Table of Contents

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

10 exceptions in EJB

When beans go bad

Expect the unexpected. Despite your best efforts, things can go wrong. Terribly, *tragically*, wrong. You need to protect yourself. Others depend on you. You can't let your entire program collapse, just because one bean in the family throws an exception. ***The application must go on.*** You can't *prevent* tragedy, but you can *prepare* for it. You need to know what is and is *not* recoverable, and *who* is responsible when a problem occurs. Should the Bean Provider tell the Container there's still hope? Should the Container tell the client to try again? Or should the Container cut its losses, kill the bean, and let the Sys Admin sort it out later...

this is a new chapter 525

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

***exam** objectives*

Exceptions

Official:		*What it really means:*
12.1	Identify correct and incorrect statements or examples about exception handling in EJB.	You have to know that application exceptions are things the client expects, and might recover from, while system exceptions are for things the client can't recover from. You need to know the five standard EJB application exceptions, and all of the common system exceptions. You have to know exactly how the Container behaves when the bean throws an application exception vs. a system exception, and you have to know the difference between exceptions for local clients and exceptions for remote clients.
12.2	Given a list of responsibilities related to exceptions, identify those which are the Bean Provider's, and those which are the responsibility of the Container. Be prepared to recognize responsibilities for which neither the bean or Container are responsible.	You need to know that if the bean throws an application exception, the Container does not automatically rollback the transaction, and that the Container gives the exception to the client exactly as the bean threw it. With system exceptions, however, the Container always rolls back the transaction, and gives the exception to a remote client as a RemoteException, and to a local client as an EJBException. You'll need to know that if a bean wants to rollback a transaction, as part of an application exception, the bean must call setRollbackOnly(), since the rollback won't happen automatically. You need to know that the Container will log system exceptions but not application exceptions, and that the Container will kill any bean that raises a system exception.
12.3 & 12.4	Identify correct and incorrect statements or examples about application exceptions and system exceptions in entity beans, session beans, and message-driven beans. Given a particular method condition, identify the following: whether an exception will be thrown, the type of exception thrown, the Container's action, and the client's view.	If you're going to answer these questions correctly, you have to know bean details from previous chapters. For example, you have to know that a message-driven bean doesn't have a client view (i.e. client interfaces) so a message-driven bean can't declare or throw any application exceptions. And you have to know, for example, that a bean will get an IllegalStateException if it tries to call methods on its context from places where those operations aren't allowed.

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

What can go wrong?

1 In the Bean

- The business logic in a method realizes that it cannot do its job because of a *problem the client expects*. For example, a banking bean can't do a balance transfer, because the 'transfer from' account doesn't have any money!
- The *bean catches a checked exception* while running business logic. For example, the bean fails to get a JDBC connection, or a JNDI lookup throws a NamingException to the bean. This is a problem that the *bean* expects, but the *client* doesn't.
- A method in the bean (or a method the bean calls) *throws a runtime exception* (like NullPointerException) that the bean doesn't catch, and the client doesn't *expect*.

2 In the Container

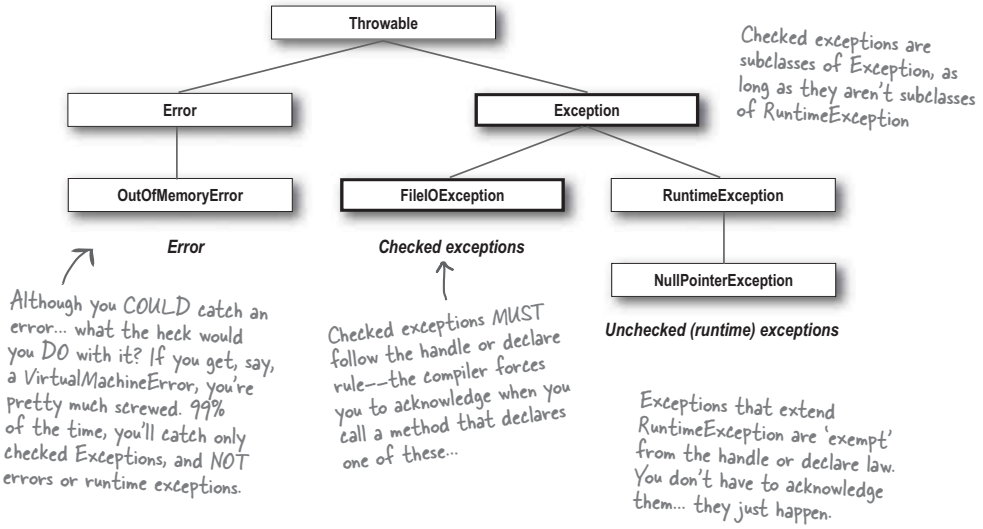
- The Container can't complete an operation for which its responsible, such as updating the state of an entity bean in the database, that causes a problem the client *expects*.
- The Container catches a checked exception thrown by the bean, that the client *expects*.
- A Container catches a runtime exception thrown by the bean, or by some other object the Container interacts with. This is a problem that the client does *not* expect.

3 In the RMI subsystem, or some other part of the communication path between client and container.

- The EJB object throws a runtime exception before it communicates with the client about the result of a client's business method call to the bean.
- The RMI subsystem can't communicate with the client.
- The client stub throws a runtime exception, or a RemoteException while trying to send a method call, or receive a return value.

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

Remember, Java exceptions can be checked or unchecked



We know that you remember all of this, but just so we're all speaking the same language here—Java has both *checked* and *unchecked* exceptions. Checked as in *compiler-checked*. If you call a method that declares a throws clause with a checked exception, you *must* reassure the compiler that you know all about this potential problem, and you're ready to take the risk. It's the ***handle or declare*** rule. You either wrap the risky method call in a try/catch, or you declare that you, too, throw the exception.

Remember, declaring an exception means *ducking* it—letting someone else in the call stack deal with it. Declaring an exception is like saying, "I don't want to catch this exception, because I think someone else can do a better job of handling it (or someone else *should* be handling it).

If you handle an exception and exit the catch block, it should make no difference whether the exception occurred or not. If it does matter, then you aren't handling the exception correctly, and you probably should have ducked it instead.

Runtime exceptions are unchecked (they matter only at *runtime*, not *compile* time). You can declare them, catch them, and throw them in code any way that you like, but the compiler won't care. But if your code causes a runtime exception, the call stack/thread of execution you're running in dies. If that's the last non-daemon thread in your program, your whole program shuts down.

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

It's all about expectations...

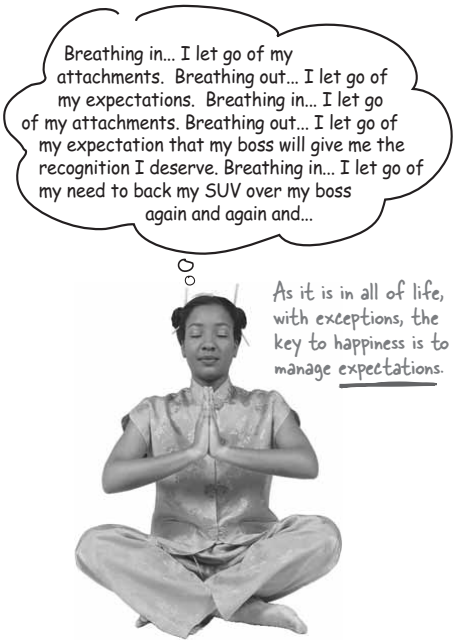
Exception handling in EJB (or Java in general) centers on expectations. You *hope* that everything works correctly. You *hope* that you don't get a runtime exception. But you take risks. So you *expect* that some things *will* go wrong. The key is in knowing what those things *are*, and what you can do to *recover*.

Most of the time, the things that can go wrong are pretty obvious. You do a JNDI lookup, but the object isn't there. You try to connect to a naming service, and can't. You try to connect to the database and can't. You try to create a new entity bean, and the insert fails because you have a duplicate primary key. And for each of those, you can write a catch block that nows how to deal with the specific problem you have. Can't connect to the naming service? Maybe you have a second naming service to try. Can't connect to the database? Maybe you can try using a local cache for now, until you can re-establish your connection. Can't do the insert? Try again, using a different primary key.

When you provide a catch block for a risky method call, you're saying that you expect something *specific* might go wrong. Almost every exception for which you provide a catch block is an exception that you expect. So you might, for example, expect a NamingException or a FinderException or a CreateException.

Wrapping a risky method call in a try/catch tells the compiler (and your design) that you expect that certain things might go wrong.

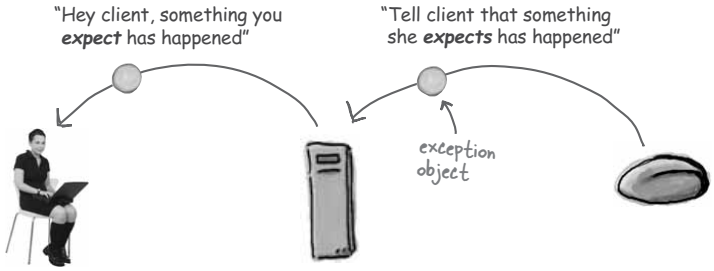
You hope that everything works correctly, but you can't guarantee it, so you have a list of catch blocks, for things you expect MIGHT go wrong...



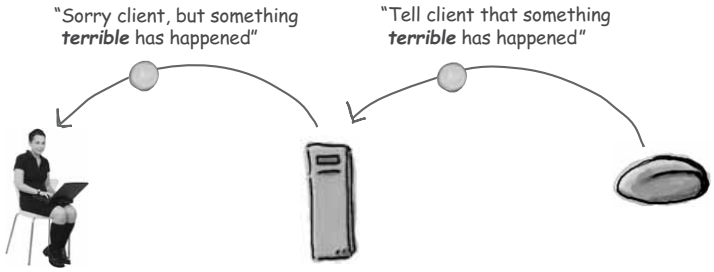
Head First EJB™ By Bert Bates, Kathy Sierra ISBN: 0596005717 Publisher: O'Reilly
Print Publication Date: 2003/10/01

Exception pathways

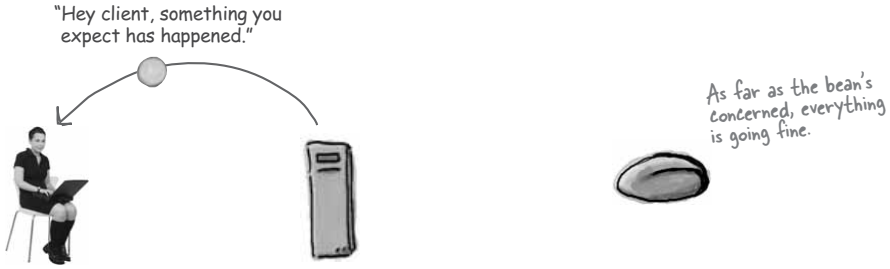
1 Bean throws something the client expects



2 Bean throws something the client does NOT expect

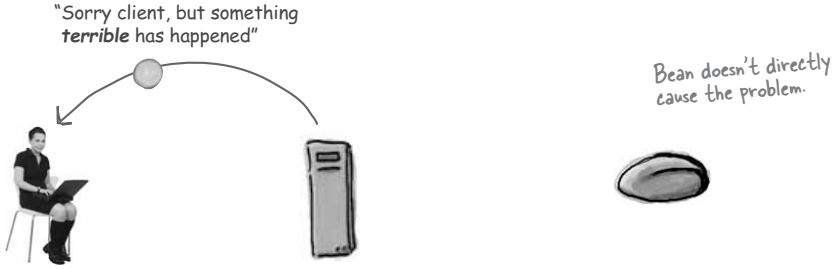


3 Container throws something the client expects

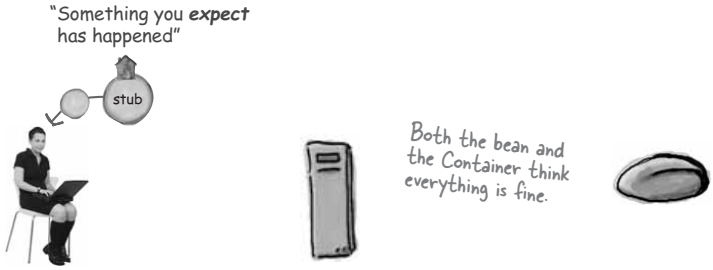


Head First EJB™ By Bert Bates, Kathy Sierra ISBN: 0596005717 Publisher: O'Reilly
Print Publication Date: 2003/10/01

④ Container throws something the client does NOT expect



⑤ The stub throws something the client expects



⑥ The stub throws something the client does NOT expect

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

In EJB, exceptions come in two flavors: application and system

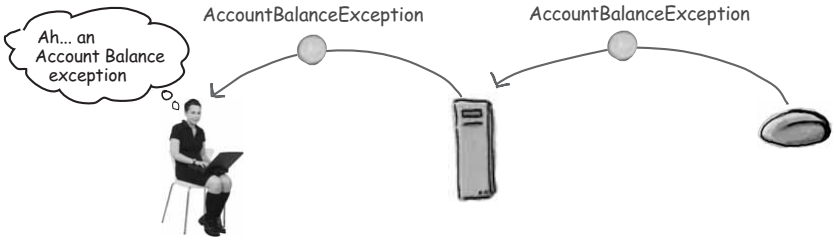
Application exceptions are things the client *expects*, which means the client has to handle them one way or another. This means that application exceptions are declared in the interfaces exposed to the client, and the client uses a try/catch when she calls the method. Since the client can *catch* the expected exception, the client might be able to *recover* and keep going. Application exceptions include things like CreateException (maybe the client-supplied arguments weren't valid), or AccountBalanceException (perhaps the client can try a different account with the next call), or ObjectNotFoundException (that entity is no longer in the database). Application exceptions include all checked exceptions thrown by a bean or a Container, except java.rmi.RemoteException. Although RemoteException is checked, and the client catches it, as far as the client is concerned, whatever caused the RemoteException on the server was *unexpected*.

System exceptions are for things the client does *not* expect and/or can't recover from. This could be virtually any runtime exception that happens on the server (from the bean or the Container or any other object on the server), or even in the client's local stub object. System exceptions are things the client really can't recover from, either because they're not recoverable (like a failure in the database or a NullPointerException) or because the client doesn't have enough information to know what to do (like with a RemoteException which—even though it *is* a checked exception the client catches—tells the client only that something unexpectedly bad happened on the server.)

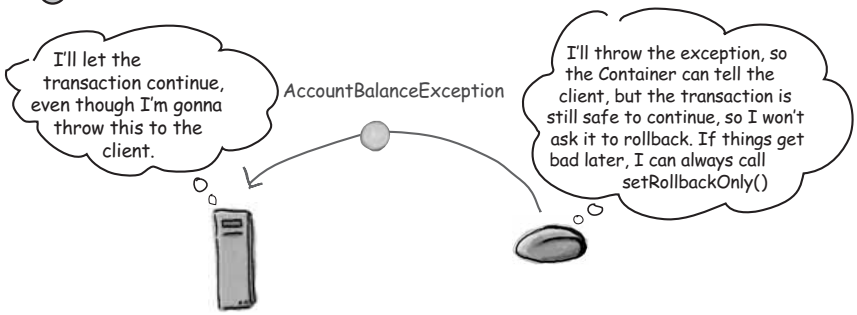
Head First EJB™ By Bert Bates, Kathy Sierra ISBN: 0596005717 Publisher: O'Reilly
Print Publication Date: 2003/10/01

With an Application Exception, the Container will...

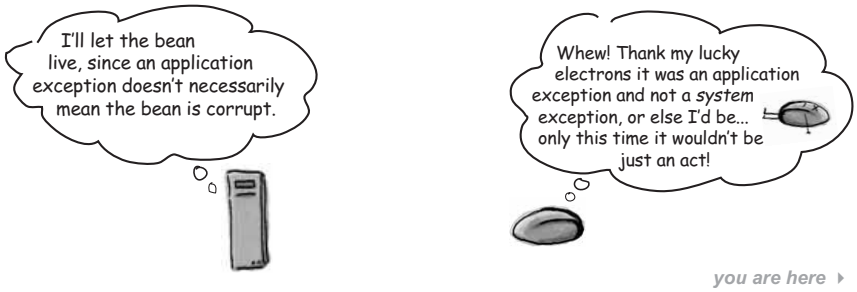
1 send it back to the client EXACTLY as it was thrown



2 NOT set the transaction to rollback



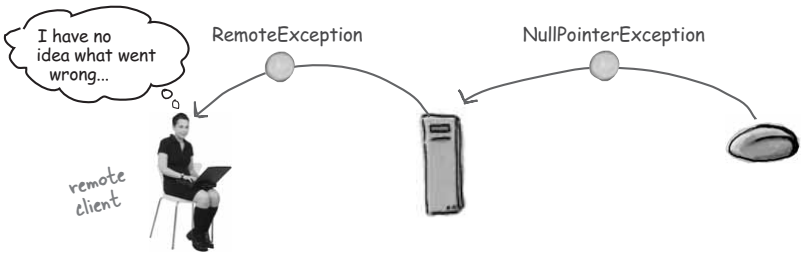
3 spare the bean's life



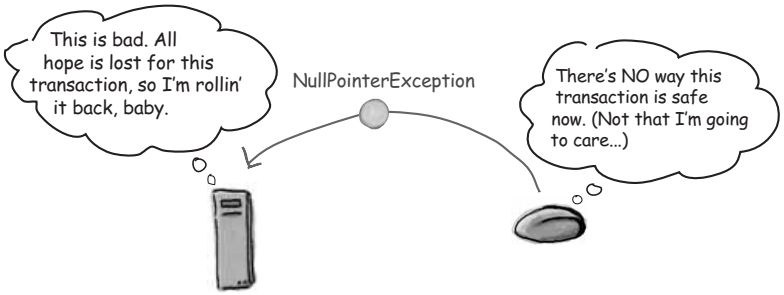
Head First EJB™ By Bert Bates, Kathy Sierra ISBN: 0596005717 Publisher: O'Reilly
Print Publication Date: 2003/10/01

With a System Exception, the Container will...

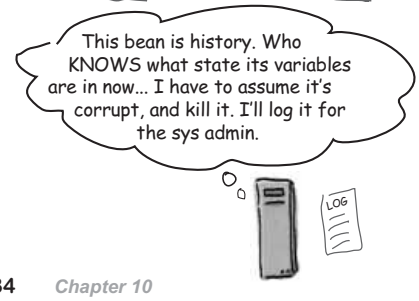
1 send it back to a Remote client as a RemoteException, or to a local client as an EJBException



2 always cause a transaction rollback



3 kill the bean, and log the exception



this time, when the bean dies, it means the bean INSTANCE is discarded and made eligible for garbage collection. If it's an entity bean, the real entity is still alive, and can be reloaded into a new bean instance.

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

exceptions in EJB

	Application Exceptions	System Exceptions
Client recovery	 client can try to recover	 client can't recover
Transaction status	 transaction proceeds (unless bean calls setRollbackOnly())	 automatic rollback
Bean instance	 instance lives	 instance dies
Logging	 NOT logged	 exception logged

Examples	`CreateException`
	`RemoveException`
	`FinderException`
	`ObjectNotFoundException`
	`DuplicateKeyException`
	
	`AccountBalanceException`
	`BadQueryArgsException`
	`BookMappingException`
	`Exception`
		`RemoteException (remote client)`
		`EJBException (local client)`
		`IllegalStateException`
		`TransactionRequiredException`
		`NoSuchObjectException`
		`ArrayIndexOutOfBoundsException`
		`NullPointerException`
		`RuntimeException`

Compiler checking and other rules	All application exceptions MUST be checked exceptions, and must NOT extend RemoteException	The only system exception that is checked is RemoteException. All others are a RuntimeException (or one of its subclasses)

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

BRAIN POWER

What, if anything, can you do if you know you want to throw an application exception to the client, but you do NOT want the transaction to commit?

Think about that for a moment.

We'll take a look at this scenario in a few pages.

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

Warning! RemoteException is checked, but not expected!

Normally when we think of expected vs. unexpected exceptions, we map it to checked vs. unchecked exceptions. A FileIOException is expected. A NullPointerException is not. Any direct subclass of Exception is expected. A subclass of RuntimeException is not. A checked exception must be handled. An unchecked exception usually won't be.

But there's one big exception to the whole exceptions and expectations thing—*java.rmi.RemoteException*.

RemoteException *is* a checked exception, of course, so the client is forced to acknowledge a RemoteException by handling it with a try/catch. But unlike the other checked exceptions a client sees in a bean's interface, RemoteException is still considered unexpected. OK, not *exactly* unexpected... but *unexpected*.

From the client's point of view, think of RemoteException a kind of runtime exception on the remote part of the application. Because that's often what it means. A NullPointerException on the server means a RemoteException to the client. A DivideByZeroException on the server means a RemoteException to the client. A ClassCastException on the server means a RemoteException to the client. Those are all unchecked exceptions on the server, but they propagate back to the client as a checked RemoteException. In other words, the *client* has to *expect* that the *server* can throw something *unexpected*.

Does this mean that every RemoteException on the client was originally triggered by a bean getting a runtime exception? No. A bean might, for example, catch a checked exception as part of its business logic, and then realize it can't recover. At that point, the bean turns what was originally a checked exception (from the bean's perspective) to an unchecked (system) exception by wrapping and rethrowing it as an *EJBException*, which ultimately shows up as a RemoteException on the remote client.

In fact, the client can get a RemoteException even *without* the server. A stub, for example, might throw a RemoteException because it can't even *reach* the server. The key here is to think of *application* exceptions as checked exceptions that the client must acknowledge *and might recover from*, and system exceptions as all *other* exceptions, including RemoteException (and its subclasses).

Application exceptions are always compiler-checked exceptions, except for RemoteException.

Think of RemoteException as "a runtime exception on the server", even though RemoteException is a checked exception to the client.

In other words, a remote client must EXPECT that the server can throw something UNEXPECTED.

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

declaring exceptions

A Remote entity bean home interface declares application exceptions and one system exception (RemoteException)

```
package headfirst;

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

public interface CustomerHome extends EJBHome {

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

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

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

application exception

system exception

something the client expects, and can potentially recover from (an application exception)

something unexpected happened on the server, so the client can't recover.

A local entity bean home interface declares only application exceptions

```
package headfirst;

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

public interface CustomerHomeLocal extends EJBLocalHome {

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

   public Customer findByPrimaryKey(String key) throws FinderException;

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

application exception

application exception

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

RemoteException goes to remote clients
EJBException goes to local clients

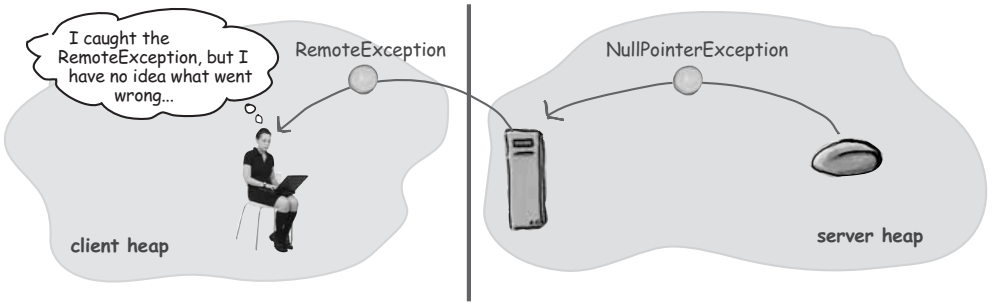
When something unexpected happens on the server, the Container tells the client by throwing either a ***RemoteException*** or an ***EJBException***.

RemoteException is for remote clients only, and even though it *is* a checked exception, it's telling the client that something *unexpected* happened. Something from which the client can't recover.

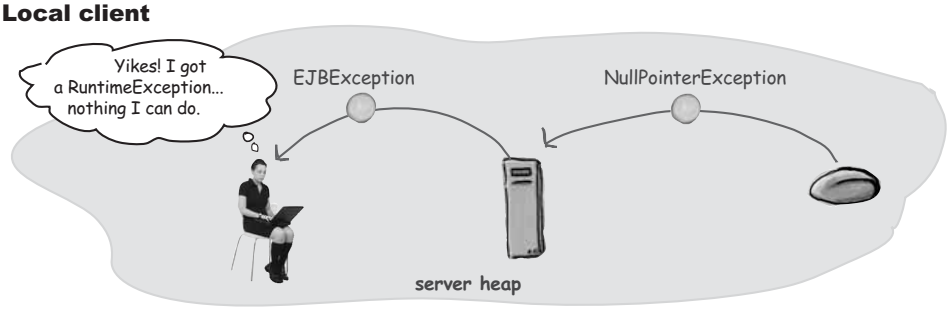
EJBException is for local clients only, and it's *unchecked*. In other words, EJBException is a subclass of RuntimeException. To the client, getting an EJBException isn't much different from getting, say, an ArrayIndexOutOfBoundsException. It means something unexpected went wrong, and there's nothing you can do to recover. The only difference is that the client does have to *catch* the RemoteException, but once he catches it, the client usually won't be able to tell what happened (at least not in any *recoverable* way).

When something unexpected happens on the server, a local client gets a RuntimeException (EJBException) but a remote client gets a compiler-checked RemoteException

Remote client



Local client



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

question on exceptions

there are no Dumb Questions

Q: In the picture, you show the client thinking, "I have no idea what happened" when she gets a RemoteException. But when I get a RemoteException in my shell terminal, it usually tells me what happened with a message like, "A NullPointerException occurred on the server" or something like that...

A: And at runtime that helps you *how*? Sure, it helps you a TON at development and testing time, but when your application is actually running, your client code probably isn't going to parse the stack trace. When we say that the client has no idea what happened, we mean that the client wasn't able to catch something specific—something *recoverable*.

Contrast that with something like CreateException, where the client can catch the exception, and from the catch block try the create again, possibly with different arguments.

It's tempting to think of application exceptions vs. system exceptions as simply checked vs. unchecked.

And in fact, from the client's perspective, the only system exceptions that are NOT unchecked (runtime) exceptions are RemoteExceptions. So for local clients, ALL system exceptions are unchecked exceptions (subclasses of RuntimeException) while all application exceptions are checked (subclasses of Exception, but not RuntimeException). For remote clients, all application exceptions are checked, but there is also one checked exception that's considered a system exception. From the client's point of view, you can think of RemoteException as being a wrapper for an unchecked runtime exception that happens on the server. (Although from the bean's point of view, the original exception might have been a checked exception... but one that the bean knew it couldn't recover from.)

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

Bean Provider's responsibilities

1 If your business logic catches (or creates) an application exception, throw it to the Container as the application exception

In bean code, if you catch an application exception thrown by other code (or you create the exception as part of your business logic), throw it to the Container exactly as-is. For example, if your bean code gets a FinderException while trying to look up another bean (in other words, while calling a finder method on another bean's home interface), give the FinderException to the Container exactly as you got it. That means either declaring it (ducking it and letting it propagate back down the stack to the Container):

```
public void someBusinessMethod() throws FinderException {
    CustomerHome home = null;
    try {
        InitialContext ic = new InitialContext();
        Object o = ic.lookup("Customers");
        home = (CustomerHome) PortableRemoteObject.narrow(o, CustomerHome.class);
    } catch(NamingException ne) {
          // deal with NamingException
    }
    try {
        Customer cust = home.findByPrimaryKey("42");
          // more stuff
    } catch(RemoteException re) {
         // deal with RemoteException
    }
}
```

if you intend to throw it, you have to declare it (just regular Java law)

We catch a NamingException and RemoteException, but NOT the FinderException, so the FinderException propagates back to the Container automatically

Or catching it and rethrowing it as-as:

```
public void someOtherBusinessMethod() throws FinderException {
    CustomerHome home = null;
    try {
        InitialContext ic = new InitialContext();
        Object o = ic.lookup("Customers");
        home = (CustomerHome) PortableRemoteObject.narrow(o, CustomerHome.class);
    } catch(NamingException ne) {
          // deal with naming exception
    }
    try {
        Customer cust = home.findByPrimaryKey("42");
          // more stuff
    } catch(RemoteException re) {
         // deal with it
    } catch(FinderException fe) {
         if (! recoverable()) {
            throw fe;
         } else { //other stuff}
    }
}
```

still have to declare it, of course

We caught it, but if we can't recover, we rethrow it exactly as we got it

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

Bean Provider's responsibilities

② If you catch an application exception, and find that you can't continue the transaction, call setRollbackOnly() before throwing the exception to the Container.

Remember, the Container won't rollback a transaction just because there's an application exception. With an application exception, the Container assumes that the whole thing might be recoverable, and that the transaction can continue. Unless you find out, in your business logic, that committing would be a Really Bad Idea (it often *is*). What's your option? Force the Container to rollback the transaction using setRollbackOnly() on your EJBContext (for a CMT bean) or on your UserTransaction (for a BMT bean).

```
public void someOtherBusinessMethod() throws FinderException {
    CustomerHome home = null;
    try {
        InitialContext ic = new InitialContext();
        Object o = ic.lookup("Customers");
        home = (CustomerHome) PortableRemoteObject.narrow(o, CustomerHome.class);
    } catch(NamingException ne) {
          // deal with naming exception
    }
    try {
        Customer cust = home.findByPrimaryKey("42");
          // more stuff
    } catch(RemoteException re) {
         // deal with it
    } catch(FinderException fe) {
         if (! recoverable()) {
            context.setRollbackOnly();
            throw fe;
         } else { //other stuff}
    }
}
```

We caught it, but we know that we can't recover AND we can't let the transaction commit!!

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

Bean Provider's responsibilities

3 If your business logic catches an exception the client is not expecting (in other words, not declared in your client view) wrap it and rethrow it as an EJBException.

How much should the client know about your internal behavior? Little or nothing, right? For example, is it any of the client's business that you're using JDBC to get your work done? And that you might, as part of your business logic, catch an SQLException? Throwing an EJBException is the programmer's way of telling the Container, "I've lost control."

Legal, but a bad idea:

```
public interface BadService extend EJBLocalObject {
   public void someMethod() throws SQLException;
}
```

Do you really think the client should KNOW that you're using SQL?

```
public class BadServiceBean implements SessionBean {
   public void someMethod() throws SQLException {
     // do stuff that might cause an SQLException
   }
    // other bean methods
}
```

The way you SHOULD do it:

```
public interface GoodService extend EJBLocalObject {
   public void someMethod();
}


public class GoodServiceBean implements SessionBean {

public void someMethod() {
   try {
      // do stuff
   } catch (SQLException se) {
        // if we can't recover
      throw new EJBException(se);
   }
}
```

EJBException is a runtime exception, so you don't have to declare it.

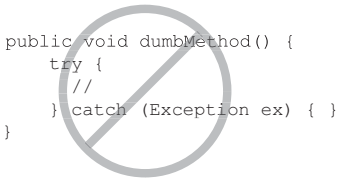
wrap and throw as an EJBException

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

Bean Provider's responsibilities

④ If your business logic encounters a runtime exception, let it propagate to the Container. Don't try to catch it!

Just don't do it. Don't use a try/catch to catch, say, Exception. That would mean you'd catch everything, and the worst thing you can do is to *eat* runtime exceptions without letting them pass back up to the Container. If you do need to catch a runtime exception, but then find that you can't recover, you can simply throw it as-is. But whatever you do, don't do this:



```
public void dumbMethod() {
    try {
      //
    } catch (Exception ex) { }
}
```

Really stupid idea. Don't catch all Exceptions (or even worse -- all Throwables!) Catch only what you need for your business logic, and let the other runtime exceptions propagate back down the call stack to the Container.

⑤ If your business logic generates an application exception, you must have declared the exception in both your client interface AND your bean class.

Remember, just because you declare an exception in an interface doesn't mean you have to declare the exception in your implementation of the method. That's true regardless of whether you implement the method in the formal Java way (because your class says, "extends ThisInterface"). But if there's a chance that you'll throw an application exception, either because you have a try/catch in your code, or because your own business logic can create one, you must declare the exception in your bean class.

You have to declare it, if you might throw it.

```
public void withdraw(double d) throws AccountBalanceException {
    if ((balance - d) < 0) {
       throw new AccountBalanceException(overdrawnMsg);
    } else {
       balance -= d;
    }
  }
}

public abstract void withdraw(double d) throws AccountBalanceException;
```


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

Bean Provider's responsibilities

6 If you create your own application exceptions, they must extend (directly or indirectly) Exception, but not RuntimeException or RemoteException

Although you're encouraged to use, or extend, the standard EJB exceptions (we'll get to those in a minute), when your design calls for your own custom exceptions, you must make them checked exceptions. That means they must extend java.lang.Exception (or one of its subclasses) as long as it does not extend java.lang.RuntimeException. The other restriction is that application exceptions must not extend java.rmi.RemoteException (directly or indirectly). Remember, an application exception is any checked exception declared in the client view, except RemoteException.

```
public void withdraw(double d) throws AccountBalanceException {
    if ((balance - d) < 0) {
       throw new AccountBalanceException(overdrawnMsg);
    } else {balance -= d;}
  }
}
class AccountBalanceException extends Exception {
    AccountBalanceException(String s) {
       super(s);
    }
}
```

there are no Dumb Questions

Q: I just realized something... the container callbacks declared in the SessionBean and EntityBean interfaces don't declare checked exceptions, right? So, doesn't this mean that you can't throw an application exception from, say, ejbActivate()?

A: That's right! You can throw only *unchecked* exceptions from a container callback that is not part of your client view. This is just plain old Java... *you can't throw a checked exception from a method that doesn't declare the exception*, and since those interfaces don't declare any checked exceptions that you can use, you're stuck with runtime exceptions. The only thing you should throw from one of the container callbacks of SessionBean, EntityBean, or MessageDrivenBean, is an EJBException.

Q: You said the interfaces don't declare any checked exceptions THAT YOU CAN USE. What does that mean? What's an example of a declared checked exception that you *can't* use?

A: If you go to the J2EE API (we'll wait, while you do that... still waiting... waiting... waiting), you can see that all of the methods declare a RemoteException! Yes, a Big Fat Checked exception. (They also declare EJBException, as a nice gesture, but not a requirement since EJBException is a subclass of RuntimeException). Does this mean you can throw a RemoteException yourself? Like, if you caught one while your bean is being a client to another bean, for example? NO! NO! A 1024 times NO!

In the days of steam-driven containers, the EJB 1.0 spec allowed *you* to throw RemoteExceptions, from your bean. Those days are over, and EJB 2.0 doesn"t allow it. So unless you're a poor soul tasked with legacy bean maintenance, you should avert your gaze when you look at the API docs, and pretend you never saw that RemoteException...

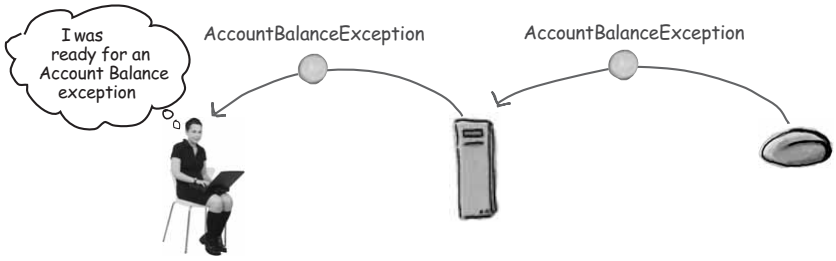
Head First EJB™ By Bert Bates, Kathy Sierra ISBN: 0596005717 Publisher: O'Reilly
Print Publication Date: 2003/10/01

The Container's responsibilities

1 If the bean throws an application exception, send it back to the client EXACTLY as it was thrown, and do NOT rollback the transaction.

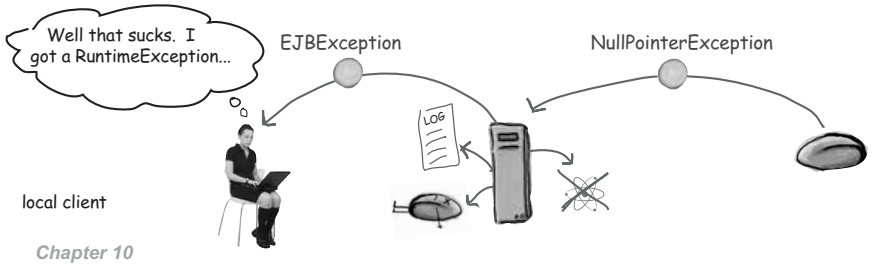
If the bean throws a CreateException, send that exception to the client. If the bean throws a FinderException, send that exception to the client. If the bean throws an AccountBalanceException, send that exception to the client.

It makes no difference whether the exception is one of the standard EJB exceptions from the java.ejb package, or one that the Bean Provider defined.



2 If the bean throws a system exception (including EJBException or any runtime exception)

- Throw a ***RemoteException*** if the client is ***Remote***
- Throw an ***EJBException*** if the client is ***local***
- Log the exception
- Rollback the transaction
- Discard the bean instance (assume it's toast)



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

Think about the client for a moment. If the client gets a RemoteException, does the client know for certain that the business method completed?

Does the client know for certain that the transaction was rolled back?

Is there any way the client might be able to find out?
What if the client is another bean?

There's nothing on the exam about non-J2EE client strategies for coping with transaction failures.

But as a developer, you need to think about what might happen on the client side, especially for situations in which you don't know whether a transaction succeeded. If your client is a bean, or another part of your J2EE application, you're in good shape. The J2EE client can usually find out the status of the transaction. But a non-J2EE client will need some other mechanism, or at least a way to guarantee that if he attempts the same transaction again, after it already succeeded (but he doesn't know), that the second (or any subsequent) attempt won't cause problems. When you have an operation that won't cause problems if attempted after its already succeeded, that means your operation is 'idempotent'. And though we really don't like the sound of that word, it might be critical to the integrity of your app.

From the list of possible options, select what you, as a Bean Provider, should do in each of the following scenarios. Assume that they all take place within a business method of a session bean.

Options (you may use an option more than once)

A. Throw an EJBException

B. Throw a RemoteException

C. Invoke setRollbackOnly()

D. Allow the exception to propagate (in other words, duck it).

Scenarios

You catch a checked exception in your ejbActivate method. The method is not in a transaction.

A DivideByZero exception occurs as your business logic is running. You do not have a try/catch for this.

You throw a CreateException from your ejbCreate() method, and you realize that you probably cannot safely complete your transaction.

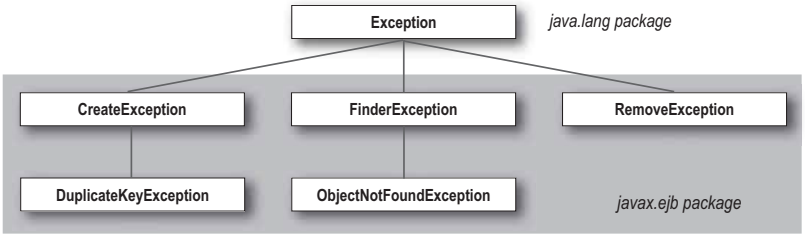
You catch a checked exception in a business method, and realize that your bean is probably corrupt.

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

Application exceptions

The five standard EJB application exceptions

The javax.ejb package has five standard application exceptions used by the EJB container, but you can use them as well, either directly or as superclasses to your own custom exceptions.



1 CreateException

The Container throws this from, you guessed it, a create method, if something goes wrong during creation. This includes scenarios in which the bean code throws CreateException itself, while running a create method. (Although if the bean's gonna throw it, the bean's gotta declare it in the bean class, not just the home interface)

2 DuplicateKeyException

The Container throws this to the client from an entity bean create method (see how it's a subclass of CreateException?) if the business logic related to the create asks the Container to insert a new entity using a primary key that has already been assigned to another entity.

3 FinderException

The Container throws this to the client from an entity bean's finder method, to tell the client that something went wrong during the finder. The bean might have thrown it (BMP beans only), but for CMP beans, only the Container can throw this method.

4 ObjectNotFoundException

The Container throws this to the client ONLY during single-entity finder methods, when there's no entity in the database matching the primary key parameter of the finder method.

5 RemoveException

The Container throws this to the client from a session or entity bean when something goes wrong in a remove method. But... the bean provider can throw this exception if, for example, he doesn't want to let clients remove entity beans.

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

The five standard application exceptions from the client's point of view

Two of the five are more specific—DuplicateKeyException and ObjectNotFoundException. If the client gets an ObjectNotFoundException, the client has more information than if he gets a more abstract FinderException. And if the client gets a DuplicateKeyException, he knows a lot more about what went wrong than if he gets a generic CreateException.

CreateException

The client does *not* know for certain whether the bean was actually created. The Container might have had a problem *after* the transaction committed.

DuplicateKeyException

If the client catches a DuplicateKeyException, she can be 100% certain that the bean was not created.

FinderException

The client does *not* know whether a matching entity (or entities, for multiple-entity finders) exists in the database. The Container might throw a FinderException because of something that went wrong before it was able to look in the datbase.

ObjectNotFoundException

If the client catches an ObjectNotFoundException, she can be 100% certain that there was no match for this primary key in the database. Remember, ObjectNotFoundException is for single-entity finders only, so a client will never get this for a multi-entity finder.

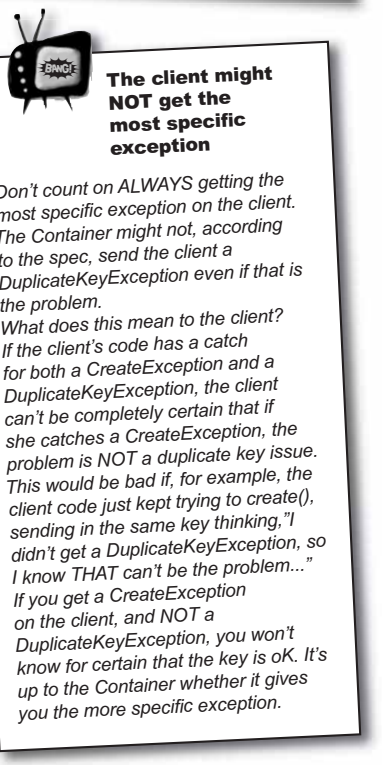
RemoveException

The client does *not* know for certain whether the bean was actually removed. The bean provider might simply have rejected the client's request, as in, "You can ask all day long, but there is no WAY that I'm going to remove that entity from the database." Or the entity might have been removed from the database, but then something else went wrong that triggered the RemoveException.

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

Stateless session bean clients will NEVER get a RemoveException!

If you see a scenario where the client gets a RemoveException, you can rule out stateless session beans. Think about it... a stateless session bean's removal is NEVER tied to a client, so there's nobody to give the exception to. You can throw a RemoveException from your ejbRemove() method, but the client will never see it if the bean is a stateless session bean.



The client might NOT get the most specific exception

Don't count on ALWAYS getting the most specific exception on the client. The Container might not, according to the spec, send the client a DuplicateKeyException even if that is the problem.
What does this mean to the client? If the client's code has a catch for both a CreateException and a DuplicateKeyException, the client can't be completely certain that if she catches a CreateException, the problem is NOT a duplicate key issue. This would be bad if, for example, the client code just kept trying to create(), sending in the same key thinking,"I didn't get a DuplicateKeyException, so I know THAT can't be the problem..." If you get a CreateException on the client, and NOT a DuplicateKeyException, you won't know for certain that the key is oK. It's up to the Container whether it gives you the more specific exception.

there are no Dumb Questions

Q: If a RemoveException makes no sense for a stateless session bean, why are you allowed to throw it (assuming you declare it on your bean's remove() method)?

A: Remember, there's nothing in the bean class that can tell you whether the bean is definitely state*less*. You *can* tell if a bean is state*ful*, of course, because any bean with overloaded creates (or it's only create has args) must be stateful. But even if a session bean has only a single no-arg create method, you still don't know whether that bean is going to be marked stateless or stateful at deploytime.

This would all be different if there were a separate interface for StatelessSessionBean and StatefulSessionBean, that your bean had to implement (instead of just SessionBean), but that's not how it works.

Q: If the Bean Provider doesn't want to let entity bean clients remove a bean, why does he have a remove method?

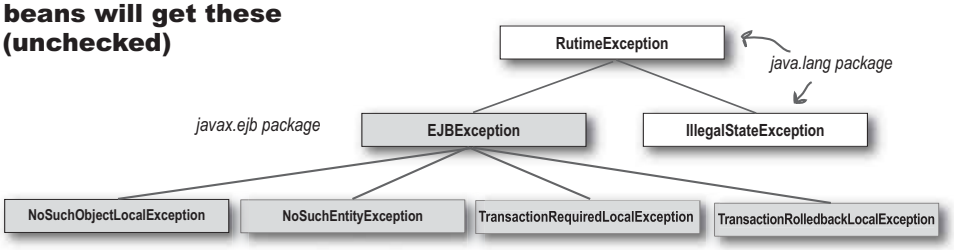
A: Aren't you forgetting something obvious here? Where do the remove() methods *live*? How are they exposed to the client? That's right. In the client interfaces. Remember, there are three remove() methods available to a remote entity bean client (the no-arg in the EJBObject interface, and the remove that takes a key and the remove that takes a handle in the EJBHome interface). A local entity bean client has two remove methods—the one in EJBObject, and the one in the home that takes a primary key. So there's nothing you can do to hide the remove() methods from a client, but if you don't support it from your bean, throw a RemoveException from your bean's ejbRemove() method, and the Container won't go through with the remove. This is your way of telling the Container,"Don't do it!! I don't care what the client says, don't you dare go through with the remove!"

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

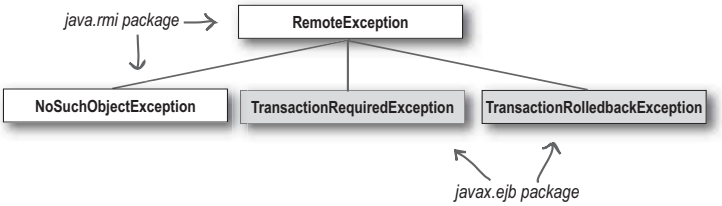
Common system exceptions

Besides the standard application exceptions for EJB, there are several important system exceptions. A few (including EJBException) are part of the J2EE API, but two of the most likely system exceptions in EJB include java.rmi.NoSuchObjectException and java.lang.IllegalStateException.

Local clients and beans will get these (unchecked)



Only Remote clients will get these (checked)



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

Common system exceptions

1 IllegalStateException

The Container throws this to a bean if the bean calls a method on its context that isn't allowed at that time. For example, a session bean can't ask its context for a reference to its EJB object while in the setSessionContext() method. It's too early. A bean can also get an IllegalStateException if it calls a transaction method like setRollbackOnly() or getRollbackOnly(), when there's no transaction!

2 EJBException

The *bean* throws this to tell the Container a system exception has occurred (which forces the Container to rollback the transaction, log the exception, kill the bean, and give a local client the EJBException and a remote client a RemoteException). But the Container can also throw this for a variety of other reasons we'll look at in a minute.

3 NoSuchObjectLocalException

This exception is kind of a local companion to java.rmi.NoSuchObjectException. The Container throws it to the client when the client invokes a method on a local home or component interface, but there's no underlying bean to support the object. This can happen, for example, if the bean has already been removed (either through a previous client call to remove(), or because the Container killed the bean due to an exception, stateful bean timeout, or to reduce the size of the pool).

4 NoSuchEntityException

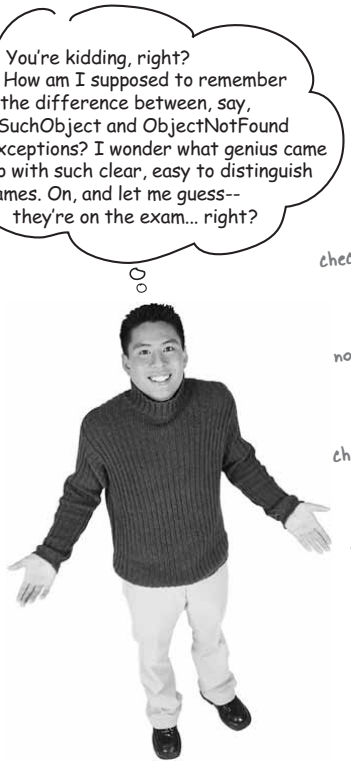
You probably won't see or use this exception much, especially with CMP, but it's for you to throw from your bean code when you want to tell the Container that the entity you're trying to access is no longer in the database (perhaps because it was removed by an admin application).

5 TransactionRequiredLocalException / TransactionRequiredException

The Container throws this to the caller when the called method has a transaction attribute of mandatory, but there's no transaction context coming in with the call. In other words, it's mandatory that the caller invokes the method within an existing transaction, and if there isn't one, the appropriate TransactionRequired exception is thrown (depending on whether the client is local or remote)

6 TransactionRolledbackException / TransactionRolledbackLocalException

The Container throws this to the caller when the Container can't commit the transaction, and the caller invoked the method within an existing transaction context. But the Container will not throw this exception if the failure to commit is because the bean explicitly called the setRollbackOnly() method. In that case, the Container will rollback the transaction and pass the business method result back to the client (unless the bean also throws an application exception).



checked
java.rmi. NoSuchObjectException
the only one that's NOT in javax.ejb is in java.rmi. It means the client has a stub, but the Remote object is gone.

not checked
javax.ejb.NoSuchObjectLocalException
this was added to EJB when local interfaces came in version 2.0. It means the client has an invalid EJB object reference.

checked
javax.ejb.ObjectNotFoundException
this is ONLY for finder methods! (single-entity finders)

not checked
javax.ejb.NoSuchEntityException
clients will never see this, and you'll probably never use it... just don't confuse it with any of the others!

OK, we agree. The names feel a little, shall we say, arbitrary. But remember, the NoSuchObjectException came first. It was designed for the scenario where the client has a stub to a Remote object, and for whatever reason the Remote object is no longer usable. Might have been a server crash. Might be that the service itself has been undeployed. Doesn't matter. As far as the client's concerned, the phone has been disconnected over at the server side.

NoSuchObject*Local*Exception was added when local client views were added in EJB 2.0. The concept is slightly different, because the local client is using not a stub, but a real Java object reference. But from the client's perspective, it really doesn't matter whether it's local or Remote—a NoSuchObject<whatever> exception means you can't use your EJB object reference to get to a bean! You have to go back through the home and start over.

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

***confusingly-named** exceptions*

Object**NotFound**Exception is a **Finder**Exception

NoSuchObectException is when the client still has a stub to a Remote object, but the object has been **removed** (or is no longer valid for any other reason.)

NoSuchObject*Local*Exception is when the local client has a reference to an EJB object for a bean that's been **removed** (or is no longer valid for any other reason).

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

BULLET POINTS

- Exceptions in Java can be checked or unchecked. Checked exceptions extend java.lang.Exception, but not java.lang.RuntimeException
- Exceptions in EJB can be thrown from the bean to the Container, and from the Container to the bean.
- EJB exceptions come in two types: *application* and *system*.
- *Application* exceptions are checked exceptions that the client is expecting, and might be able to recover from. This includes all checked exceptions *except* java.rmi.RemoteException and its subclasses.
- *System* exceptions are all other exceptions, and include all runtime exceptions, plus RemoteException.
- For the client, a RemoteException can be treated as though a runtime exception (i.e. something unexpected) happened on the server.
- When the Container gets a system exception, it will rollback the transaction, log the exception, and throw away the bean.
- When the Container gets an application exception, it will send it to the client exactly as it was received. The transaction will not automatically rollback, and the bean's life will be spared.
- The Bean Provider should throw application exceptions to the Container as-is, so that the Container can pass them on to the client. If the bean's transaction is possibly corrupt, the Bean Provider should call setRollbackOnly() before throwing application exception, so that the Container will not commit the transaction.
- If the Bean Provider catches an exception from which the bean cannot recover, he should wrap and re-throw the exception as an EJBException (a runtime exception, so he didn't need to declare it).
- There are five standard EJB application exceptions: CreateException, DuplicateKeyException (extends CreateException), FinderException, ObjectNotFoundException (extends FinderException), and RemoveException
- Common system exceptions include EJBException, IllegalStateException, TransactionRequiredLocalException, and NoSuchObjectLocalException.
- ObjectNotFoundException is for when single-entity finder methods cannot find an entity in the persistent store, that matches the primary key argument to the finder method.
- NoSuchObjectException is an RMI exception for when a Remote client has a stub to a Remote object that's been removed, or is invalid for some other reason.
- NoSuchObjectLocalException is a runtime exception for when a local client has a reference to an EJB object that's no longer valid (most likely because the bean has already been removed).

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

Scenarios: what do you think happens?

Technically, this section is like a big stealth Sharpen Your Pencil exercise. Only we just told you, so maybe not so stealthy. Sure, we could just tell you everything one fact after another... but you'll have a much better chance at remembering it if you work it out for yourself. We summarize everything near the end of the chapter, but do NOT jump there now! Even if you're someone who never does the exercises, do this scenario walk-through. You'll think of us fondly when you're holding your lovely lapel pin that you get from passing the certification exam.

1. A message-driven bean's onMessage() method catches an application exception. Can it re-throw the application exception to the Container?

   (Hint: what does a Container usually do when it gets an application exception from a bean? Would the Container be able to do that in this scenario?)

2. A session bean using CMT has a method marked with the NotSupported transaction attribute. While the method is running, the bean calls setRollbackOnly() on its context. Will this cause an exception? What kind?

   (Hint: think about what setRollbackOnly() does, and what state the bean has to be in.)

3. A message-driven bean, in the onMessage() method, calls getCallerPrincipal(). What happens?

   (Hint: what's getCallerPrincipal() used for? Does that make sense here?)

4. A session bean using CMT has a method marked with the Mandatory transaction attribute. The client calling the method is not in a transaction. What happens?

   (Hint: Think about the names of the common system exceptions. Is there one that makes sense here?)

5. A bean realizes it can't commit a transaction, but it doesn't want the client to get an exception. What can the bean do?

6. A bean wants the client to get an application exception, but the bean still wants the transaction to commit. What should the bean do?

Match the scenarios with the exception(s) that might occur with that scenario. Don't turn the page!! The answers are just a page away.

Scenarios

A CMT bean calls context.getUserTransaction().

Client calls remove() on a bean that's already been removed.

Client calls remove() on a stateful bean that is still in an open transaction.

A session bean calls getPrimaryKey() on its context.

A CMT bean calls getRollbackOnly(), from a method marked NotSupported.

Client calls a method on a Remote CMT bean, and the method is marked Mandatory. The caller does not have a transaction context in place when the call comes in.

A message-driven bean calls isCallerInRole() on its context, from within the onMessage() method.

Client calls the home remove method on the LOCAL home interface of a session bean

A stateless session bean calls getCallerPrincipal() on its context, during the setSessionContext method.

Client Foo calls a method on a remote stateful bean, while that same bean is already executing a method for client Bar.

Client calls getPrimaryKey() on the local component interface of a session bean.

A message-driven bean catches a NamingException, from which it can't recover. Which exception (if any) can the bean throw to tell the Container?

A session bean wants the Container to know that the transaction should be rolled back and the bean should be killed.

Client calls findByPrimaryKey("23"), where there is no entity with a primary key of "23".

Although the bean is fine, the Container has a system exception it wants to throw to a local client.

Transactions

IllegalStateException

EJBException

CreateException

RemoveException

ObjectNotFoundException

NoSuchObjectException

RemoteException

TransactionRequiredException

NotSupportedException

RuntimeException

NoSuchObjectLocalException

DuplicateKeyException

SystemException

NoSuchEntityException

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

Scenario Summary

Transaction Scenarios

A CMT bean calls context.getUserTransaction().

The bean gets an IllegalStateException. (Only BMT beans can get a UserTransaction)

Client calls remove() on a stateful bean that is still in an open transaction.

The client gets a RemoveException. You can't remove a stateful bean while its in a transaction. (Which is just one of a gazillion reasons why it's a Bad Idea to leave a transaction open across multiple client invocations. In other words, if you begin a BMT transaction in a method, you should end it in that method!)

A CMT bean calls getRollbackOnly(), from a method marked NotSupported.

The bean gets an IllegalStateException. You must be in a transaction when you call getRollbackOnly() or setRollbackOnly(). That means you can't call them within a method marked NotSupported, Never, or Supports (session and entity beans) or NotSupported (message-driven beans—remember, message-driven beans can't use Never or Supports, because they don't make sense given that transactions can never propagate into a message-driven bean.)

Client calls a method on a Remote CMT bean, and the method is marked Mandatory. The caller does not have a transaction context in place when the call comes in.

The client gets a TransactionRequiredException (a local client would get TransactionRequiredLocalException).

A session bean wants the Container to know that the transaction should be rolled back and the bean should be killed.

The bean should throw an EJBException. The Container takes over and does its normal Container thing for system exceptions—rollback the transaction, log the exception, kill the bean, and throw a RemoteException to a Remote client or the EJBException to a local client.

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

Scenario Summary

Client Scenarios

Client calls remove() on a bean that's already been removed.

You might be tempted to say RemoveException, but that's not it! Remember, remove() is just another method in the bean's interface, and if you call it on a removed bean, you'll get the same exception you'd see if you called any other business method on a removed bean—remote clients get RemoteException, and local clients get EJBException.

Client calls the home remove method on the LOCAL home interface of a session bean.

The client gets a RemoveException. Why? Because the only remove() method in a session bean's local home is the one that takes a primary key, and that can never work. Remote clients would also get a RemoveException.

Client Foo calls a method on a remote stateful bean, while that same bean is already executing a method for client Bar.

Client Foo gets a RemoteException (if client Foo had been local, he'd get an EJBException). A session bean handle only one client at a time!

Client calls getPrimaryKey() on the local component interface of a session bean.

This is just like the one where the client calls remove() on a local bean's home. The client gets an EJBException (if the client were local, he'd get a RemoteException). The point is? Session beans don't have primary keys! Any method you call related to the primary key of a bean will fail if that bean is a session bean.

Client calls findByPrimaryKey("23"), where there is no entity with a primary key of "23".

The client gets an ObjectNotFoundException, a subclass of FinderException.

Although the bean is fine, the Container has a system exception it wants to throw to a local client.

The client gets an EJBException.

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

Scenario Summary

Bean Scenarios

A session bean calls getPrimaryKey() on its context.

The bean gets an IllegalStateException. You know why. Session beans and primary keys don't go together...

A message-driven bean calls isCallerInRole() on its context, from within the onMessage() method.

Think about it. Does a message-driven bean have a calling client (we don't really count the Container as a 'client', although it is calling the bean's methods). If there's no client, then WHOSE security information would the bean get? The bean gets an IllegalStateException just for being clueless enough to even try.

A stateless session bean calls getCallerPrincipal() on its context, during the setSessionContext method.

The bean gets an IllegalStateException, because setSessionContext() is too early in the bean's life to get client information. In fact, a stateless bean can get client security information ONLY while running a business method of the component interface. And even if the bean were stateFUL, it would still be too early for client information, although a stateful bean (but not stateless) could call getCallerPrincipal() and isCallerInRole() from within ejbCreate().

A message-driven bean catches a NamingException, from which it can't recover. Which exception (if any) can the bean throw to tell the Container?

The bean should throw an EJBException. Remember, message-driven beans don't have clients. They don't have client interfaces. So there's no place to declare application exceptions. That means your bean can't throw anything but system exceptions. The only exception the message-driven bean should ever throw is EJBException. It can never throw application exceptions, and it should let other system exceptions propagate to the Container.

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

Fill this chart in to describe the differences between remote and local clients. You'll want to use this as a cheat sheet before the exam, so don't screw it up.

Some things might be the same for both remote and local clients, but we aren't telling.

	Remote clients	Local clients
How system exceptions in the bean are delivered to the client.		
The exception for when the client calls a method on a bean that's been removed.		
The exception for when the client calls a method marked Mandatory, without a transaction context in place.		
The exception for when the client starts a transaction, and the Container has to roll it back.		
The exception for when the client calls getPrimaryKey() on the component interface of a session bean.		
The exception for when the client calls a method in a session bean and the bean is already executing a method for another client.		
The exception for when the client calls a remove() method on a stateful session bean that's still in a transaction.		

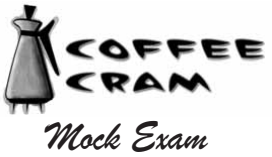
Head First EJB™ By Bert Bates, Kathy Sierra ISBN: 0596005717 Publisher: O'Reilly
Print Publication Date: 2003/10/01

exception *exercise*

Deja vu? Yes, you've seen this. Near the beginning of this chapter. Except it was all filled in. Now it's your turn. Bonus points if you use drawings along with your words.

	Application Exceptions	System Exceptions
Client recovery		
Transaction status		
Bean instance		
Logging		
Examples		
Compiler checking and other rules		

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



1 What is true when an entity bean's client receives a javax.ejb.EJBException? (Choose all that apply.)

- ❑ A. The client must be remote.
- ❑ B. The client must be local.
- ❑ C. A client will never receive this exception.
- ❑ D. The client must handle or declare this exception.
- ❑ E. This exception can only occur if the client is in a transaction.

2 Which scenario will cause a java.rmi.NoSuchObjectException to be thrown? (Choose all that apply.)

- ❑ A. A remote client invokes a method on a stateful session bean which has been removed.
- ❑ B. A remote client invokes a method on an entity bean which has been removed.
- ❑ C. A remote client invokes a finder method with invalid arguments.
- ❑ D. The container invokes an ejbPassivate() method on a bean that is not ready to be serialized.

3 Which is a subclass of java.lang.RuntimeException? (Choose all that apply.)

- ❑ A. `javax.ejb.EJBException`
- ❑ B. `javax.ejb.RemoveException`
- ❑ C. `javax.ejb.CreateException`
- ❑ D. `javax.ejb.NoSuchEntityException`
- ❑ E. `java.rmi.RemoteException`
- ❑ F. `javax.ejb.ObjectNotFoundException`
- ❑ G. `java.rmi.NoSuchObjectException`

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

4 Which of the following are EJB 2.0 specification guidelines regarding system exceptions? (Choose all that apply.)

- ❑ A. Bean methods should catch RuntimeException exceptions.
- ❑ B. Bean methods should wrap unrecoverable checked exceptions in a **`javax.ejb.EJBException`** exception.
- ❑ C. For remote clients, bean methods should wrap unrecoverable checked exceptions in a **`java.rmi.RemoteException`**.
- ❑ D. If a CMT bean throws a system exception, the transaction will still commit unless the bean invokes **`setRollbackOnly`**.

5 If a business method of a CMT demarcated bean throws a system exception, in which case will the container always throw a **`javax.transaction.TransactionRolledbackException`**? (Choose all that apply.)

- ❑ A. If the method's transaction attribute is marked 'RequiresNew'.
- ❑ B. If the method's transaction attribute is marked 'Mandatory'.
- ❑ C. If the method's transaction attribute is marked 'Never'.
- ❑ D. If the method's transaction attribute is marked 'NotSupported'.

6 Which is a subclass of javax.ejb.FinderException? (Choose all that apply.)

- ❑ A. CreateException
- ❑ B. NoSuchEntityException
- ❑ C. RemoveException
- ❑ D. DuplicateKeyException
- ❑ E. ObjectNotFoundException

7 From which methods can MDBs with CMT demarcation throw application exceptions? (Choose all that apply.)

- ❑ A. **`onMessage()`**
- ❑ B. **`ejbCreate()`**
- ❑ C. **`ejbRemove()`**
- ❑ D. **`getUserTransaction()`**
- ❑ E. **`setMessageDrivenContext()`**
- ❑ F. none of the above

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

8 What's true for a local client that receives an exception from an EJB invocation? (Choose all that apply.)

- ❑ A. The exception might be from the java.rmi package.
- ❑ B. The exception might be an application exception.
- ❑ C. The exception might be a system exception.
- ❑ D. None of the above.

9 Which action(s) will the container take if a message-driven bean with BMT demarcation throws a system exception? (Choose all that apply.)

- ❑ A. Log the exception.
- ❑ B. Discard the instance.
- ❑ C. Mark the transaction for rollback.
- ❑ D. Commit the transaction unless the bean has invoked `setRollbackOnly()`.
- ❑ E. None of the above

10 From which types of beans can clients receive system exceptions? (Choose all that apply.)

- ❑ A. Session beans with CMT demarcation.
- ❑ B. Session beans with BMT demarcation.
- ❑ C. Message-driven beans with CMT demarcation.
- ❑ D. Message-driven beans with BMT demarcation.
- ❑ E. Entity beans with CMT demarcation.

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

Mock Exam Answers

---

1 What is true when an entity bean's client receives a javax.ejb.EJBException? (Choose all that apply.) (spec: 374)

- ❑ A. The client must be remote. – Remote clients get RemoteException
- ☑ B. The client must be local.
- ❑ C. A client will never receive this exception.
- ❑ D. The client must handle or declare this exception.
- ❑ E. This exception can only occur if the client is in a transaction.

---

2 Which scenario will cause a java.rmi.NoSuchObjectException to be thrown? (Choose all that apply.) (spec: 374)

- ☑ A. A remote client invokes a method on a stateful session bean which has been removed.
- ☑ B. A remote client invokes a method on an entity bean which has been removed.
- ❑ C. A remote client invokes a finder method with invalid arguments.
- ❑ D. The container invokes an ejbPassivate() method on a bean that is not ready to be serialized.

– client gets FinderException or ObjectNotFoundException (subclass of FinderException)

---

3 Which is a subclass of java.lang.RuntimeException? (Choose all that apply.)

- ☑ A. **`javax.ejb.EJBException`**
- ❑ B. **`javax.ejb.RemoveException`**
- ❑ C. **`javax.ejb.CreateException`**
- ☑ D. **`javax.ejb.NoSuchEntityException`**
- ❑ E. **`java.rmi.RemoteException`**
- ❑ F. **`javax.ejb.ObjectNotFoundException`**
- ❑ G. **`java.rmi.NoSuchObjectException`**

(API docs)

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

4 Which of the following are EJB 2.0 specification guidelines regarding system exceptions? (Choose all that apply.) (spec: 373–374)

- ❑ A. Bean methods should catch RuntimeException exceptions.
- ☑ B. Bean methods should wrap unrecoverable checked exceptions in a **`javax.ejb.EJBException`** exception.
- ❑ C. For remote clients, bean methods should wrap unrecoverable checked exceptions in a **`java.rmi.RemoteException`**. — No, the Container will do this
- ❑ D. If a CMT bean throws a system exception, the transaction will still commit unless the bean invokes **`setRollbackOnly`**. — system exceptions always cause a tx rollback

5 If a business method of a CMT demarcated bean throws a system exception, in which case will the container always throw a **`javax.transaction.TransactionRolledbackException`**? (Choose all that apply.) (spec: 375–376)

- ❑ A. If the method's transaction attribute is marked 'RequiresNew'.
- ☑ B. If the method's transaction attribute is marked 'Mandatory'.
- ❑ C. If the method's transaction attribute is marked 'Never'.
- ❑ D. If the method's transaction attribute is marked 'NotSupported'.

The others are wrong because they indicate that the caller's tx was not the active tx when the exception occurred. There's no need to tell a caller (through an exception) unless it's the caller's transaction that is rolled back!

6 Which is a subclass of javax.ejb.FinderException? (Choose all that apply.) (spec: 263–264)

- ❑ A. CreateException
- ❑ B. NoSuchEntityException
- ❑ C. RemoveException
- ❑ D. DuplicateKeyException — subclass of CreateException
- ☑ E. ObjectNotFoundException

7 From which methods can MDBs with CMT demarcation throw application exceptions? (Choose all that apply.) (spec: 377)

- ❑ A. **`onMessage()`**
- ❑ B. **`ejbCreate()`**
- ❑ C. **`ejbRemove()`**
- ❑ D. **`getUserTransaction()`**
- ❑ E. **`setMessageDrivenContext()`**
- ☑ F. none of the above

— MDBs cannot declare / throw any application exceptions... who would catch them? there's no client.

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

8 What's true for a local client that receives an exception from an EJB invocation? (Choose all that apply.) (spec: 381–382)

- ❑ A. The exception might be from the java.rmi package.
- ☑ B. The exception might be an application exception.
- ☑ C. The exception might be a system exception.
- ❑ D. None of the above.

9 Which action(s) will the container take if a message-driven bean with BMT demarcation throws a system exception? (Choose all that apply.) (spec: 378)

- ☑ A. Log the exception.
- ☑ B. Discard the instance.
- ☑ C. Mark the transaction for rollback.
- ❑ D. Commit the transaction unless the bean has invoked `setRollbackOnly()`.
- ❑ E. None of the above

10 From which types of beans can clients receive system exceptions? (Choose all that apply.) (spec: 377)

- ☑ A. Session beans with CMT demarcation.
- ☑ B. Session beans with BMT demarcation.
- ❑ C. Message-driven beans with CMT demarcation.
- ❑ D. Message-driven beans with BMT demarcation. – MDBs have no clients!
- ☑ E. Entity beans with CMT demarcation.

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