure out in your business logic that a transaction is going to end badly, call setRollbackOnly(). The container won't *end* the transaction at that point, but when it *does* end at its natural time, it definitely won't commit.

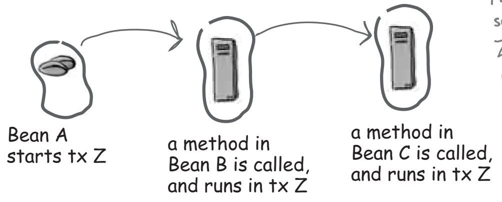
So, when *does* a transaction end? Assuming no System Exceptions are thrown, a transaction ends when the CMT method that started the transaction completes, or for a BMT bean, when the bean's code calls commit() or rollback().

It's tragic.

When you call the setRollbackOnly() method, you set a flag that can tell other beans that the transaction will end only one way... horribly.

If you know, in your code, that things aren't going to work, call this method.

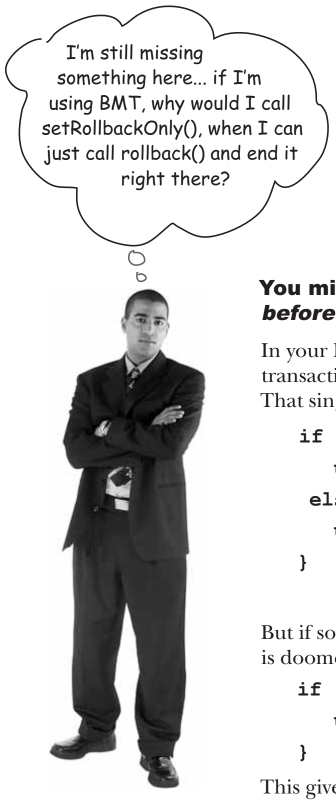
If the transaction is going to die, it's probably for the best. It was meant to be. It's going to a better place.


Bean A starts tx Z
a method in Bean B is called, and runs in tx Z
a method in Bean C is called, and runs in tx Z

Any bean in this transaction can call setRollbackOnly() to make sure transaction Z never commits.

If Bean B calls setRollbackOnly(), it won't matter what Bean C does -- the transactional code in Bean C will never be committed. Wouldn't it be nice if Bean C could find that out BEFORE Bean C does a bunch of work? We'll see that in a minute.

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


I'm still missing something here... if I'm using BMT, why would I call setRollbackOnly(), when I can just call rollback() and end it right there?

You might know HOW a BMT transaction should end *before* it's time to actually end it.

In your BMT code, you might have a single place at the end of the transactional code where you say either ut.commit() or ut.rollback(). That single place might be a simple *if* test:

```
if (thingsLookGood) {
   ut.commit();
 else {
   ut.rollback();
}
```

A typical example: at the end of the transaction, commit or rollback based on some conditions.

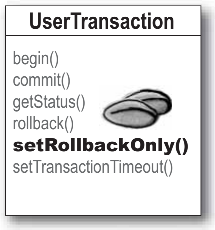
But if somewhere *earlier* in your code you can tell that the transaction is doomed, you should call setRollbackOnly():

```
if (thingsLookBad) {
   ut.setRollbackOnly();
}
```

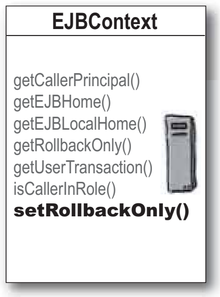
This gives other transaction participants a signal (if they care to check) that the transaction is already doomed.

Even if *your* BMT code doesn't call setRollbackOnly(), some *other* code in the transaction could have, so you might want to find that out. (In just a few more pages from now, we'll learn how a bean can check whether anyone has marked a transaction for rollback.)

setRollbackOnly() lives in TWO interfaces



Everything in UserTransaction is for BMT beans ONLY!



setRollbackOnly() in EJBContext is for CMT beans ONLY!

The methods in the UserTransaction interface are for BMT beans only; CMT beans can't use *anything* in UserTransaction.

The EJBContext interface, on the other hand, is for both BMT and CMT beans, *except for the two transaction methods.*

The setRollbackOnly() and getRollbackOnly() methods in EJBContext are off-limits to BMT beans.

Bottom line: BMT beans call setRollbackOnly() on a UserTransaction; CMT beans call setRollbackOnly() on an EJBContext.

A bean that calls setRollbackOnly() MUST be in a transaction!

You can call setRollbackOnly() from a BMT bean ONLY if you're inside a transaction. In other words, only within code that's between a ut.begin() and a ut.commit() or ut.rollback(). For a CMT bean, the method that calls setRollbackOnly() must have an appropriate transaction attribute (we'll get into that a few pages from now).

Be SURE you know the rules for BOTH of the setRollbackOnly() methods!

Be ready for the exam to test you on the use of setRollbackOnly() for both BMT and CMT beans.

***EJBContext**.setRollbackOnly()*

***UserTransaction**.setRollbackOnly()*

Remember, no single bean can ever use BOTH of these!

***CMT** beans can use **only** the **EJBContext**.setRollbackOnly()*

***BMT** beans can use **only** the **UserTransaction**.setRollbackOnly()*

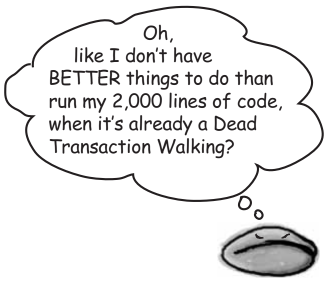
*Expect to see code examples where you'll need to figure out if the bean is BMT or CMT by looking at the code. And it won't be as obvious as a call to getUserTransaction(). If you see a call to EJBContext.setRollbackOnly(), for example, you know that this **must** be a CMT bean. So if that same bean later starts a UserTransaction, you know the code is illegal.*

getRollbackOnly()

Because life's too short for a bean to waste time

Once a bean has called setRollbackOnly(), the transaction is sentenced to death. It will never commit. But the transaction might still have a long way to go, with plenty of other methods in other beans, and with lots of heavy code.

Imagine you're a bean. How would *you* feel if the transaction were already doomed before your methods were called, *but nobody told you?*


Oh, like I don't have BETTER things to do than run my 2,000 lines of code, when it's already a Dead Transaction Walking?

CMT beans call **getRollbackOnly()** to find out if the transaction they're in is already doomed. If the transaction is never going to commit, why should the bean waste time with lots of code?

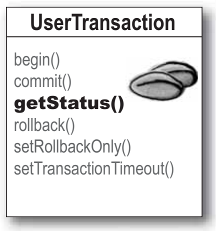
If you're a CMT bean, you can call getRollbackOnly() to find out if your transaction has already been sentenced to death. If it has, why bother doing any work?

```
if (!getRollbackOnly()) {
      saveWorld();
} else {
     abandonAllHope();
}
```

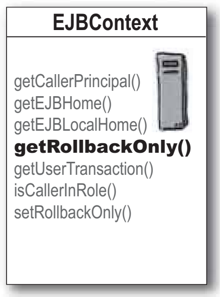
getRollbackOnly() is NOT for BMT beans! Only CMT beans can call getRollbackOnly()

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

BMT beans use getStatus() instead of getRollbackOnly()



There's no getRollbackOnly() in UserTransaction. BMT beans call getStatus() instead.



getRollbackOnly() is for CMT beans only.

The getRollbackOnly() method returns a boolean—true if the method has been marked for rollback, false if nobody's asked for a rollback. That's all a CMT bean can (and needs) to know about the transaction's status.

BMT beans, on the other hand, are more involved in controlling the transaction, and they might want to know a lot more. The getStatus() method in UserTransaction can tell you anything you'd ever want to know, and so much more, about how the transaction is doing.

Relax ***You don't need to memorize the status constants.***

The getStatus() method returns an int representing a constant for things like: STATUS_ACTIVE, STATUS_COMMITTED, STATUS_COMMITTING, STATUS_THINKING_ABOUT_COMMITTING (just kidding on this one), STATUS_MARKED_ROLLBACK, STATUS_ROLLING_BACK, and our personal and most useful favorite, STATUS_UNKNOWN.

These constants are defined in the javax.transaction.Status interface (which has no methods, just a pile of these status constants), and you might find them helpful if you're writing BMT code. But they're not on the exam. You DO need to know that the getStatus() method is in UserTransaction, and that it's the only way a BMT bean can find out if somebody called setRollbackOnly(), but that's it.

BMT can be a really BAD idea. BMT hurts bean reuse

Can you figure out why?

Think about what you learned on the last few pages, especially about transaction propagation (the whole one-way thing).

If you write a BMT bean, nobody else can ever include your bean in their transaction!

Your BMT bean puts up a big fat wall so calling transactions can't pass. Remember, a BMT bean will run only in the transactions the bean itself creates and starts. You defeat the whole point of a component model if you lock down the transaction demarcation inside the bean. Remember, the cool thing about a component model is that components can be mixed and combined in new ways to make new applications that the Bean Provider hadn't ever thought about. The purpose of the deployment descriptor is to give the application assembler a way to configure transactions specific to a particular application, ***without touching the bean code!***

If it's so bad to use BMT, why is it there?

Because it lets you do a few things you simply cannot do with CMT. But most of the time, you won't need these things.

With BMT, you can reduce the scope of a transaction.

Using CMT, you cannot mark a transaction at anything smaller than a single method. You put in the deployment descriptor which transaction attribute (we're getting there) goes with which method. You can't specify a part of a method. But with BMT, you can start the transaction and end it at a smaller scope. This can improve performance because the longer a transaction lasts, the more likely you are to hurt your concurrency. But the tradeoff—hurting your reuse—is almost never worth it, and there are usually *much* better ways of increasing the performance...

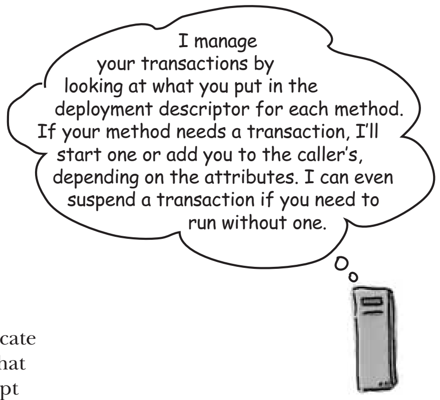
With BMT, you can leave a stateful session bean transaction open across multiple invocations from the client.

With BMT, you can open a transaction (call ut.begin()) in one method, and end the method without ending the transaction. (A big no-no for message-driven or stateless session beans.) But this is almost always a really bad design idea, so it's probably never going to be a good reason for BMT.

With BMT, you separate transaction commit status from message acknowledgment. This might be a good reason for BMT.

We covered this in more detail in the MDB chapter, but the short version is: with CMT, message acknowledgment is sent only when (and if) the transaction commits. In some designs, you can end up with poison messages.

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


I manage your transactions by looking at what you put in the deployment descriptor for each method. If your method needs a transaction, I'll start one or add you to the caller's, depending on the attributes. I can even suspend a transaction if you need to run without one.

Container-managed transactions

Now that we've looked at the do-it-yourself way to demarcate transactions, you'll see how easy it is with CMT. So easy that you don't write anything transactional in your code except maybe an occasional call to EJBContext.setRollbackOnly() or EJBContext.getRollbackOnly().

With CMT, transactions are started and completed (with either a commit or rollback) by the container, based solely on the deployment descriptor. You (OK, technically the Application Assembler) mark some attributes in the deployment descriptor and that's it.

Almost.

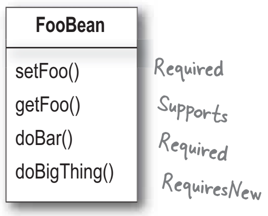
Unless you understand exactly what the six transaction attributes are, and the implications of how different attributes interact at runtime, you won't have a clue about whether your bean is even going to be in a transaction. Or, how big the transaction will be. Or, whether you've created a dangerous situation that could blowup at runtime.

Fortunately, there are only six. And the rules for how the container behaves with each of those attributes is very clear and simple.

How attributes work

① Mark a method with one of six transaction attributes:

- Required
- RequiresNew
- Mandatory
- Supports
- NotSupported
- Never



② When the method is called, the container uses the attribute to do one of five things:

- Run the method in the ***caller's transaction.***

OR

- ***Suspend*** the caller's transaction and start a ***new*** transaction.

OR

- ***Suspend*** the caller's transaction and run the method ***without*** a transaction.

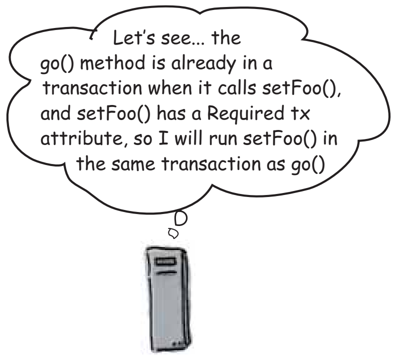
OR

- Throw an ***exception*** because the caller does ***not*** have a transaction.

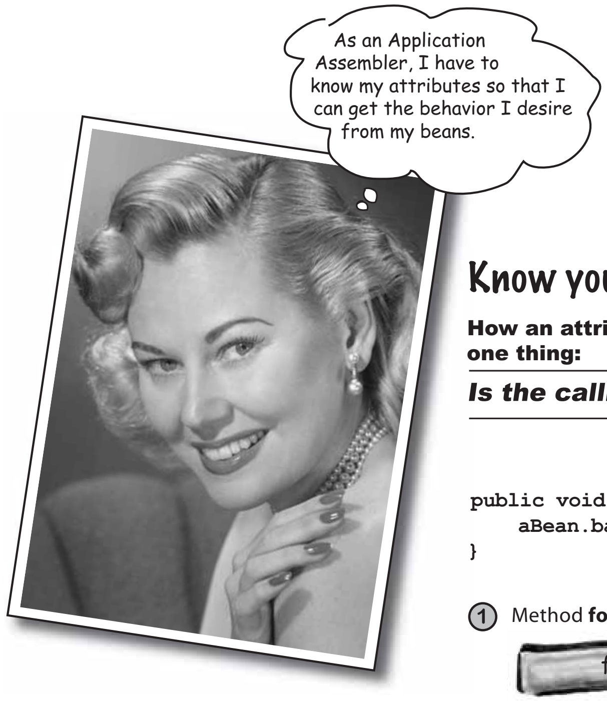
OR

- Throw an ***exception*** because the caller ***does*** have a transaction.

```
void go() {
    aFooBean.setFoo();
}
```


Let's see... the go() method is already in a transaction when it calls setFoo(), and setFoo() has a Required tx attribute, so I will run setFoo() in the same transaction as go()

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


As an Application Assembler, I have to know my attributes so that I can get the behavior I desire from my beans.

Know your attributes

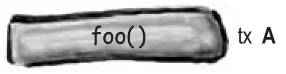
How an attribute affects behavior depends on one thing:

Is the calling method in a transaction?

```
public void foo() {
    aBean.bar();
}
```

foo() is in a transaction (tx A) when it calls bar() on another bean. The bar() method is marked as Required.

1 Method **foo()** is in a transaction (tx A).

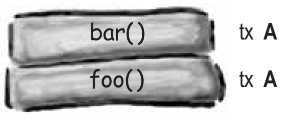


2 Method **bar()** is marked with the ***Required*** transaction attribute.

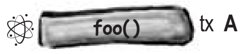
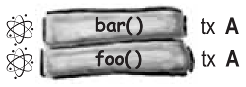
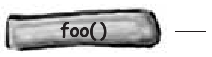
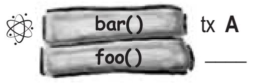
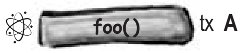
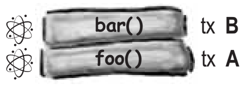
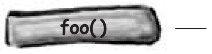
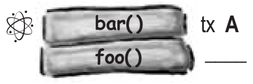
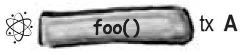
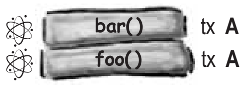
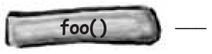
```
<method>
   <ejb-name>MyBean</ejb-name>
   <method-name>bar</method-name>
</method>
<trans-attribute>Required</trans-attribute>
```

The bar() method runs in the same transaction as foo() because bar() is marked Required and foo() has a transaction. If foo() had NOT been in a transaction, the container would have started a new transaction for bar() to run in.

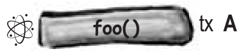
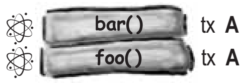
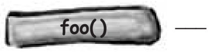
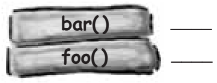
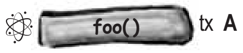
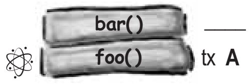
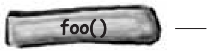
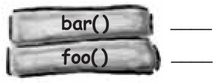
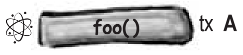
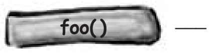
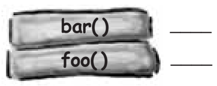
3 The **bar**() method runs in the caller's transaction (tx A).



Transaction attributes that require a transaction

Attribute for bar()	foo() transaction status	result
Required If the method is called with an existing transaction context, the method runs in that existing transaction. If there isn't a transaction, the container will start a new one.	 in transaction A (tx A)	 bar() runs in tx A
	 no transaction	 container starts a new transaction (tx A) for bar()
RequiresNew The method will always run with a new transaction. If the method is called with an existing transaction context, the caller's transaction is suspended until this method completes.	 in transaction A (tx A)	 container suspends tx A and starts a new transaction (tx B) for bar()
	 no transaction	 container starts a new transaction (tx A) for bar()
Mandatory Danger! Mandatory really means "RequiresExisting". If the method is called without an existing transaction context, the container throws an exception!	 in transaction A (tx A)	 bar() runs in tx A
	 no transaction	Exception!

Transaction attributes that do not require a transaction

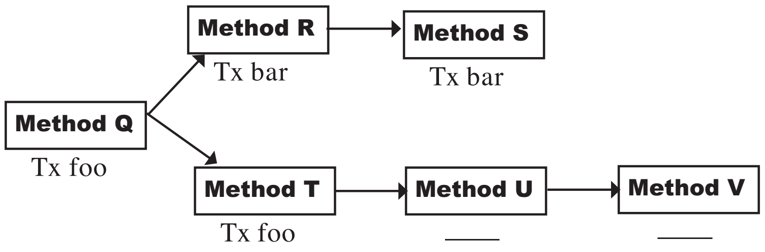
Attribute for bar()	foo() transaction status	result
Supports If the method is called with an existing transaction context, the method runs in that transaction. If there isn't a transaction, the method runs with an "unspecified transaction context."	 in transaction A (tx A)	 bar() runs in tx A
	 no transaction	 bar() runs with an "unspecified transaction context"
NotSupported If the method is called with an existing transaction context, the caller's transaction is suspended. Regardless of whether there is an existing transaction, the method will run in an "unspecified transaction context."	 in transaction A (tx A)	 container suspends tx A, bar() runs with an "unspecified transaction context"
	 no transaction	 bar() runs with an "unspecified transaction context"
Never Never means "No Pre-Existing". If the method is called with an existing transaction context, the container throws an exception. If there isn't a transaction, the method runs with an "unspecified transaction context."	 in transaction A (tx A)	Exception!
	 no transaction	 bar() runs with an "unspecified transaction context"

Sharpen your pencil

Know your attributes

The exam expects you to figure out which combination of attributes can (or will) lead to a particular outcome. You might be asked to look at a sequence of methods, where the methods show which transaction they're running in, and you have to figure out which combination of transaction attributes could make that scenario possible. They might be formatted something like this...

(The answers are at the bottom of the next page.)



QUESTION: Which two combinations of attributes would make this possible?

1) R-Supports, S-Required, T-Mandatory, U-NotSupported, V-Never

2) R-RequiresNew, S-Required, T-Required, U-Never, V-NotSupported

3) R-RequiresNew, S-Supports, T-Supports, U-NotSupported, V-Supports

4) R-Requires, S-Mandatory, T-Mandatory, U-Supports, V-Never

5) R-RequiresNew, S-Required, T-Required, U-NotSupported, V-NotSupported

Sharpen your pencil

For the exam (and bean developer life in general) you have to know some REALLY important rules about transactions, and it will be much easier for you if you take the time now to figure some of this out for yourself. Understanding is much better than memorizing, and it's not like you don't have enough to memorize as it is. You'll find all of these questions answered over the next few pages, but you should *really* try to do this first.

1. **Of the six transaction attributes, which one (or ones) must NOT be used by a bean that calls getRollbackOnly() or setRollbackOnly()?**

2. **Which transaction attribute (or attributes) must NOT be used by a message-driven bean?**
   (Hint: remember, a message-driven bean doesn't have a "client"; the container invokes the onMessage() method.)

3. **Under what circumstances do you think the container should automatically roll back a transaction?**
   If the bean gets a runtime exception?
   If the bean throws an application exception? (e.g. InsufficientFundsException?)

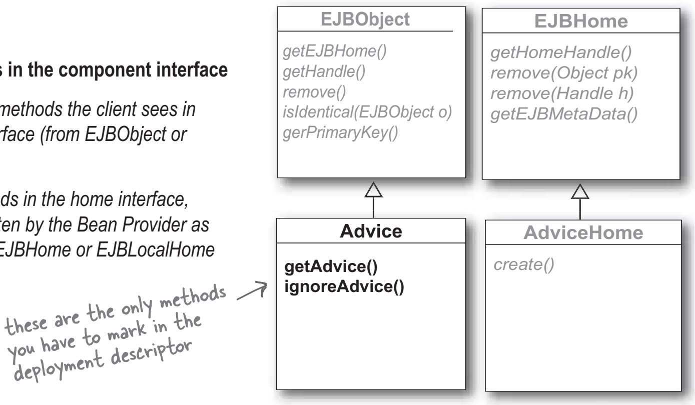
4. **Of the six transaction attributes, three of them can be dangerous, with one in particular being EXTREMELY risky. Keeping in mind that the Bean Provider is NOT the one who specifies the attributes for the bean's methods, which of the six is potentially the most dangerous?**

answer to the Attributes sharpen: three and five

These are the methods you MUST mark with an attribute (for a CMT bean)

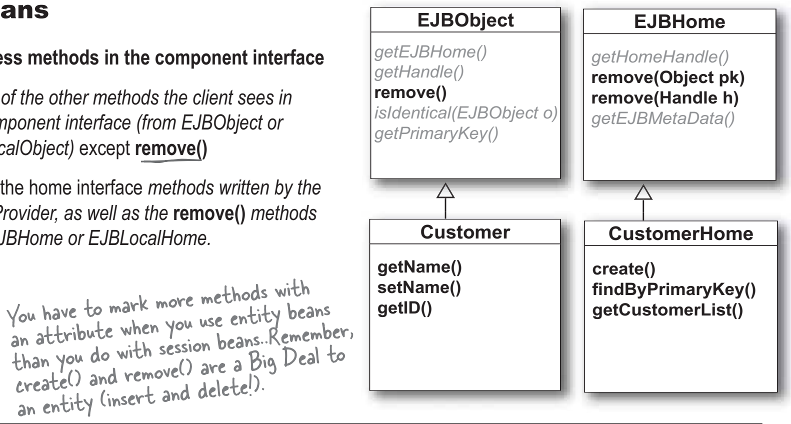
Session beans

- **Business methods in the component interface**
- *NONE of the other methods the client sees in the component interface (from EJBObject or EJBLocalObject)*
- *NONE of the methods in the home interface, including those written by the Bean Provider as well as those from EJBHome or EJBLocalHome*



Entity beans

- **Business methods in the component interface**
- *NONE of the other methods the client sees in the component interface (from EJBObject or EJBLocalObject)* except **remove()**
- ALL of the home interface *methods written by the Bean Provider, as well as the* **remove()** *methods from EJBHome or EJBLocalHome.*

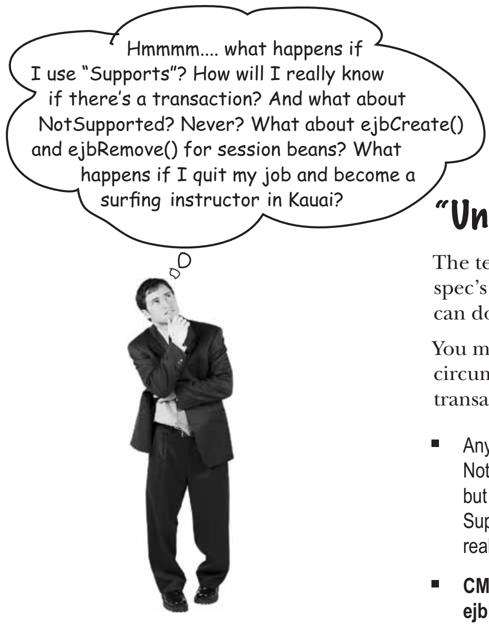


Message-driven beans

- **onMessage()**

A message-driven bean doesn't have any client interfaces.

onMessage()


Hmmmm.... what happens if I use "Supports"? How will I really know if there's a transaction? And what about NotSupported? Never? What about ejbCreate() and ejbRemove() for session beans? What happens if I quit my job and become a surfing instructor in Kauai?

"Unspecified Transaction Context"

The term "an unspecified transaction context" is the EJB spec's way of saying, "You have no clue. I (the container) can do whatever I want and you'll just have to deal!"

You must know, for the exam, the methods (and circumstances) that might be running in an "unspecified transaction context."

- Any CMT method marked **NotSupported**, **Never**, or **Supports.** NotSupported and Never are supposed to mean "no transaction", but in reality, the container can do whatever it wants. And with Supports, you never know *anyway* (which is why we think it's a really dumb attribute that nobody should ever use).
- **CMT session bean** methods **ejbCreate()** (any of them), **ejbRemove()**, **ejbPassivate()**, **ejbActivate().** The create and remove methods of a session bean are not considered part of a client's transaction (unlike the way it works with entity beans). And remember, activate and passivate will *never* be called if the session bean is in a transaction.
- **CMT message-driven bean** methods **ejbCreate()** and **ejbRemove().** Remember, for a message-driven bean, ejbCreate() and ejbRemove() are called by the container when it wants to add or remove beans from the pool. There's no calling client transaction, because a message-driven bean doesn't have a real client!

The point is... YOU DON'T KNOW!

The key point here is that you must NOT rely on any exact behavior from the container because you just don't know. As a bean developer, you'll probably never lose a moment's sleep over this. But you need to know what is and isn't guaranteed.

We think being a snowboard instructor is better than teaching surfing. Just as fun, but without all that neoprene. Or sharks.

So, is there a transaction or not? Why is it "unspecified"? Why isn't it just "definitely *no* transaction"?

The spec lets the Container do whatever it wants. The spec suggests several options, including everything from executing without any transaction at all to merging multiple calls to a resource manager together into one transaction.

Burn these in

These are all things you might be tested on. But remember, you won't be asked a simple true or false question, like, "The getRollbackOnly() method can be called from a method with a transaction attribute of NotSupported." (In which case the answer would be *false*, of course). No, you're likely to see something much more clever, like bean code plus the bean's deployment descriptor, and you have to decide if it all works together.

getRollbackOnly() MUST be called from a bean in a transaction!

You already know that getRollbackOnly() can be called only by CMT beans, and that the method exists only in EJBContext and NOT in UserTransaction. (Remember? BMT beans can call getStatus() on UserTransaction, but they can't call getRollbackOnly().)

But for getRollbackOnly() to work, the method must be in a transaction, which means you MUST use only:

**** Requires***
**** RequiresNew***
**** Mandatory***

Message-driven beans can use only TWO attributes: *Requires* and *NotSupported*.

Think about it. A message-driven bean doesn't have a calling client! The container calls the onMessage() method, so it doesn't make any sense to use the other ones. RequiresNew? That's useless because there will never be a pre-existing transaction. Mandatory? That would blow up every time (by "blow up" we mean throw an exception) because Mandatory means "Requires Pre-Existing". Never? Well, that is always the case anyway, and Supports is silly too. You can say only two things for onMessage():
YES, I want to start a transaction (Required)
or
NO, I don't want a transaction (NotSupported)

Remember: only SYSTEM exceptions cause an automatic rollback!

We'll cover this in detail in the Exceptions chapter, but for now be aware that application exceptions do NOT automatically cause a rollback. An application exception is any checked exception that is declared on a bean interface, unless it's a RemoteException (RemoteException is in a special category).

Let's say you see some code on the exam that shows the bean throwing an application exception (say, a FinderException). Do NOT say that the transaction will be rolled back! If YOU (Bean Provider) decide in your logic that the situation is too bleak to allow the transaction to continue, then YOU must call setRollbackOnly(). The container does an automatic rollback ONLY for system exceptions (like EJBException or any other runtime exception).

(We caught a BMT bean and a CMT bean arguing at the bar.)

BMT: You suck.

CMT: Oh, that's so *clever.* What a way with words. And what do you mean by that? And is it just me personally, or all CMT beans in general?

BMT: All of you. Weenies. You're not bean enough to handle your own transactions, so you leave it all up to the container. You're probably afraid of the garbage collector, too.

CMT: Weenie? Don't tell me you actually *believe* that *you're* managing your own transactions? The container is *always* in charge my friend. Same for you as it is for me.

BMT: That's not true! *I* start the transaction and *I* decide if—and when—to commit or rollback. Where's the container in all that? I mean, sure, the container has to *give* me the transaction, but after that, I'm in charge.

CMT: *No*, you're *not*.

BMT: *Yes,* I *am*.

CMT: All you're doing is demarcating the boundaries of the transaction. You get to say where it *starts* and *stops.* End of story.

BMT: Uh, you're forgetting that *I* have the power to rollback.

CMT: So do I.

BMT: *No*, you *don't*.

CMT: *Yes,* I *do.* What do you think setRollbackOnly() is for?

BMT: You can't call that. That's in the UserTransaction interface and that's off-limits to you CMT peasants.

CMT: I can't believe they let you be a bean. You *know* that EJBContext has a setRollbackOnly() method just for CMT beans.

BMT: Oh. I forgot about that. But so what? You still aren't in control of your transaction boundaries.

CMT: But what does that matter? My methods are all marked with how I want transactions to be applied, so how is that different from controlling the boundaries of a transaction?

BMT: HELLO! Your transactions must be at *least* as long as a whole method! I can scope my transactions to something more granular than the whole darn method.

CMT: But why would you ever *want* to?

BMT: You really have to ask that. OK, let me break it down so that even *you* can understand: transactions hurt concurrency. That means they hurt performance and scalability. And that means—

CMT: [interrupting] Yeah, yeah I know all that. But without transactions, you can't even run your business.

BMT: Duh. I'm not talking about *not* using transactions. I'm talking about keeping them *as short as possible*. It's just like the difference between synchronizing an entire method versus making a synchronized block.

CMT: I'm not sure that really matters, but OK. No, wait, if your methods are that big, you probably have a bad OO design anyway. So if that's the only benefit to BMT...

BMT: It's not. I can do things you can't *ever* do.

CMT: For example?

BMT: Like open a transaction in one stateful bean method and close the transaction in some other method of that bean.

CMT: Oh, yeah, like *that's* not gonna kill your performance? Don't you know how much that hurts your scalability? That prevents your stateful beans from being passivated! You risk just leaving the transaction open, like, forever. And how are you gonna even guarantee that the method that starts the transaction will be called *before* the method that ends it? Geez, you could call commit() or rollback() before you ever said begin()!

BMT: Yeah but if someone *needs* to do that, BMT is the only way to do it.

CMT: Except almost nobody ever needs to do that! Or should. OK, yes, I'll agree that if in the unbelievably and incredibly unlikely event that a developer needs to do that, you're the only way. But there's no other reason I can see to use BMT, especially since BMT defeats the whole reusable component thing. How arrogant can you be? You can *never* run in anybody else's transactions! Only your own. Talk about "doesn't play well with others..."

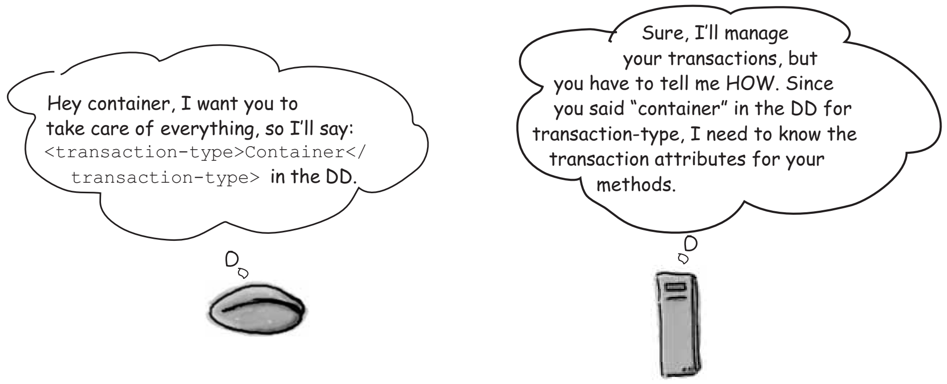
BMT: Well, OK, maybe I really am just for *special* occasions, but next time we'll have to talk about preventing poison message-driven bean messages. I'm the *only* one who can do *that*.

CMT: Yeah, well, we'll just have to see about that when we get to the MDB chapter. I'm outta here.

Marking transactions in the DD

All beans must say whether they're using bean- or container-managed transactions. For BMT beans, that's it for the DD.

But for CMT beans that's just the beginning.


Hey container, I want you to take care of everything, so I'll say: <transaction-type>Container</transaction-type> in the DD.
Sure, I'll manage your transactions, but you have to tell me HOW. Since you said "container" in the DD for transaction-type, I need to know the transaction attributes for your methods.

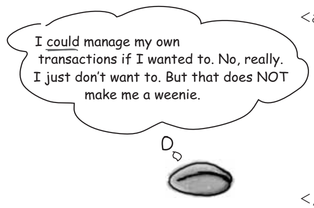
Transaction-related DD info lives in two places

1. **For both CMT and BMT beans:**
   The <transaction-type> element in the <enterprise-beans> section

```
<enterprise-beans>
   <session>
    ...
     <transaction-type>Container</transaction-type>
   </session>
</enterprise-beans>
```

for BMT, the type is Bean

2. **For CMT beans only:**
   The <container-transaction> element in the <assembly-descriptor> section


I could manage my own transactions if I wanted to. No, really. I just don't want to. But that does NOT make me a weenie.

```
<assembly-descriptor>
  <container-transaction>
     <method>
        <ejb-name>MyBean</ejb-name>
        <method-name>bar</method-name>
     </method>
     <trans-attribute>Required</trans-attribute>
  </container-transaction>
</assembly-descriptor>
```

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

DD example for CMT

In the <enterprise-beans> element

```
<enterprise-beans>

  <session>
    <display-name>AdviceBean</display-name>
    <ejb-name>AdviceBean</ejb-name>
    <home>headfirst.AdviceHome</home>
    <remote>headfirst.Advice</remote>
    <ejb-class>headfirst.AdviceBean</ejb-class>
    <session-type>Stateless</session-type>
    <transaction-type>Container</transaction-type>
    <security-identity>
      <use-caller-identity></use-caller-identity>
    </security-identity>
  </session>

</enterprise-beans>
```

for BMT you'd say (surprise!) "Bean" instead of "Container"

In the <assembly-descriptor> element

```
<assembly-descriptor>
  <container-transaction>
     <method>
        <ejb-name>MyOtherBean</ejb-name>
        <method-name>foo</method-name>
     </method>
     <trans-attribute>RequiresNew</trans-attribute>
  </container-transaction>
</assembly-descriptor>
```

there are a few variations of the <method> element (we'll look at them on the next page)

You specify attributes by listing a specific attribute (RequiresNew, Supports, etc.) and putting in all of the methods that are supposed to have that attribute. You might *think* you're supposed to first specify a method and then give the attribute for that method (because that's how it looks in this example), but that's not how it works. It'll all make sense on the next page...

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

More DD examples for CMT

Option 1: Wildcard

Use a wildcard to say that all methods in the specified bean have the attribute in the <trans-attribute> tag.

```
<container-transaction>
     <method>
        <ejb-name>BigBean</ejb-name>
        <method-name> * </method-name>
     </method>
     <trans-attribute>RequiresNew</trans-attribute>
  </container-transaction>
```

Specify a method by giving the bean name and the method name. But here, we use a wildcard (*) which means "ALL methods of BigBean have RequiresNew attribute."

Option 2: Individually-named methods

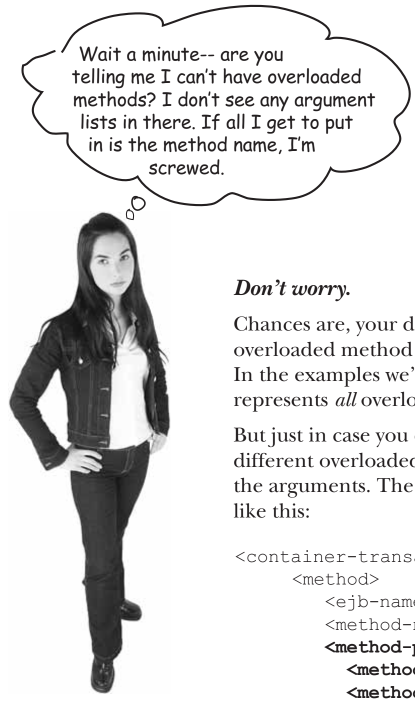
Specify each method of each CMT bean in the ejb-jar (unless you use the wildcard to indicate all methods of the specified bean class have the specified attribute).

```
<container-transaction>
     <method>
        <ejb-name>BigBean</ejb-name>
        <method-name>foo</method-name>
     </method>
     <method>
        <ejb-name>TinyBean</ejb-name>
        <method-name>go</method-name>
     </method>
     <method>
        <ejb-name>MyBean</ejb-name>
        <method-name>bar</method-name>
     </method>
     <trans-attribute>RequiresNew</trans-attribute>
</container-transaction>
<container-transaction>
     <method>
        <ejb-name>BigBean</ejb-name>
        <method-name>doStuff</method-name>
     </method>
     <trans-attribute>Required</trans-attribute>
  </container-transaction>
```

Here we have three different methods, in two different beans, that all have the RequiresNew attribute.

This is a different transaction attribute -- Requires -- so here we list the method that uses Requires. Notice that we're naming the same bean we used in the previous <container-transaction> element.

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


Wait a minute-- are you telling me I can't have overloaded methods? I don't see any argument lists in there. If all I get to put in is the method name, I'm screwed.

Don't worry.

Chances are, your design will treat all versions of an overloaded method in the same way (for transactions). In the examples we've seen so far, a method name represents *all* overloaded versions of that method.

But just in case you do need to distinguish between different overloaded methods, there is a way to specify the arguments. The optional **<method-params>** tag looks like this:

```
<container-transaction>
     <method>
        <ejb-name>ShoppingBean</ejb-name>
        <method-name>addItem</method-name>
        <method-params>
          <method-param>java.lang.String</method-param>
          <method-param>int</method-param>
        </method-params>
     </method>
     <trans-attribute>RequiresNew</trans-attribute>
</container-transaction>
<container-transaction>
     <method>
        <ejb-name>ShoppingBean</ejb-name>
        <method-name>addItem</method-name>
        <method-params>
          <method-param>int</method-param>
        </method-params>
     </method>
     <trans-attribute>Requires</trans-attribute>
</container-transaction>
```

Here we have an overloaded method, addItem, where one version takes a String and an int, and the other takes only an int. The first one uses RequiresNew and the second version uses Required. So we have to specify WHICH overloaded version we're talking about in each section. The optional <method-params> and <method-param> tags give you a way to do that.

Transaction attributes are part of application assembly, NOT bean development.

That means the attributes are specified in the assembly part of the DD rather than in the bean part.

The only thing you say about transactions in the bean part is whether the bean is using Container- or Bean-managed transactions.

Yes you DO need to know the tags for transactions.

You're not expected to memorize every tag in the whole DD, but you DO need to know how to specify transaction attributes, including the way you can use the wildcard (*) to represent all the methods of a bean, and that there IS a way to give overloaded methods different transaction attributes.

Be sure you know that transaction attributes are NOT specified in the <enterprise-beans> part of the DD, but rather in the <assembly-descriptor> section. Think about why that makes sense... a Bean Provider has to say (in the DD) whether the bean is using BMT or CMT, but that's it. The app assembler will come along later and decide exactly how transactions should be handled for this particular deployment of the bean.

Q: Can I combine the wildcard with specific method names? I mean, what happens if I want to have all the bean's methods, except one use RequiresNew, and just one method to use NotSupported?

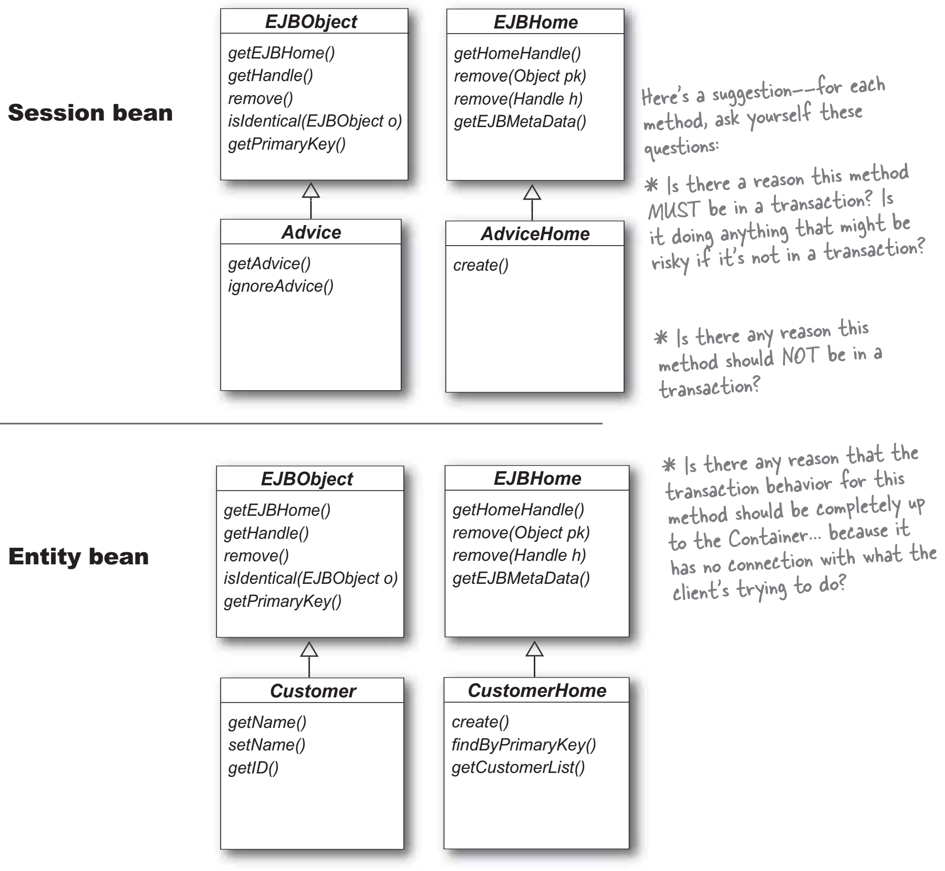
A: No problem. Using the wildcard is like saying, "All methods *not otherwise specified in the DD* should use this attribute." So go ahead and use the wildcard for the RequiresNew part of the DD, and when you get to the NotSupported attribute, give the method name.

```
<container-transaction>
  <method>
     <ejb-name>BigBean</ejb-name>
     <method-name> * </method-name>
  </method>
  <trans-attribute>RequiresNew</trans-attribute>
</container-transaction>
<container-transaction>
  <method>
     <ejb-name>BigBean</ejb-name>
     <method-name>useOldDatabase</method-name>
  </method>
  <trans-attribute>NotSupported</trans-attribute>
</container-transaction>
```

Sharpen your pencil

Which methods need transaction attributes?

This one is a combination of how much you remember, and how much you can think through what makes sense. Looking at the interfaces below, one for a session bean and one for an entity bean, figure out which methods MUST have transaction attributes when the bean is using CMT. Put a checkmark by it, circle it, run your highlighter through it. But don't try to simply remember what you've seen in this chapter... really think about it.

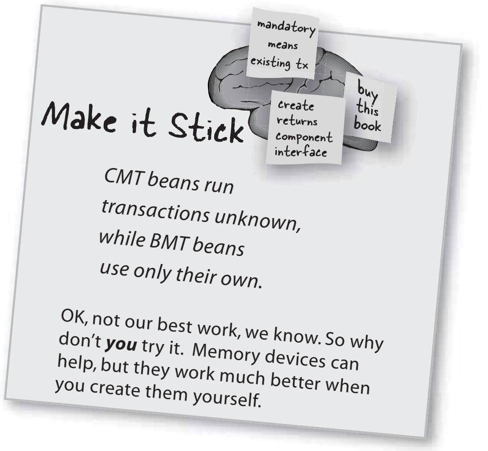


Head First EJB™ By Bert Bates, Kathy Sierra ISBN: 0596005717 Publisher: O'Reilly
Print Publication Date: 2003/10/01

Summary of Bean-managed demarcation

Bean-managed (BMT)

- Used by stateless and stateful session beans.
- Used by message-driven beans.
- Must *NOT* be used by entity beans.
- Can be used to reduce the scope of a transaction, which can help performance.
- Can be used to keep a transaction open across multiple invocations to a stateful method bean.
- Can be used by a message-driven bean to acknowledge a method even though the transaction ends in a rollback.
- Message-driven beans must complete a transaction by the end of onMessage().
- Stateless session beans must complete a transaction by the end of the business method in which the transaction was started.
- The BMT bean must not start a transaction without first ending the previous transaction. (Remember, no nested transactions in EJB!)
- The BMT bean must *not* use the getRollbackOnly() and setRollbackOnly() methods of EJBContext.
- The BMT bean can call the setRollbackOnly() method on the UserTransaction.
- The bean gets a UserTransaction from the bean's EJBContext.
- Propagation of transactions in a BMT bean are one-way: a bean-started transaction can propagate *out to other beans*, but no transaction can ever be propagated *in* to a bean using BMT.
- The UserTransaction interface does NOT have a getRollbackOnly() method, but BMT beans can call getStatus() to find out if someone in the transaction has called setRollbackOnly().
- The getStatus() method returns an int representing a constant, one of which is STATUS_MARKED_ROLLBACK.
- Session beans using BMT must not implement SessionSynchronization.


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

CMT beans run transactions unknown, while BMT beans use only their own.

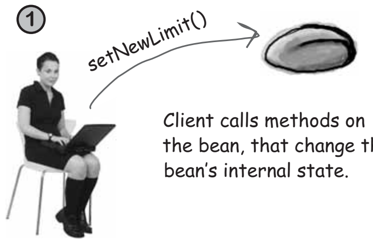
OK, not our best work, we know. So why don't ***you*** try it. Memory devices can help, but they work much better when you create them yourself.

Container-managed (CMT)

- Used by all bean types.
- Compulsory for entity beans.
- Transaction attributes are specified in the deployment descriptor.
- The six transaction attributes are: Required, RequiresNew, Supports, Mandatory, Never, Not Supported.
- Transactions can be propagated into and out of a CMT bean.
- The CMT bean must not attempt to use BMT, including getting a UserTransaction reference.
- The CMT bean must not use the setRollbackOnly method of UserTransaction. (Guess you could figure that out from the previous bullet point...)
- The CMT bean can use the getRollbackOnly and setRollbackOnly methods of EJBContext.
- CMT transactions cannot stay open across multiple method invocations from a stateful session client (BMT transactions can).
- CMT transactions are scoped at the method level. Either the whole method runs in a particular transaction context, or it doesn't.
- Calling setRollbackOnly on an EJBContext means the container must NOT commit the transaction.
- A CMT bean that calls setRollbackOnly() or getRollbackOnly() MUST have an attribute of Required, RequiresNew, or Mandatory.
- A CMT session bean must specify attributes for: all business methods in the component interface, none of the methods in the home, and none of the methods from EJBObject (or EJBLocalObject).
- All entity beans must specify attributes for: all business methods in the component interface, all the methods declared by the Bean Provider in the home interface, and any remove() method that the client can access (i.e. the ones defined in EJBHome, EJBLocalHome, or EJBObject and EJBLocalObject.
- You probably do not want to use Supports because you can't know for certain whether your bean is going to run in a transaction. If you've written calls to getRollbackOnly() or setRollbackOnly(), the bean can get an exception.
- A message-driven bean can use only two attributes: Required or NotSupported.

Entity beans have ejbLoad() to stay synchronized, even if the transaction rolls back.

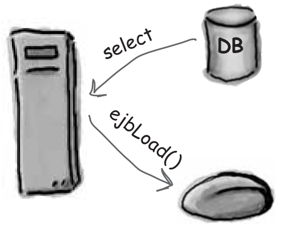
1



Client calls methods on the bean, that change the bean's internal state.

2 Bean has a problem and can't commit the transaction. But now that leaves the bean out of sync with the database. The bean's limit state is 420, but the limit in the database is 343, exactly where it was before the transaction began.

3 Container simply does a new load on the bean, to refresh it with the original data from the database.



Now everything is just like it was before. As if nothing ever happened...

4 Bean is happy, and now all of its persistent state matches the entity's data in the database.

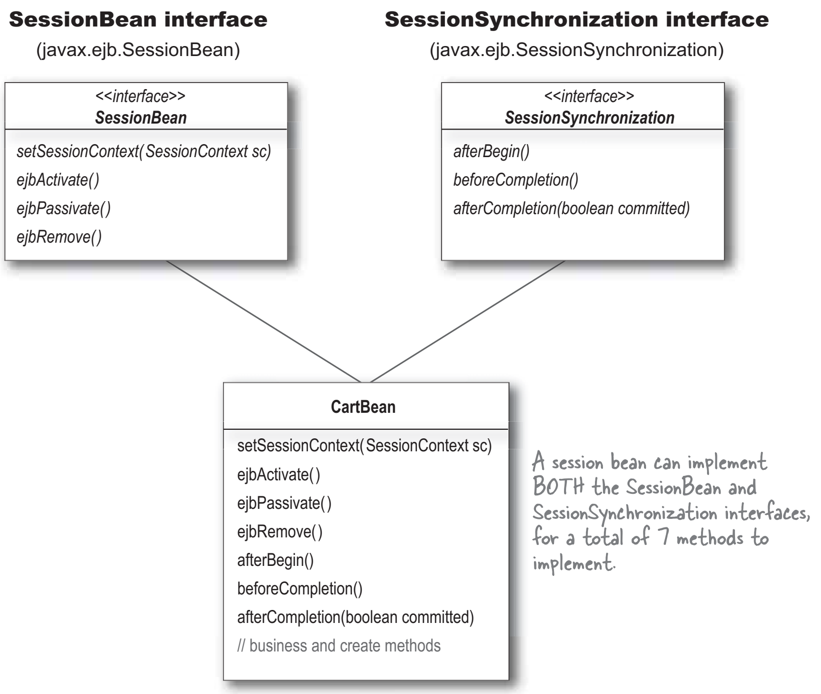
Session Synchronization

because session beans don't have ejbLoad and ejbStore

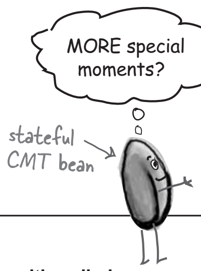
The point is this:

An entity bean has ejbLoad() and ejbStore() to tell it when to synchronize with the database. If the transaction is about to commit, ejbStore() is called to give the bean one last chance to get it's persistent state in order, ready to be written to the database. And if the transaction does *not* commit, the bean just gets another ejbLoad() to return it to its original pre-transaction state, and everything is fine.

But a session bean doesn't have that luxury. No ejbLoad()or ejbStore() to tell it when it's time to synchronize itself with a database. But if your session bean implements SessionSynchronization, you *can* give a session bean three new container callbacks, that notify the bean of three more *special moments* in the bean's transactional life: when a transaction starts, when it's about to end, and when it's over.


SessionBean interface
(javax.ejb.SessionBean)
<<interface>>
SessionBean
setSessionContext(SessionContext sc)
ejbActivate()
ejbPassivate()
ejbRemove()
SessionSynchronization interface
(javax.ejb.SessionSynchronization)
<<interface>>
SessionSynchronization
afterBegin()
beforeCompletion()
afterCompletion(boolean committed)
CartBean
setSessionContext(SessionContext sc)
ejbActivate()
ejbPassivate()
ejbRemove()
afterBegin()
beforeCompletion()
afterCompletion(boolean committed)
// business and create methods
A session bean can implement BOTH the SessionBean and SessionSynchronization interfaces, for a total of 7 methods to implement.

SessionSynchronization "special moments"



	afterBegin()	beforeCompletion()	afterCompletion()
When it's called	At the beginning of a stateful bean's transaction, BEFORE the business method that's going to run in the transaction is called.	Just before the transaction ends, AFTER the business method completes.	After the transaction has ended either in a commit or rollback. The method passes a boolean argument to tell you whether the transaction committed.
Which entity bean method it's most like	ejbLoad()	ejbStore()	Nothing in the entity bean lifecycle corresponds to this. (Because there's no need.)
What to do when you're in it	Load in data from the database, knowing that the database resource is now part of the transaction, so you can cache the data in the bean for the rest of the transaction.	Last chance to update the database before the transaction commits, and the locks on the database are released.	Find out how the transaction went, and do whatever you have to do to stay synced with the database. For example, if you were caching newly-changed data in temporary variables, you can now assign the temporary values to your permanent state variables.Or if the transaction rolled back, you might have to reset some values with data from the database.
Bean things you *can* do while you're in it	Call methods on your SessionContext: get your home, get your EJB object, get caller security info, force the transaction to rollback, check the rollback status. Access: your special JNDI context, resource managers, and other beans. ***You're in a transaction, so it's all safe!***	***You're still in a transaction...*** so you can do the same things you can do in afterBegin().	Call methods on your SessionContext: get your home, get your EJB object, get caller security info. You can't do any transaction-related things because ***you're no longer in a transaction!*** You can access ONLY your special JNDI context. It's not safe to access resource managers or other beans.

Head First EJB™ By Bert Bates, Kathy Sierra ISBN: 0596005717 Publisher: O'Reilly Prepared for Linda Martin, Safari ID: doannn16@yahoo.com
Print Publication Date: 2003/10/01 User number: 896963 Copyright 2008, Safari Books Online, LLC.

Stateless session beans can't implement SessionSynchronization!

Only stateFUL session beans can implement SessionSynchronization, because stateless session bean's aren't allowed to maintain a transaction once a method has ended.

SessionSynchronization is for CMT beans ONLY!

Does it make sense for a BMT bean to use SessionSynchronization? No. Remember, the point of SessionSynchronization is to give the bean three more "special moment" callback methods, so that the bean can find out ***when*** *a transaction starts and ends, and also* ***how*** *it ends.*

But think about it... a BMT bean knows EXACTLY when a transaction starts and ends, because the bean is the one demarcating the transaction boundaries.

It's the bean that says when the transaction starts and ends!

Nobody can end that transaction except the bean that started it, so there aren't any surprises the bean needs to be "told" about.

there are no Dumb Questions

Q: If you want synchronization so badly, why not just use an entity bean?

A: Because your bean might represent a process, and not an entity. Not everything that involves database data is an entity! If your session bean represents a process, like a shopping session, but it still uses a database, you might want to wait until the transaction is complete before updating information in the database.

On the other hand, using a session bean as as a "poor-man's entity bean" , *just for the sake of avoiding entity beans,* is silly. If you need an entity, make it an entity bean and get all the advantages the Container has to offer, like automatic synchronization with the persistent store. But if your bean is a process, but still needs to stay on top of how the transaction is going, you've got the option with SessionSynchronization (as long as it's a stateful, CMT bean.)

Pool Puzzle

Your ***job*** is to take code snippets from the pool and place them into the blank lines in the code. You will use each snippet **exactly once**.

Your ***goal*** is to make a method in a BMT bean that will create a total of five transactions. But wait, there's more! Exactly TWICE during this method, a transaction will be temporarily suspended. (We won't tell you whether it's the same transaction suspended twice, or two different transactions.) Any method with "cmt" in its name is from a CMT bean, and its transaction attribute is revealed in its name.

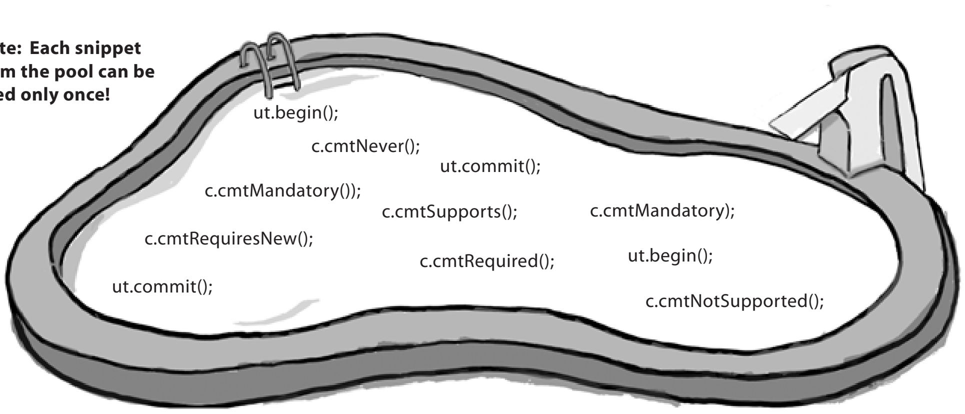
Assume that the variables "c" and "ut" have been properly initialized as instance variables.

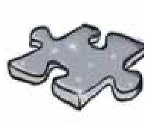
Oh yeah, you must NOT throw any transaction related exceptions!

(Answers are on the next page, so don't look, unless you have absolutely no concern for the loss of self-esteem that you'll experience by going straight for the answers.)

```
void bmtMethodWithTransactions () {

  __________________

  c.cmtRequiresNew();

  __________________

  __________________

  __________________

  __________________

  __________________

  __________________

  __________________

  __________________

  __________________

  __________________
}
```

Note: Each snippet from the pool can be used only once!





Pool Puzzle Solution

(Which of course you won't be looking at until you've completed the puzzle on the previous page.)

```
void bmtMethodWithTransactions () {
    ut.begin();                tx 1

    c.cmtRequiresNew();        tx 2,  suspend  tx 1
    c.cmtMandatory);           tx 1 (tx 2 ended when the previous method completed)
    c.cmtRequiresNew();        tx 3, suspend tx 1 (that's the second and final transaction suspension.)
    ut.commit();               now there's no tx
    c.cmtNotSupported();       still no tx
    c.cmtNever();              still no tx
    c.cmtRequired();           tx 4
    ut.begin();                tx 5 (tx 4 ended when the previous method completed)
    c.cmtSupports();           tx 5
    c.cmtMandatory());         tx 5
    ut.commit();               Done
}
```

COFFEE CRAM
Mock Exam

1 Which are true about transactions in EJB 2.0? (Choose all that apply.)

❏ A. EJB containers must support both JTA and JTS.

❏ B. EJB 2.0 supports nested transactions.

❏ C. The **`javax.transaction.UserTransaction`** API allows you to set isolation levels.

❏ D. A bean instance can run multiple transactions in parallel.

❏ E. A message-driven bean instance must complete a transaction before the 'onMessage' method returns.

2 When using BMT demarcation, which of the following must commit a transaction before the method that initiated the transaction returns? (Choose all that apply.)

❏ A. message-driven beans

❏ B. stateful session beans

❏ C. stateless session beans

❏ D. none of the above

3 Which two are true about container-managed transactions in EJB 2.0? (Choose all that apply.)

❏ A. Differentiating between overloaded methods is possible in the bean's deployment descriptor.

❏ B. Every business method in the bean class must have a transaction attribute.

❏ C. If an onMessage() method returns before committing a transaction the container will throw an exception.

❏ D. A message-driven bean with CMT demarcation must not invoke the EJBContext.getUserTransaction() method.

4 What's true when specifying transaction attributes in the deployment descriptor? (Choose all that apply.)

- ❑ A. For session beans, transaction attributes can be applied only to methods in the bean's component interface.
- ❑ B. The <method-name> tag can take a wild card.
- ❑ C. A single method can have multiple transaction attributes specified in the deployment descriptor.
- ❑ D. Transaction attributes must NOT be specified for methods in an entity bean's home interface.

5 Which transaction attributes can cause an in-progress transaction to be suspended? (Choose all that apply.)

- ❑ A. NotSupported
- ❑ B. Required
- ❑ C. Supports
- ❑ D. RequiresNew
- ❑ E. Mandatory
- ❑ F. Never

6 The use of which transaction attribute can cause a **`javax.transaction.TransactionRequiredException`** exception to be thrown?

- ❑ A. NotSupported
- ❑ B. Required
- ❑ C. Supports
- ❑ D. RequiresNew
- ❑ E. Mandatory
- ❑ F. Never

7 If a set of CMT bean methods has the following transaction attributes:

Method-1 = Supports

Method-2 = Required

Method-3 = NotSupported

Method-4 = RequiresNew

In the diagrams that follow, an arrow indicates that the method on the left calls the method on the right, and the "tx" number indicates a unique transaction. Which diagrams work with the transaction attributes listed above? (Choose all that apply.)

```
❑ A. M1 (Tx 1) –> M2 (No Tx) –> M3 (No Tx) –> M4 (Tx 2)

❑ B. M1 (No Tx) —> M2 (Tx 1)
                 \   \—–> M3 (Tx 1)
                  \——> M4 (Tx 2)

❑ C. M1 (No Tx) –> M2 (Tx 1) –> M3 (No Tx)
                   \—–> M4 (Tx 2)

❑ D. M1 (Tx 1) –> M2 (Tx 1) –> M3 (Tx 1) –> M4 (Tx 2)
```

8 Which methods of a session bean, with container-managed transaction demarcation, run in an unspecified transaction context? (Choose all that apply.)

❑ A. **`ejbActivate()`**

❑ B. **`ejbPassivate()`**

❑ C. Business method marked **`'NotSupported'`**

❑ D. **`ejbRemove()`**

❑ E. Business method marked 'RequiresNew'

9 If a session bean's business method invokes **`EJBContext.setRollbackOnly()`**, which transaction attribute settings can cause the container to throw the **`java.lang.IllegalStateException`**? (Choose all that apply.)

❑ A. NotSupported

❑ B. Required

❑ C. Supports

❑ D. RequiresNew

❑ E. Mandatory

❑ F. Never

10 Which method can be called successfully by a bean using bean-managed transaction demarcation?

- ❏ A. **`getUserTransaction()`**
- ❏ B. **`afterBegin()`**
- ❏ C. **`afterCompletion()`**
- ❏ D. **`getRollbackOnly()`**

COFFEE CRAM

Mock Exam Answers

1 Which are true about transactions in EJB 2.0? (Choose all that apply.) (spec: 340–341)

- ☐ A. EJB containers must support both JTA and JTS. – just UserTransaction from JTA
- ☐ B. EJB 2.0 supports nested transactions.
- ☐ C. The **`javax.transaction.UserTransaction`** API allows you to set isolation levels. – No
- ☐ D. A bean instance can run multiple transactions in parallel.
- ✓ E. A message-driven bean instance must complete a transaction before the 'onMessage' method returns. – Yes!

2 When using BMT demarcation, which of the following must commit a transaction before the method that initiated the transaction returns? (Choose all that apply.) (spec: 340–341)

- ✓ A. message-driven beans
- ☐ B. stateful session beans
- ✓ C. stateless session beans
- ☐ D. none of the above

3 Which two are true about container-managed transactions in EJB 2.0? (Choose all that apply.) (spec: 353–356)

- ✓ A. Differentiating between overloaded methods is possible in the bean's deployment descriptor.
- ✓ B. Every business method in the bean class must have a transaction attribute.
- ☐ C. If an onMessage() method returns before committing a transaction the container will throw an exception. – No
- ✓ D. A message-driven bean with CMT demarcation must not invoke the EJBContext.getUserTransaction() method.

4 What's true when specifying transaction attributes in the deployment descriptor? (Choose all that apply.) (spec: 351-354)

- ✓ A. For session beans, transaction attributes can be applied only to methods in the bean's component interface.
- ✓ B. The <method-name> tag can take a wild card.
- ❏ C. A single method can have multiple transaction attributes specified in the deployment descriptor.
- ❏ D. Transaction attributes must NOT be specified for methods in an entity bean's home interface. – they must be specified

5 Which transaction attributes can cause an in-progress transaction to be suspended? (Choose all that apply.) (spec: 357-359)

- ✓ A. NotSupported
- ❏ B. Required
- ❏ C. Supports
- ✓ D. RequiresNew
- ❏ E. Mandatory
- ❏ F. Never

6 The use of which transaction attribute can cause a **`javax.transaction.TransactionRequiredException`** exception to be thrown? (spec: 357-359)

- ❏ A. NotSupported
- ❏ B. Required
- ❏ C. Supports
- ❏ D. RequiresNew
- ✓ E. Mandatory
- ❏ F. Never

7 If a set of CMT bean methods has the following transaction attributes: (spec: 357–359)

Method-1 = Supports

Method-2 = Required

Method-3 = NotSupported

Method-4 = RequiresNew

In the diagrams that follow, an arrow indicates that the method on the left calls the method on the right, and the "tx" number indicates a unique transaction. Which diagrams work with the transaction attributes listed above? (Choose all that apply.)

- ❑ A. M1 (Tx 1) –> M2 (No Tx) –> M3 (No Tx) –> M4 (Tx 2)
- ❑ B. M1 (No Tx) —> M2 (Tx 1)

```
              \  \——> M3 (Tx 1)
               \——> M4 (Tx 2)
```

- ✓ C. M1 (No Tx) –> M2 (Tx 1) —> M3 (No Tx)

```
                 \——> M4 (Tx 2)
```

- ❑ D. M1 (Tx 1) –> M2 (Tx 1) –> M3 (Tx 1) –> M4 (Tx 2)

8 Which methods of a session bean, with container-managed transaction demarcation, run in an unspecified transaction context? (Choose all that apply.) (spec: 363–364, 376)

- ✓ A. **`ejbActivate()`**
- ✓ B. **`ejbPassivate()`**
- ✓ C. Business method marked **`'NotSupported'`**
- ✓ D. **`ejbRemove()`**
- ❑ E. Business method marked 'RequiresNew'

9 If a session bean's business method invokes **`EJBContext.setRollbackOnly()`**, which transaction attribute settings can cause the container to throw the **`java.lang.IllegalStateException`**? (Choose all that apply.) (spec: 360–361)

- ✓ A. NotSupported
- ❑ B. Required
- ✓ C. Supports
- ❑ D. RequiresNew
- ❑ E. Mandatory
- ✓ F. Never

10 Which method can be called successfully by a bean using bean-managed transaction demarcation? (spec: 361)

[✓ A. **`getUserTransaction()`**]

❑ B. **`afterBegin()`** } These are callbacks from

❑ C. **`afterCompletion()`** } SessionSynchronization

❑ D. **`getRollbackOnly()`** ← this is for CMT beans only