

Architecture of Enterprise Applications 7

Transaction

Haopeng Chen

***RE*liable, *IN*elligent and *Sc*alable Systems Group (**REINS**)**

Shanghai Jiao Tong University

Shanghai, China

<http://reins.se.sjtu.edu.cn/~chenhp>

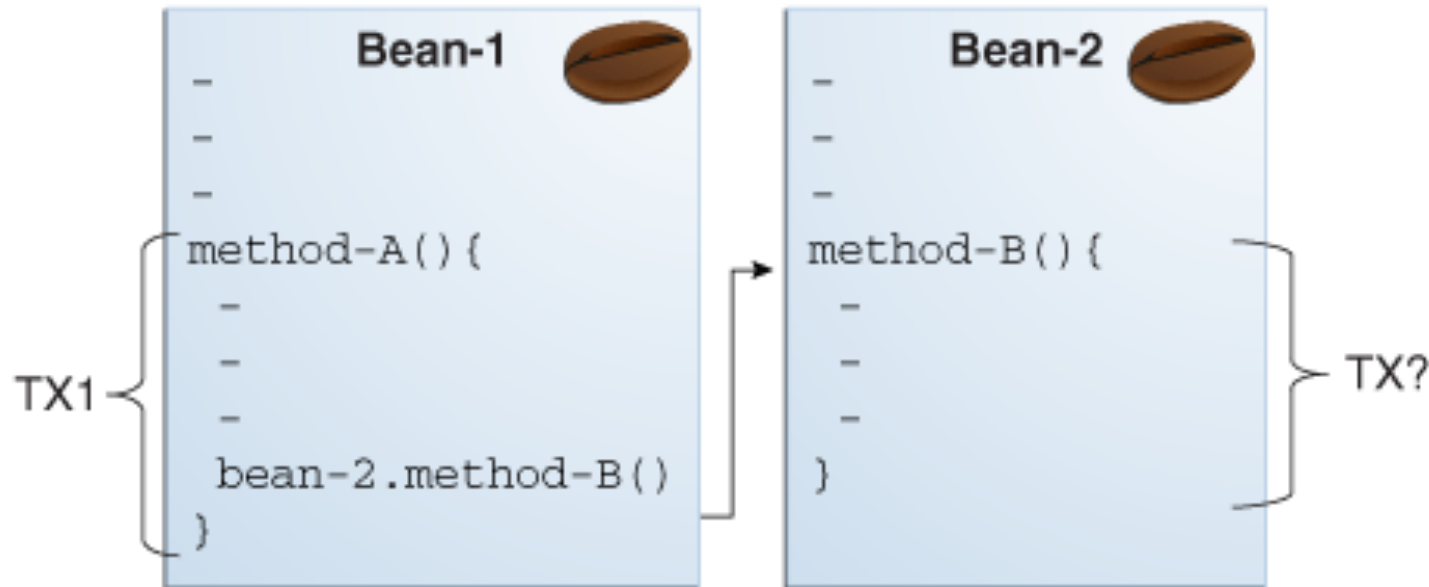
e-mail: chen-hp@sjtu.edu.cn

- Transactions in Java EE Applications
 - What Is a Transaction?
 - Container-Managed Transactions
 - Bean-Managed Transactions
 - Transaction Timeouts
 - Isolation and Database Locking
 - Updating Multiple Databases

- In a Java EE application, a transaction
 - is a series of actions that must all complete successfully, or else all the changes in each action are backed out. Transactions end in either a commit or a rollback.
- The Java Transaction API (JTA) allows applications
 - to access transactions in a manner that is independent of specific implementations
- The JTA defines the **UserTransaction** interface that applications use to start, commit, or roll back transactions.
 - Application components get a **UserTransaction** object through a JNDI lookup by using the name **java:comp/UserTransaction** or by requesting injection of a **UserTransaction** object.

- Pseudocode:
begin transaction
 debit checking account
 credit savings account
 update history log
commit transaction
- A **transaction** is often defined as an indivisible unit of work
 - Either all or none of the three steps must complete.
- A transaction can end in two ways:
 - with a commit or with a rollback.

- In an enterprise bean with **container-managed transaction demarcation**,
 - the EJB container sets the boundaries of the transactions.

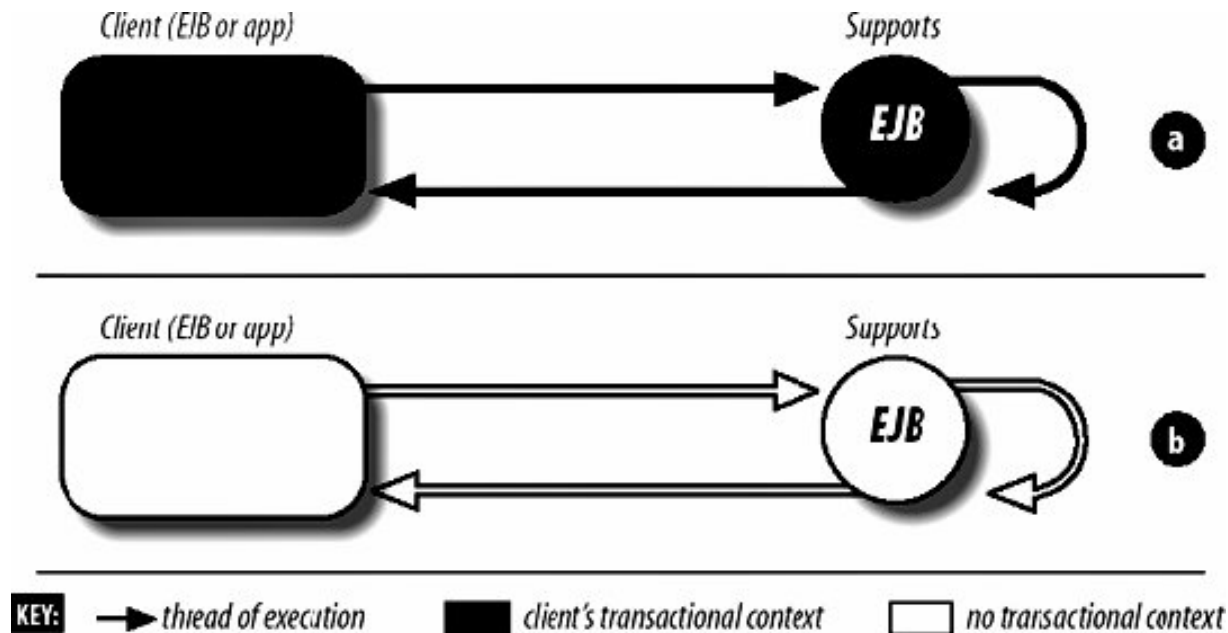


- NotSupported
 - Invoking a method on an EJB with this transaction attribute suspends the transaction until the method is completed.



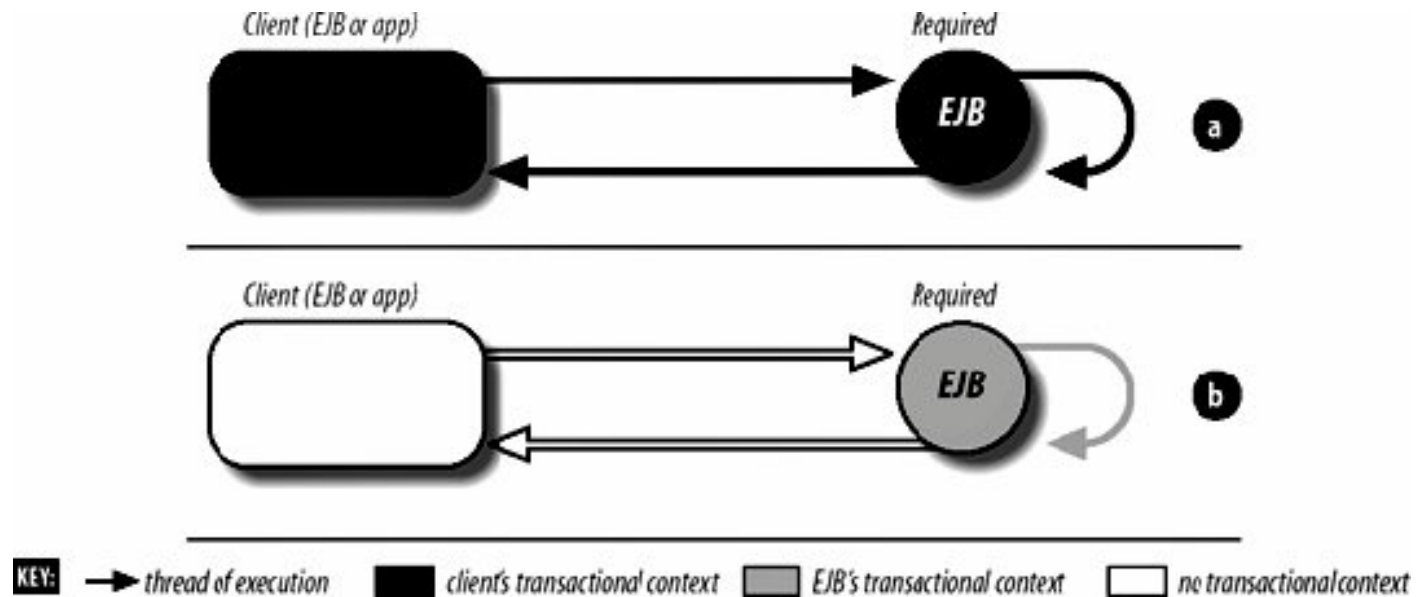
Transaction attributes defined

- Supports
 - This attribute means that the enterprise bean method will be included in the transaction scope if it is invoked within a transaction.

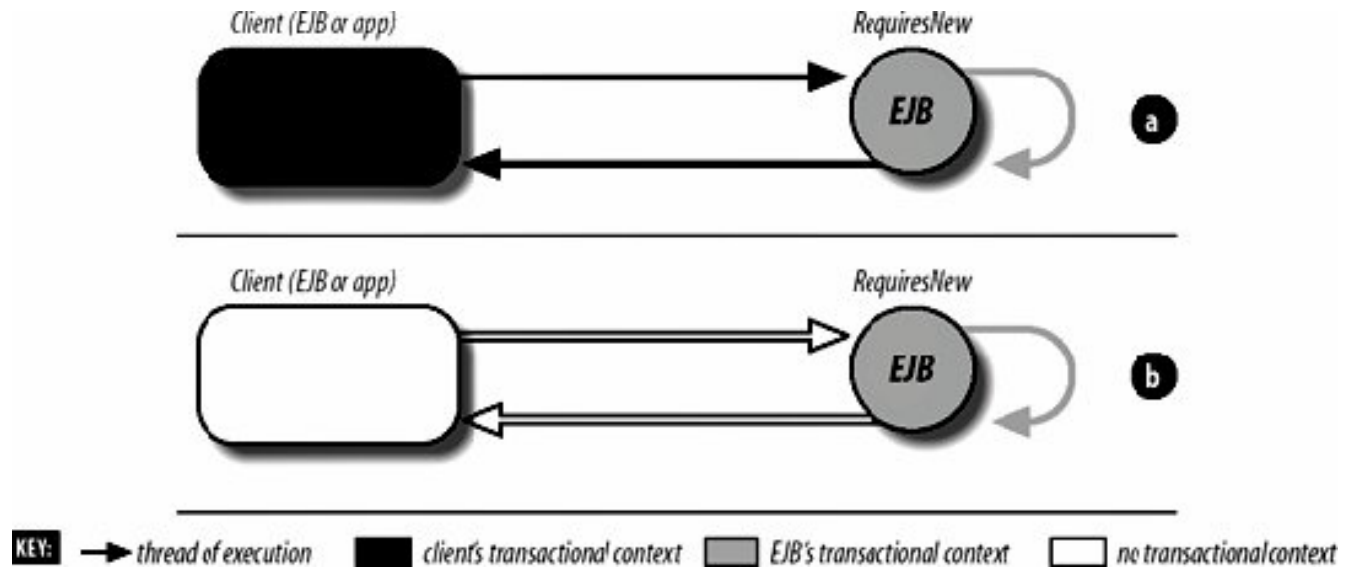


Transaction attributes defined

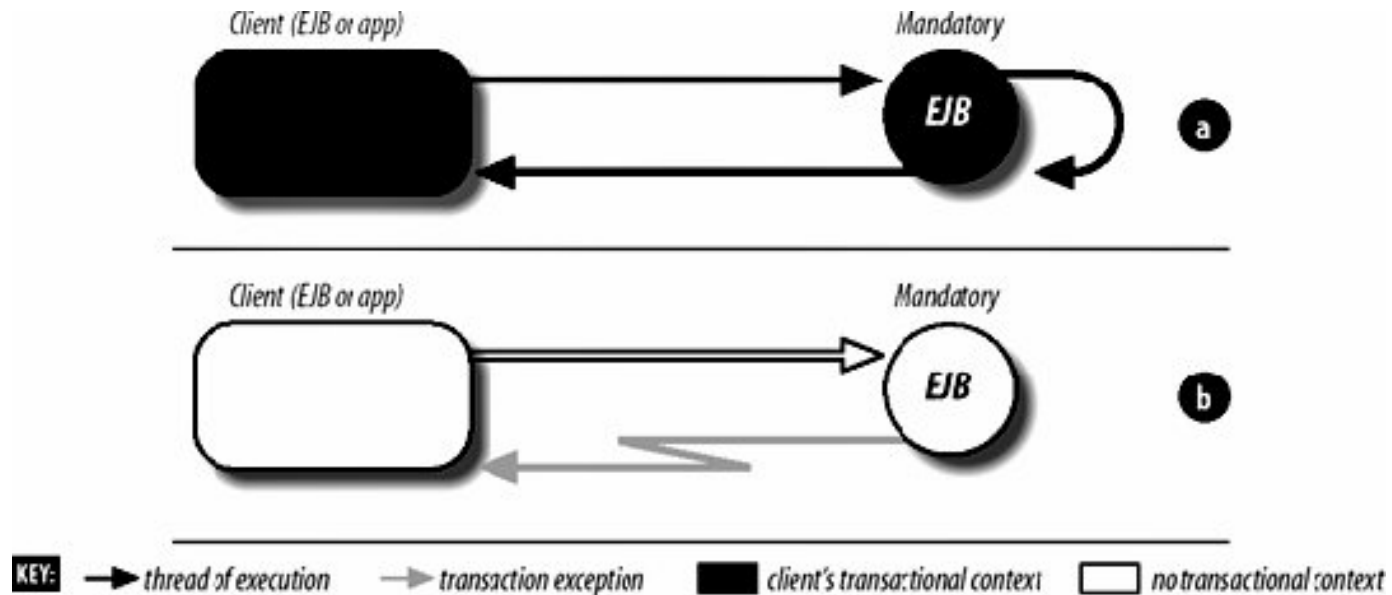
- Required
 - This attribute means that the enterprise bean method must be invoked within the scope of a transaction.



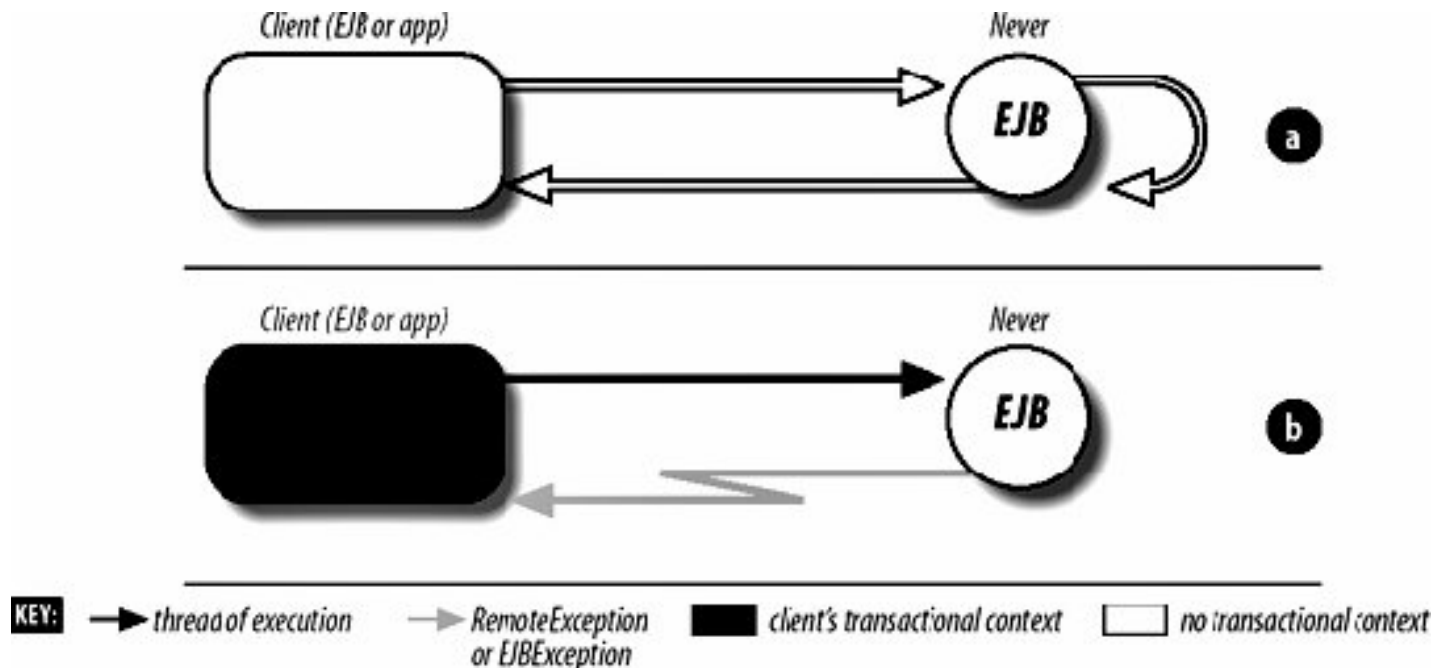
- RequiresNew
 - This attribute means that a new transaction is always started.



- Mandatory
 - This attribute means that the enterprise bean method must always be made part of the transaction scope of the calling client.



- Never
 - This attribute means that the enterprise bean method must not be invoked within the scope of a transaction.



Transaction Attributes and Scope

| Transaction Attribute | Client's Transaction | Business Method's Transaction |
|-----------------------|----------------------|-------------------------------|
| Required | None | T2 |
| Required | T1 | T1 |
| RequiresNew | None | T2 |
| RequiresNew | T1 | T2 |
| Mandatory | None | Error |
| Mandatory | T1 | T1 |
| NotSupported | None | None |
| NotSupported | T1 | None |
| Supports | None | None |
| Supports | T1 | T1 |
| Never | None | None |
| Never | T1 | Error |

- Transaction attributes are specified by
 - decorating the enterprise bean class or method with a `javax.ejb.TransactionAttribute` annotation
 - and setting it to one of the `javax.ejb.TransactionAttributeType` constants.

| Transaction Attribute | TransactionAttributeType Constant |
|-----------------------|--|
| Required | TransactionAttributeType.REQUIRED |
| RequiresNew | TransactionAttributeType.REQUIRES_NEW |
| Mandatory | TransactionAttributeType.MANDATORY |
| NotSupported | TransactionAttributeType.NOT_SUPPORTED |
| Supports | TransactionAttributeType.SUPPORTS |
| Never | TransactionAttributeType.NEVER |

- The following code snippet demonstrates how to use the `@TransactionAttribute` annotation:

```
@TransactionAttribute(NOT_SUPPORTED)
@Stateful
public class TransactionBean implements Transaction {
    ...
    @TransactionAttribute(REQUIRES_NEW)
    public void firstMethod() {...}

    @TransactionAttribute(REQUIRED)
    public void secondMethod() {...}
    public void thirdMethod() {...}
    public void fourthMethod() {...}
}
```

- There are two ways to roll back a container-managed transaction.
 - First, if a system exception is thrown, the container will automatically roll back the transaction.
 - Second, by invoking the `setRollbackOnly` method of the `EJBContext` interface, the bean method instructs the container to roll back the transaction.
 - If the bean throws an application exception, the rollback is not automatic but can be initiated by a call to `setRollbackOnly`.

- The `SessionSynchronization` interface, which is optional, allows stateful session bean instances to receive transaction synchronization notifications.
 - The container invokes the `SessionSynchronization` methods (`afterBegin`, `beforeCompletion`, and `afterCompletion`) at each of the main stages of a transaction.

```
afterBegin:
```

```
    oldvalue = value;
```

```
afterCompletion(boolean b):
```

```
    if (!b)
```

```
        value = oldvalue;
```


- The list of prohibited methods follows:
 - The `commit`, `setAutoCommit`, and `rollback` methods of `java.sql.Connection`
 - The `getUserTransaction` method of `javax.ejb.EJBContext`
 - Any method of `javax.transaction.UserTransaction`

- In **bean-managed transaction demarcation**,
 - the code in the session or message-driven bean explicitly marks the boundaries of the transaction.
- Pseudocode:

```
begin transaction
...
update table-a
...
if (condition-x)
    commit transaction
else if (condition-y)
    update table-b
    commit transaction
else rollback transaction
begin transaction
    update table-c
commit transaction
```

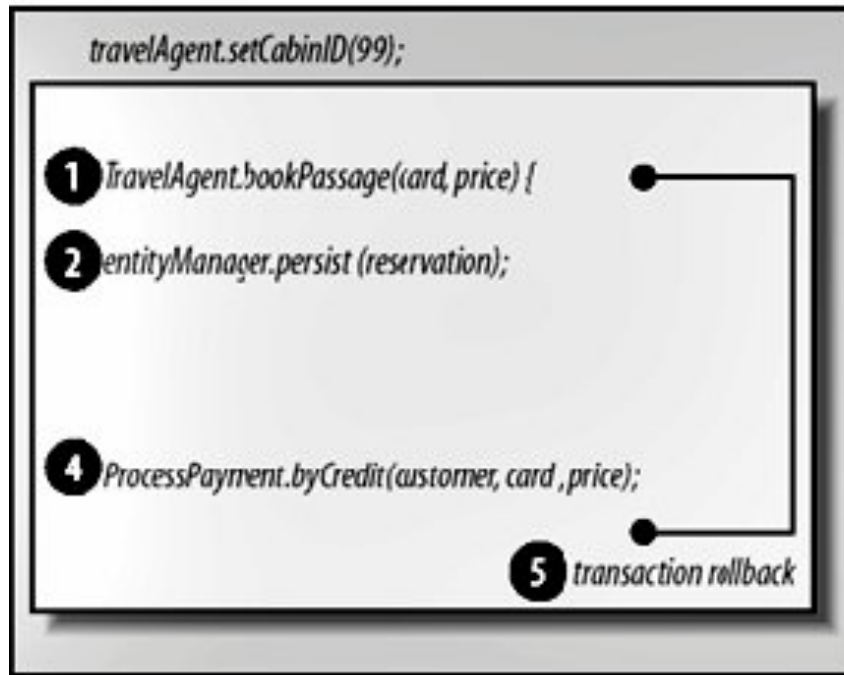
- For container-managed transactions,
 - you can use the Administration Console to configure the transaction timeout interval.
- For enterprise beans with bean-managed JTA transactions,
 - you invoke the `setTransactionTimeout` method of the `UserTransaction` interface.

- Let's think about two separate client applications accessing the same data specifically.

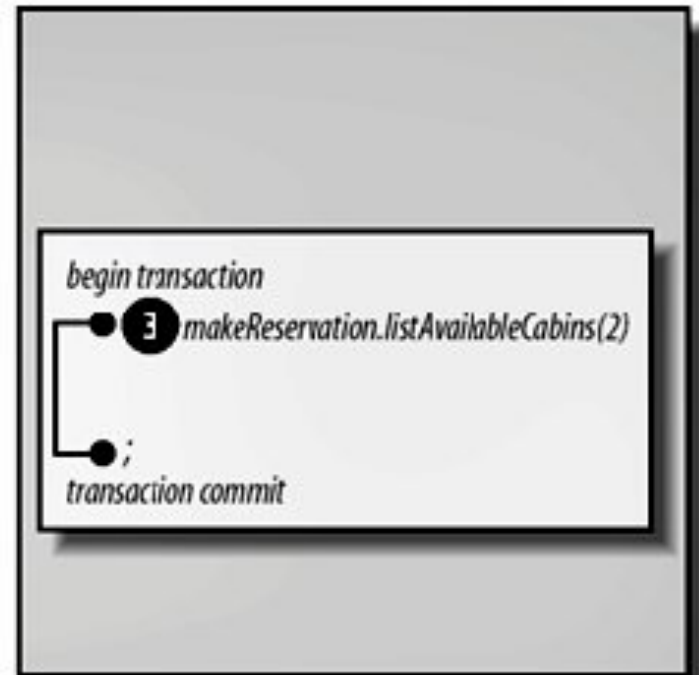
```
public List listAvailableCabins(int bedCount) {  
    Query query = entityManager.createQuery(  
        "SELECT name FROM Cabin c  
        WHERE c.ship = :ship AND  
        c.bedCount = :beds AND NOT ANY (  
        SELECT cabin from Reservation res  
        WHERE res.cruise = :cruise");  
    query.setParameter("ship", cruise.getShip( ));  
    query.setParameter("beds", bedCount);  
    query.setParameter("cruise", cruise);  
    return query.getResultList( );  
}
```

- For this example, assume that both methods have a transaction attribute of **Required**.

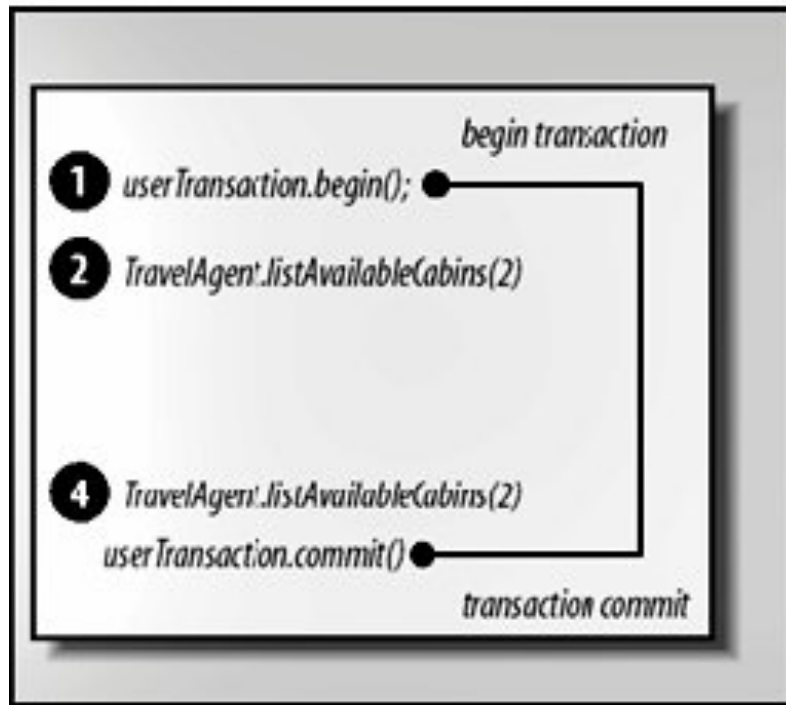
Client 1



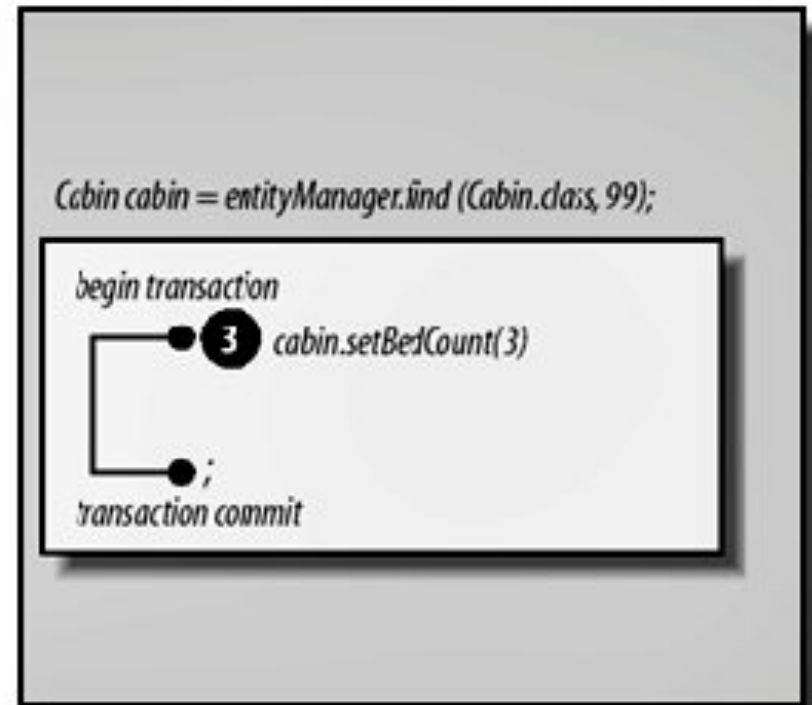
Client 2



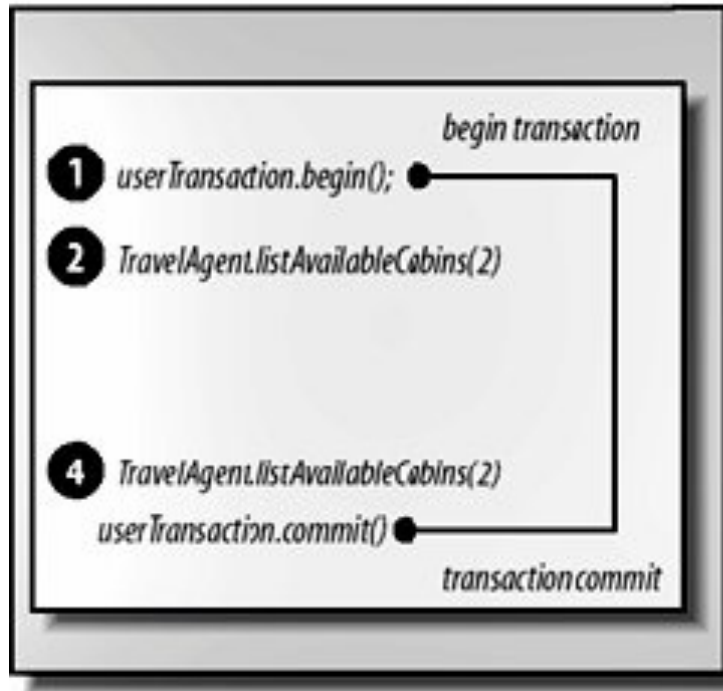
Client 1



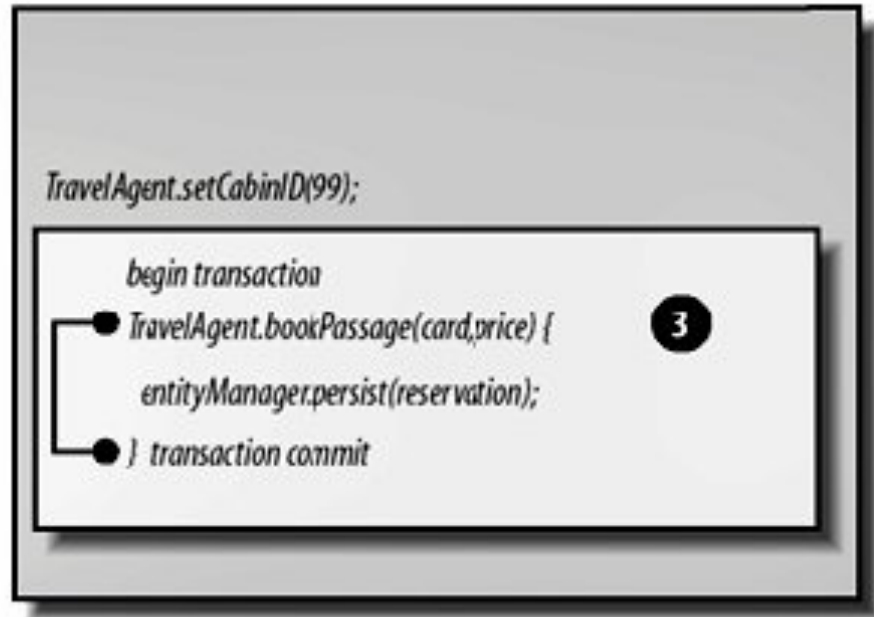
Client 2



Client 1



Client 2



- **Read locks**

- Read locks prevent other transactions from **changing data read during a transaction until the transaction ends**, thus preventing **nonrepeatable** reads.
- Other transactions can read the data but not write to it. The current transaction is **also** prohibited from making changes.

- **Write locks**

- Write locks are used for updates. A write lock prevents other transactions from **changing the data until the current transaction is complete but allows dirty reads by other transactions and by the current transaction itself**.
- In other words, the transaction can read its own **uncommitted** changes.

- **Exclusive write locks**

- Exclusive write locks are used for updates. An exclusive write lock prevents other transactions from **reading or changing the data until the current transaction is complete**.
- It also prevents **dirty reads** by other transactions.

- **Snapshots**

- A snapshot is a **frozen view** of the data that is taken when a transaction begins. Some databases get around locking by providing every transaction with its own snapshot.
- **Snapshots can prevent dirty reads, nonrepeatable reads, and phantom reads**. They can be problematic because the data is not real-time data; it is old the instant the snapshot is taken.

- **Read Uncommitted**

- The transaction can read uncommitted data (i.e., data changed by a different transaction that is still in progress).
- Dirty reads, nonrepeatable reads, and phantom reads can occur.
- Bean methods with this isolation level can read uncommitted changes.

- **Read Committed**

- The transaction cannot read uncommitted data; data that is being changed by a different transaction cannot be read.
- Dirty reads are prevented; nonrepeatable reads and phantom reads can occur.
- Bean methods with this isolation level cannot read uncommitted data.

- **Repeatable Read**
 - The transaction cannot change data that is being read by a different transaction.
 - Dirty reads and nonrepeatable reads are prevented; phantom reads can occur.
 - Bean methods with this isolation level have the same restrictions as those in the Read Committed level and can execute only repeatable reads.
- **Serializable**
 - The transaction has exclusive read and update privileges; different transactions can neither read nor write to the same data.
 - Dirty reads, nonrepeatable reads, and phantom reads are prevented.
 - This isolation level is the most restrictive.

- You to specify the transaction isolation level using the database's API.
- For example:

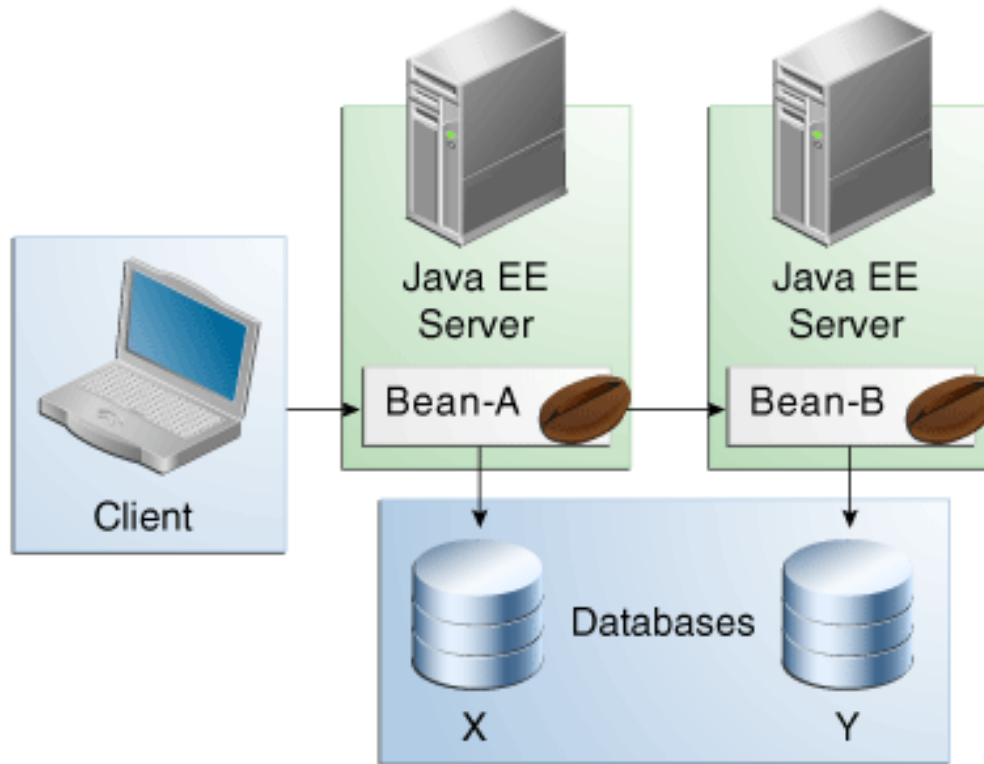
```
DataSource source = (javax.sql.DataSource)
```

```
jndiCntxt.lookup("java:comp/env/jdbc/titanDB");
```

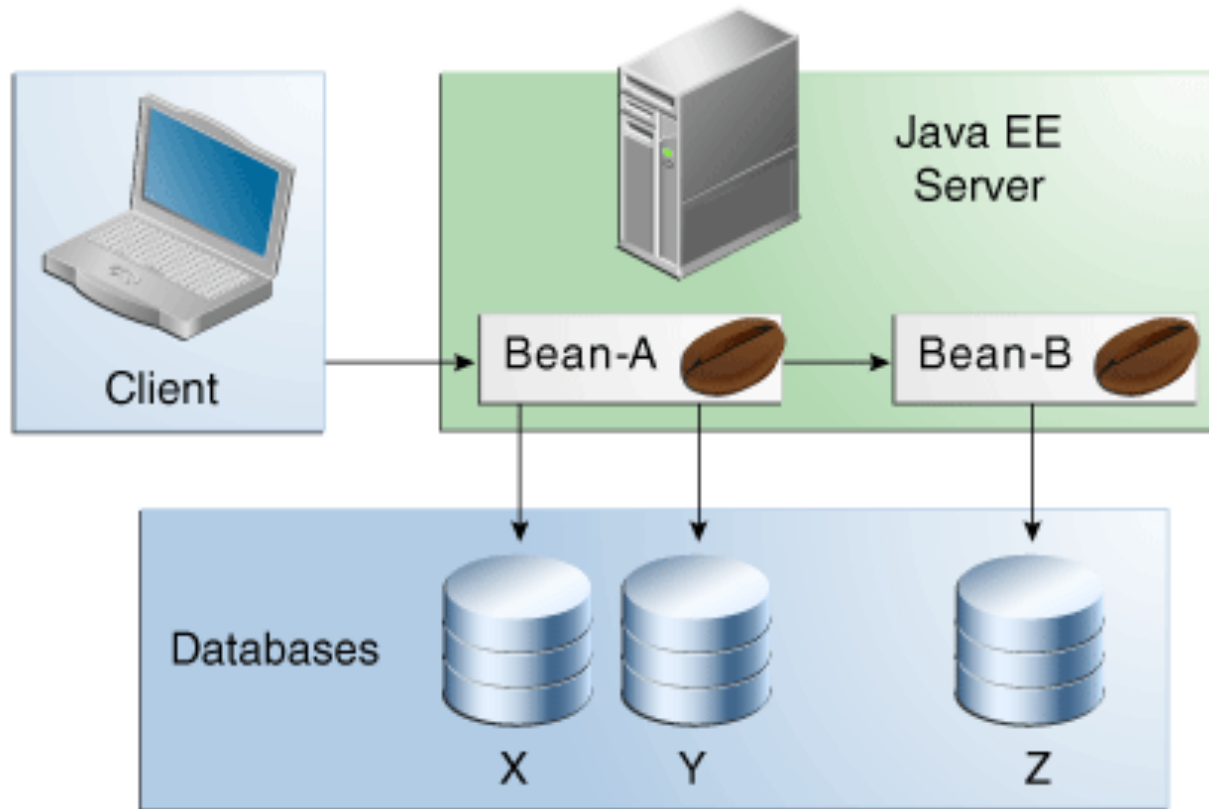
```
    Connection con = source.getConnection( );
```

```
con.setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE);
```

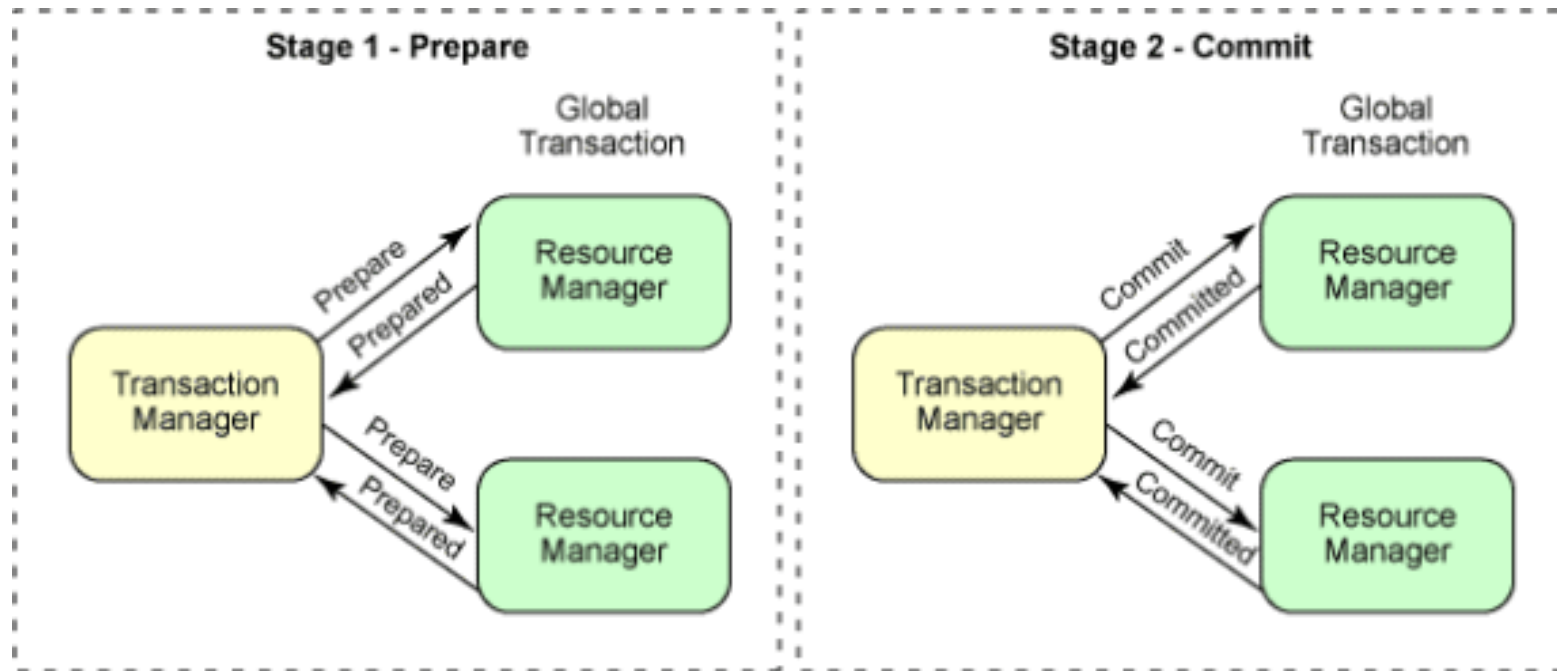
Updating Multiple Databases



Updating Multiple Databases



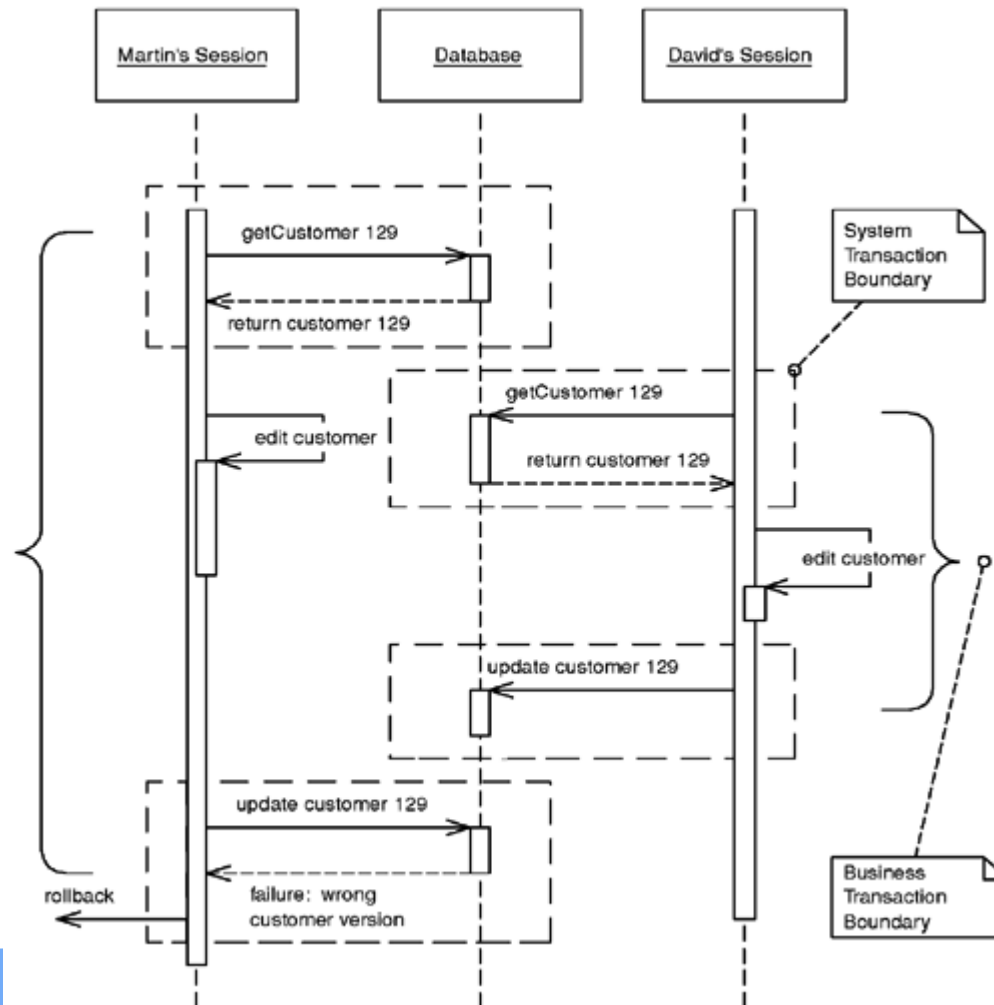
Two-phase commit



- Since we use O/R mapping, and it is an offline way to access database, we should manage the concurrent access to data.
- The core is how to prevent conflict between business transactions.
- Locking is the solution
 - What kind of lock?

Optimistic Offline Lock

- Optimistic Offline Lock solves this problem by validating that the changes about to be committed by one session don't conflict with the changes of another session.



- An Optimistic Offline Lock is obtained by validating that, in the time since a session loaded a record, another session hasn't altered it.
 - It can be acquired at any time but is valid only during the system transaction in which it is obtained.
 - Thus, in order that a business transaction not corrupt record data it must acquire an Optimistic Offline Lock for each member of its change set during the system transaction in which it applies changes to the database.
- The most common implementation is to associate a version number with each record in your system.
 - With an RDBMS data store the verification is a matter of adding the version number to the criteria of any SQL statements used to update or delete a record.

- Our data is stored in a relational database,
 - so each table must also store version and modification data.
 - Here's the schema for a customer table as well as the standard CRUD SQL necessary to support the Optimistic Offline Lock:

table customer...

```
create table customer(id bigint primary key, name varchar, createdby  
    varchar, created datetime, modifiedby varchar, modified datetime,  
    version int)
```

SQL customer CRUD...

```
INSERT INTO customer VALUES (?, ?, ?, ?, ?, ?, ?)
```

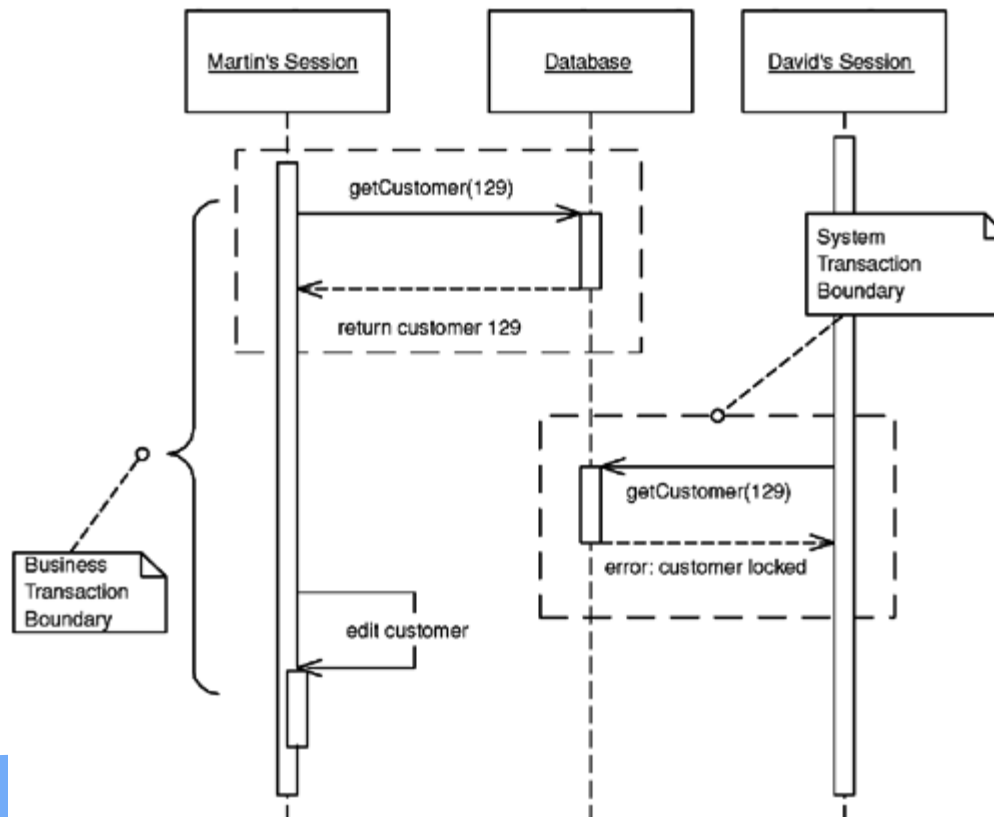
```
SELECT * FROM customer WHERE id = ?
```

```
UPDATE customer SET name = ?, modifiedBy = ?, modified = ?, version = ?  
    WHERE id = ? and version = ?
```

```
DELETE FROM customer WHERE id = ? and version = ?
```

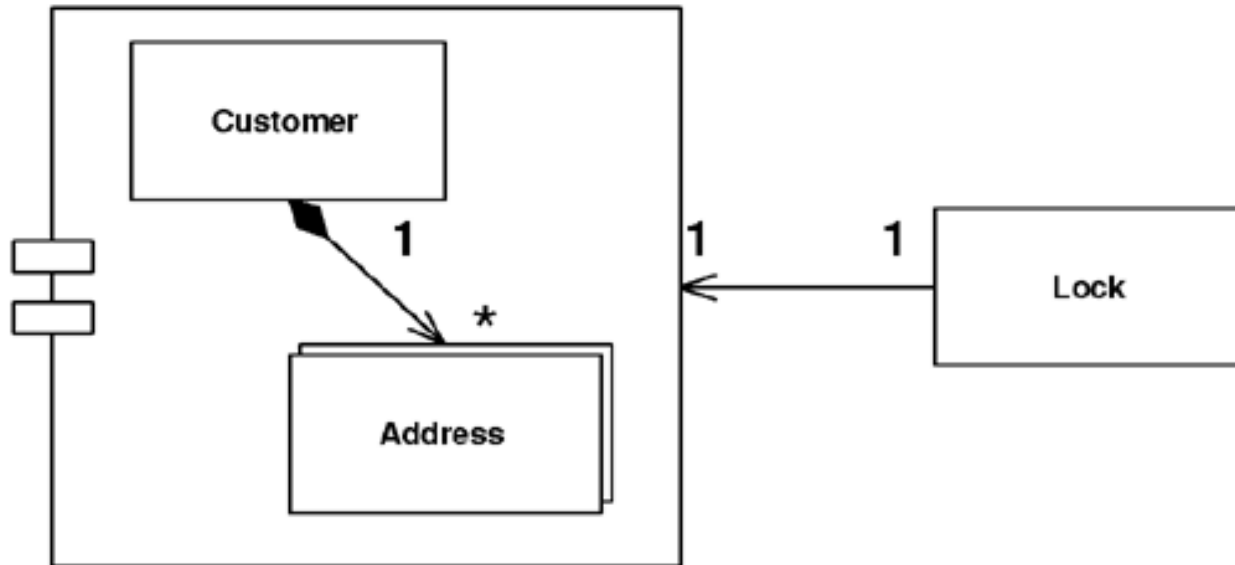
Pessimistic Offline Lock

- Pessimistic Offline Lock prevents conflicts by avoiding them altogether.
 - It forces a business transaction to acquire a lock on a piece of data before it starts to use it, so that, most of the time, once you begin a business transaction you can be pretty sure you'll complete it without being bounced by concurrency control.



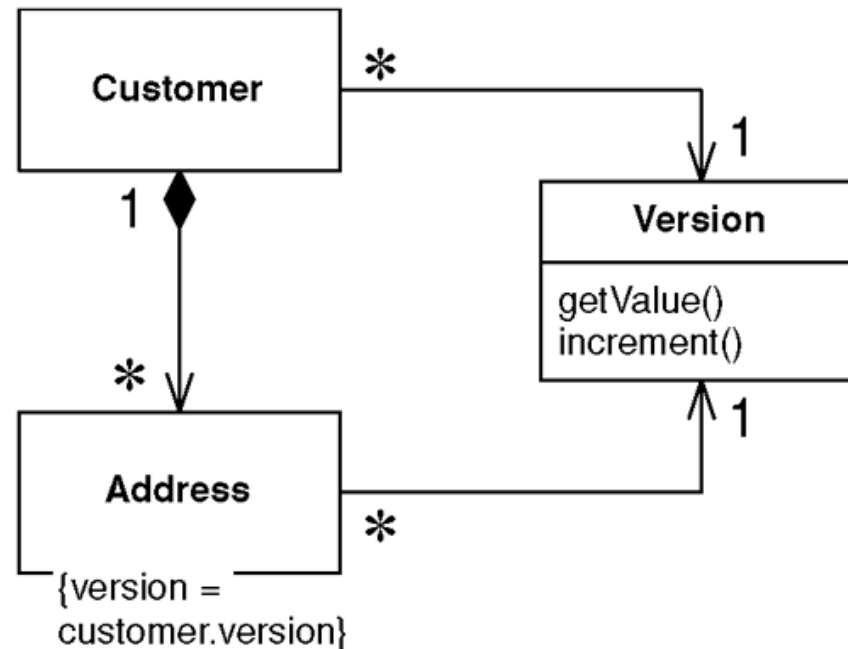
- You implement Pessimistic Offline Lock in three phases:
 - determining what type of locks you need,
 - building a lock manager,
 - and defining procedures for a business transaction to use locks.
- Lock types:
 - exclusive write lock
 - exclusive read lock
 - read/write lock
 - Read and write locks are mutually exclusive.
 - Concurrent read locks are acceptable.
- In choosing the correct lock type think about maximizing system concurrency, meeting business needs, and minimizing code complexity.
 - Also keep in mind that the locking strategy must be understood by domain modelers and analysts.

- Locks a set of related objects with a single lock.

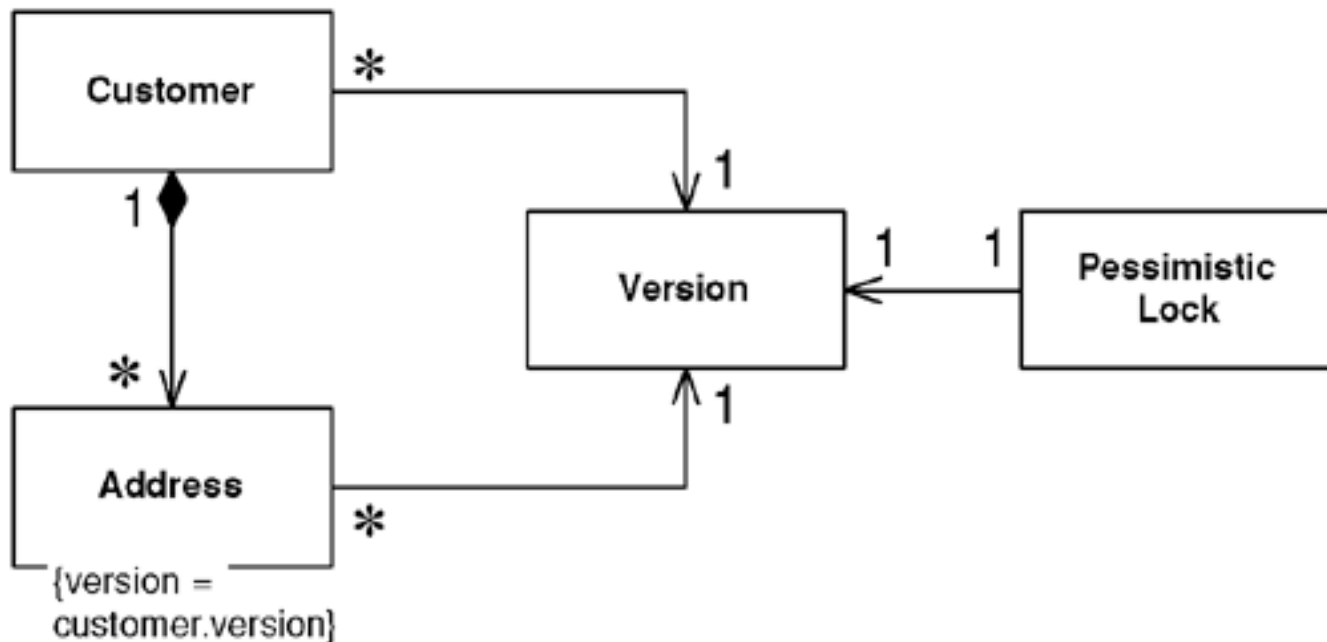


- A Coarse-Grained Lock
 - is a single lock that covers many objects.
 - It not only simplifies the locking action itself but also frees you from having to load all the members of a group in order to lock them.

- With **Optimistic Offline Lock**, having each item in a group share a version creates the single point of contention, which means sharing the **same** version, not an **equal** version.
 - Incrementing this version will lock the entire group with a shared lock.
 - Set up your model to point every member of the group at the shared version and you have certainly minimized the path to the point of contention.



- A shared **Pessimistic Offline Lock** requires that each member of the group share some sort of lockable token, on which it must then be acquired.
 - As Pessimistic Offline Lock is often used as a complement to Optimistic Offline Lock, a shared version object makes an excellent candidate for the lockable token role.



- To improve the quality of your book store website, please add necessary transaction management into it, including:
 - To setup proper isolation level of your DB.
 - To implement data access with declarative transactions.
 - To manage the declarative transactions with at least 3 types.

- Web Applications: What are They? What of Them?
 - <http://www.acunetix.com/websitesecurity/web-applications/>
- The Java EE 7 Tutorial
 - <http://docs.oracle.com/javaee/7/tutorial/doc/javaeetutorial7.pdf>
- Mastering EJB 3
- IBM WebSphere Developer Technical Journal: Exploiting the J2EE Connector Architecture
 - https://www.ibm.com/developerworks/websphere/techjournal/0607_wakelin/0607_wakelin.html



Thank You!