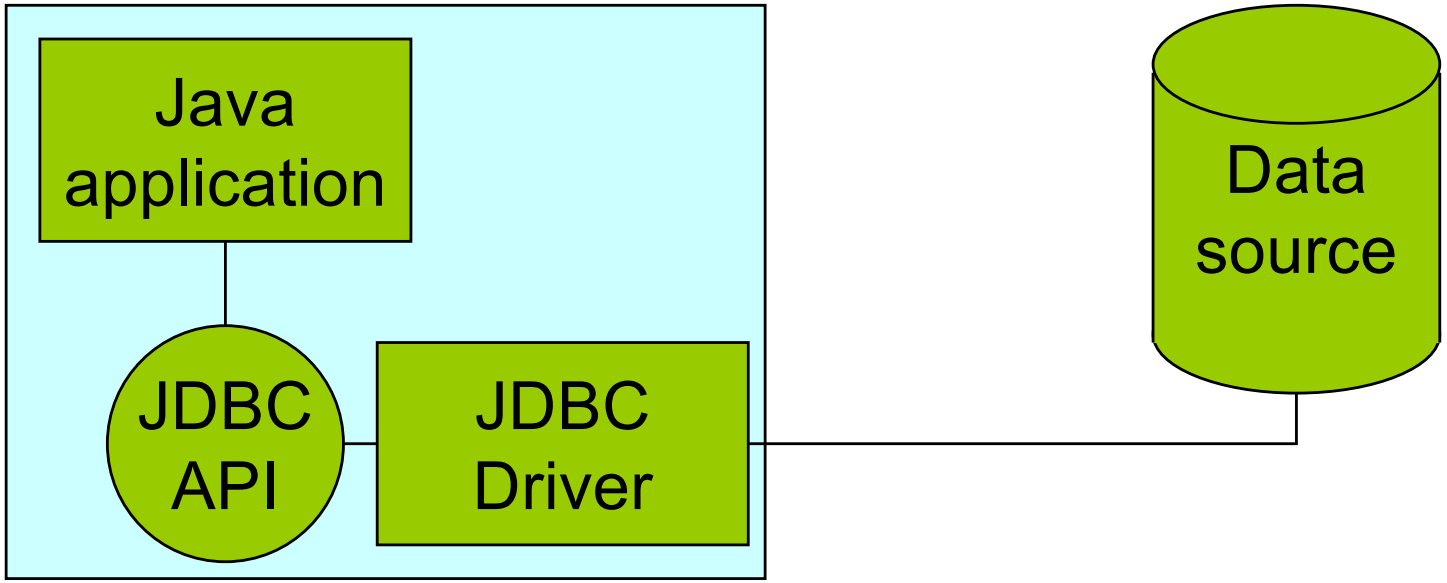
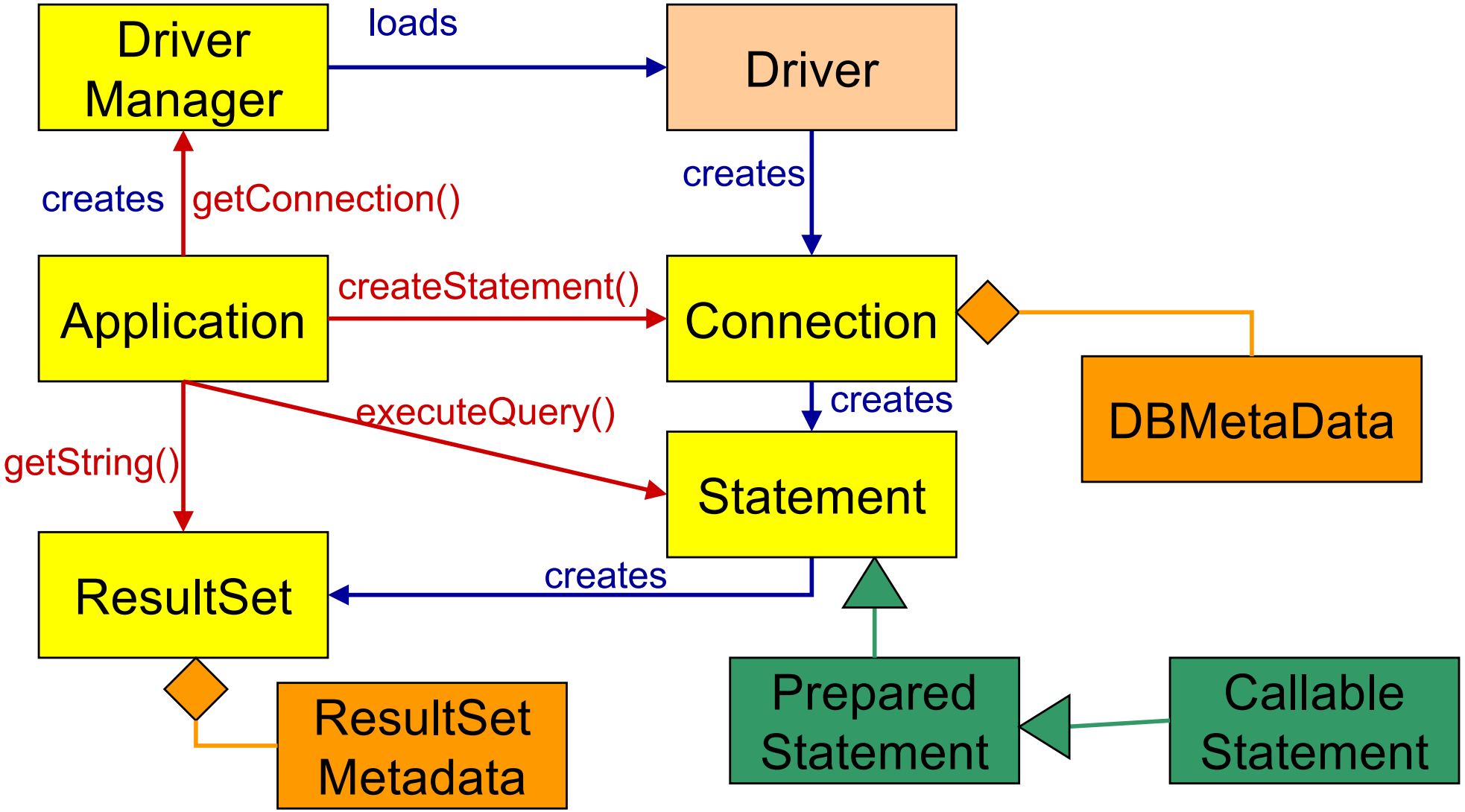
Transactions

Introduction

JDBC access (reminder)



The java.sql Object Model (reminder)



Bank

1. getConnection/setConnection

```
package transactions_1;
import java.sql.*;
public class Bank {

 public Connection getConnection(String jdbcDriverName,
                                 String jdbcURL) {
   try {
     Class.forName(jdbcDriverName);
     return DriverManager.getConnection(jdbcURL);
   } catch (ClassNotFoundException ex) { ex.printStackTrace();
   } catch (SQLException ex) { ex.printStackTrace(); }
   return null;
 }

 public void releaseConnection(Connection conn) {
   if (conn!=null)
     try {
       conn.close();
     } catch (SQLException ex) { ex.printStackTrace(); }
 }
```

Bank

2. deposit/withdraw

```
public void deposit(int account, double amount, Connection conn)
    throws SQLException{
  String sql="UPDATE Account SET Balance = Balance + "+ amount+
          "WHERE AccountId = "+account;
  Statement stmt=conn.createStatement();
  stmt.executeQuery(sql);
  System.out.println("Deposited "+amount+" to account "+account);
 }

 public void withdraw(int account, double amount, Connection conn)
    throws SQLException{
  String sql="UPDATE Account SET Balance = Balance - "+ amount+
          "WHERE AccountId = "+account;
  Statement stmt=conn.createStatement();
  stmt.executeQuery(sql);
  System.out.println("Withdrew "-amount+" from account "+
          account);
 }
```

Bank

3. printBalance

```
public void printBalance(Connection conn) {
    ResultSet rs=null;
    Statement stmt=null;
    try {
      stmt=conn.createStatement();
      rs=stmt.executeQuery("SELECT * FROM Account");
      while (rs.next())
        System.out.println("Account "+rs.getInt(1)+
                           " has a balnce of "+rs.getDouble(2));
    } catch (SQLException ex) { ex.printStackTrace(); }
    finally {
      try {
        if (rs!=null)
          rs.close();
        if (stmt!=null)
          stmt.close();
      } catch (SQLException ex) { ex.printStackTrace(); }
    }
  }
```

Bank

4. trasferFunds

```
public void transferFunds(int fromAccount, int toAccount,
                          double amount, Connection conn){
    Statement stmt=null;
    try {
      withdraw(fromAccount, amount, conn);
      deposit(toAccount,amount,conn);
    }
    catch (SQLException ex) {
      System.out.println("An error occured!");
      ex.printStackTrace();
    }
}
```

Bank

5. main

```
 public static void main(String[] args) {
   if (args.length <3) {
     System.exit(1);
   }
   Connection conn=null;
   Bank bank = new Bank();
   try {
     conn=bank.getConnection(args[0],args[1]);
     bank.transferFunds(1,2,Double.parseDouble(args[2]),conn);
     bank.printBalance(conn);
   } catch (NumberFormatException ex) { ex.printStackTrace();
   } finally {bank.releaseConnection(conn);}
 }
}
```

Bank

transferFunds – fixed version!

```
public void transferFunds(int fromAccount, int toAccount,
                          double amount, Connection conn){
   Statement stmt=null;
   try {
     conn.setAutoCommit(false);
     withdraw(fromAccount, amount, conn);
     deposit(toAccount,amount,conn);
     conn.commit();
   }
   catch (SQLException ex) {
     System.out.println("An error occured!");
     ex.printStackTrace();
     try {
       conn.rollback();
     } catch (SQLException e) { e.printStackTrace(); }
   }
}
```

TRANSACTION

Transactions

A transaction defines a logical unit of work that either completely succeeds or produces no result at all.

A distributed transaction is a transaction that accesses and updates data on two or more networked resources, and therefore must be coordinated among those resources (not necessariy only DB)

Transactions properties

ACID

The ACID Properties

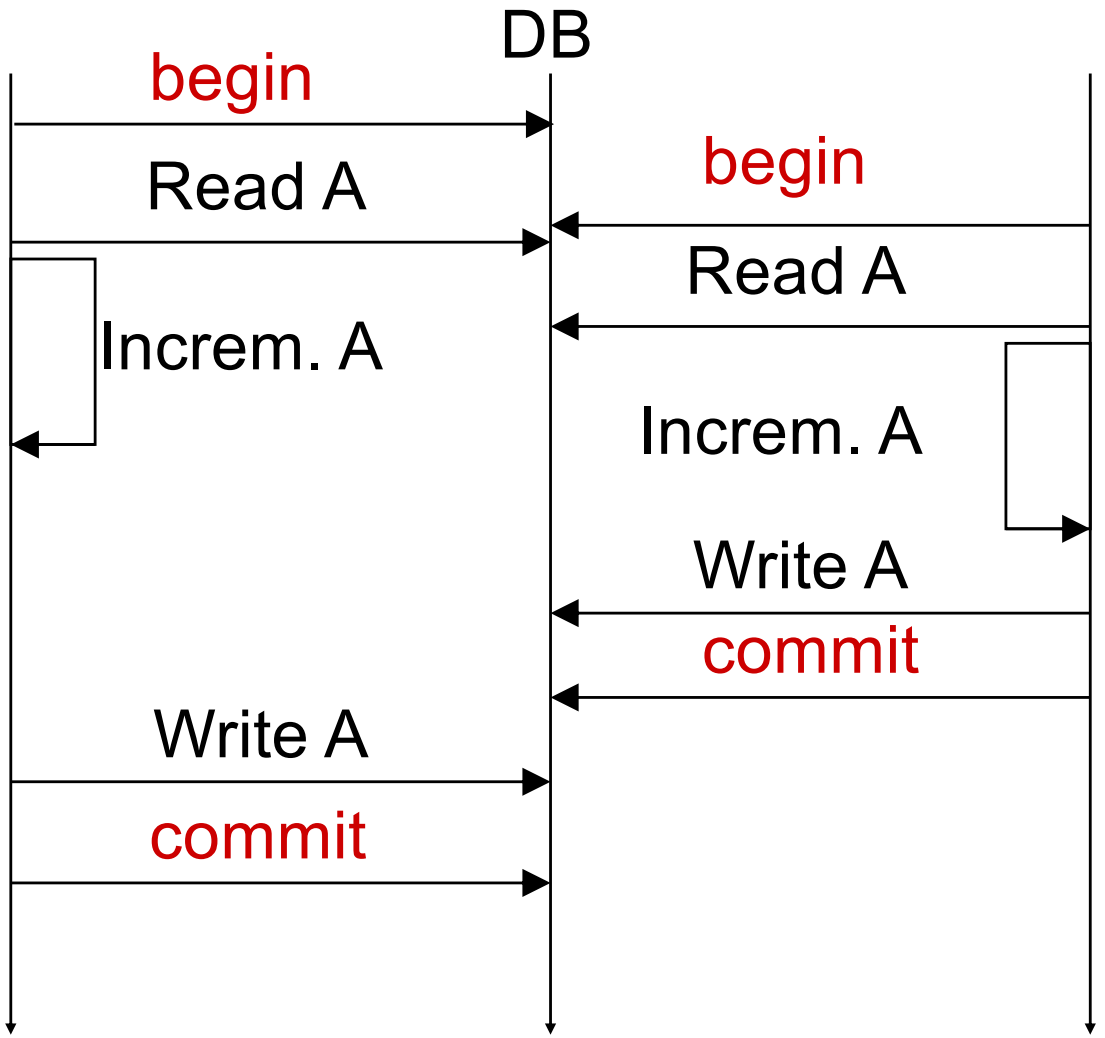
Atomicity *guarantees that many operations are bundled together and appear as one contiguous unit of work .*

Consistency *guarantees that a transaction leaves the system ′s state to be consistent after a transaction completes.*

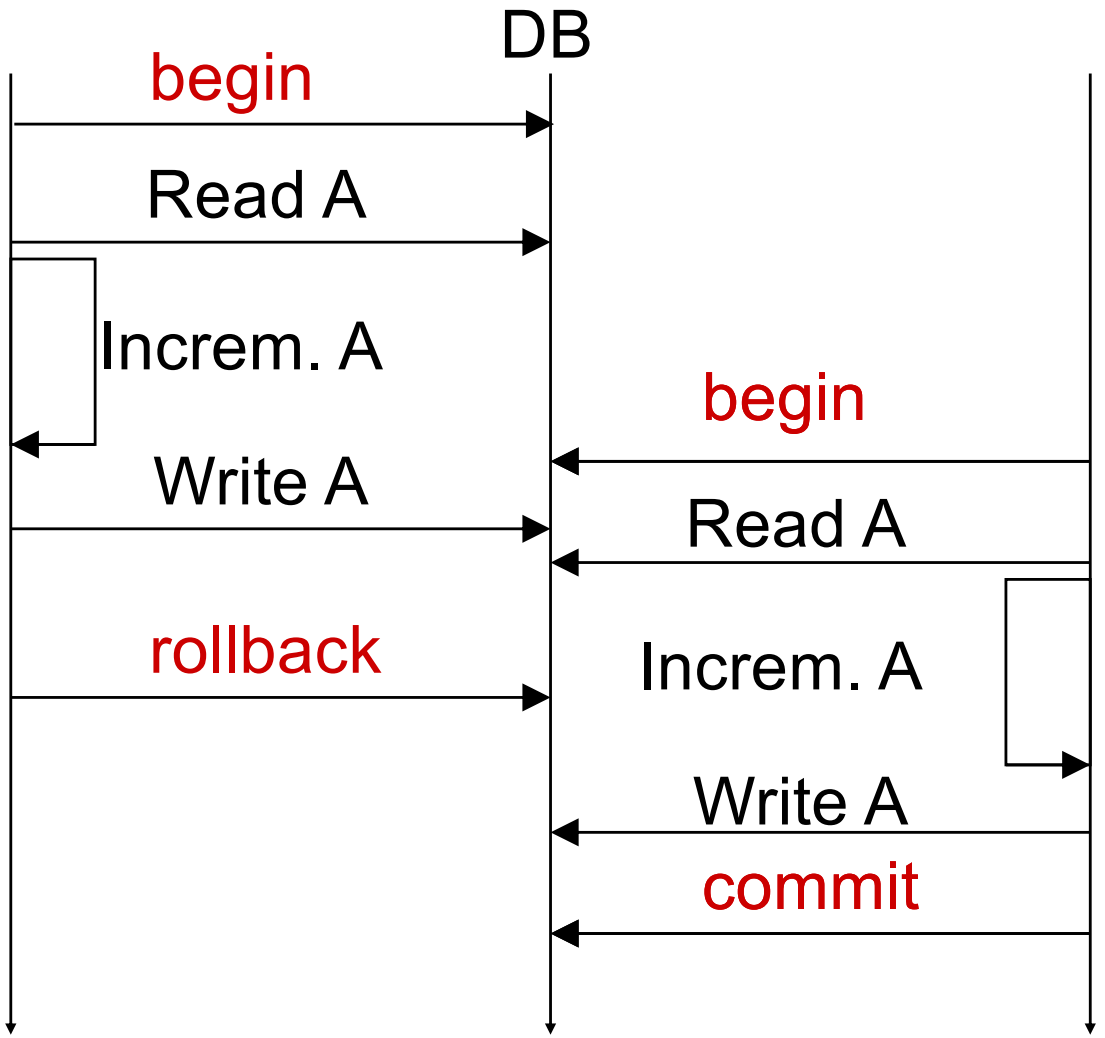
Isolation *protects concurrently executing transactions from seeing eachother ′s incomplete results.*

Durability *guarantees that updates to managed resources,such as database records,survive failures. (Recoverable resources keep a transactional log for exactly this purpose.If the resource crashes,the permanent data can be reconstructed by reapplying the steps in the log.)*

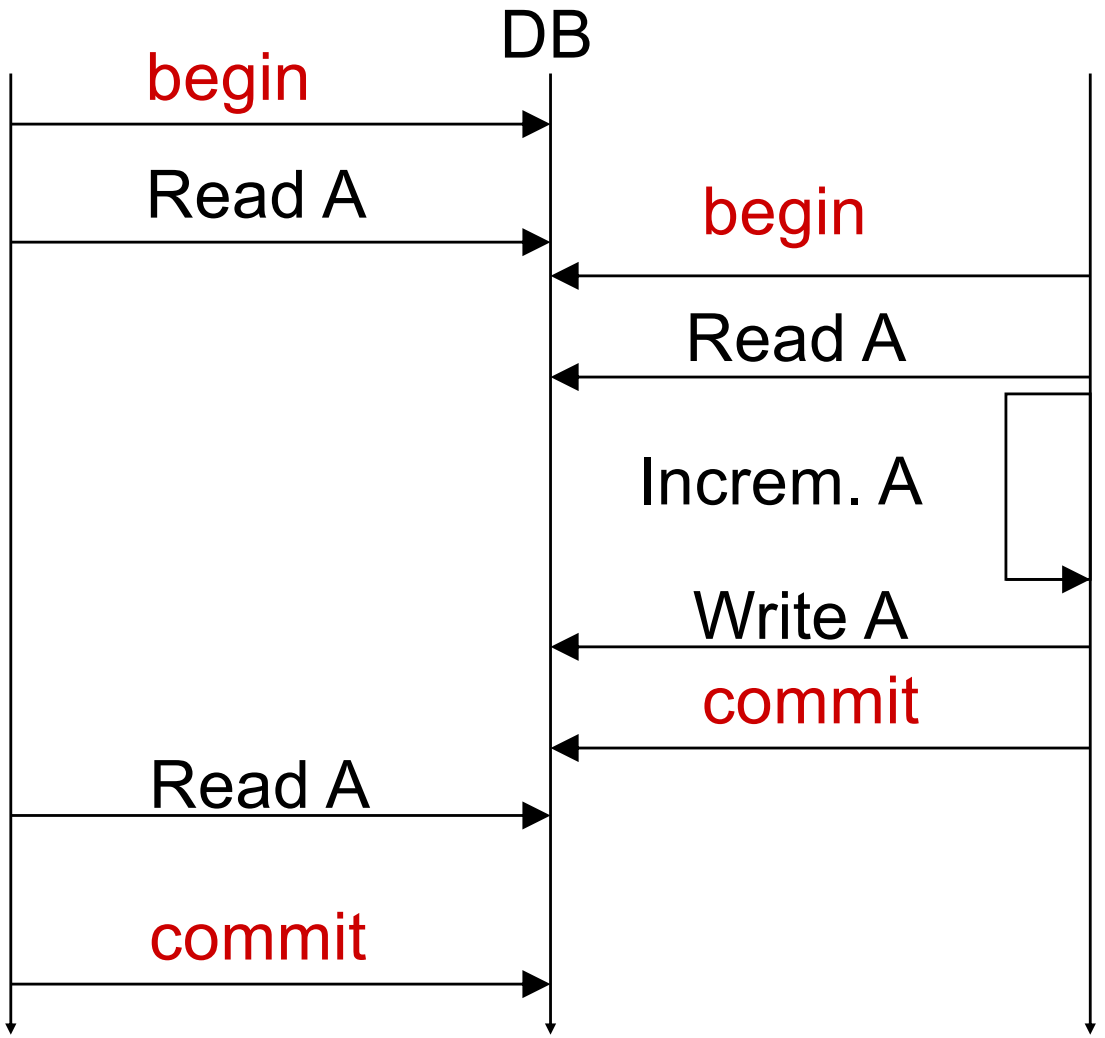
Lost Update


DB
begin
Read A
Increm. A
begin
Read A
Increm. A
Write A
commit
Write A
commit

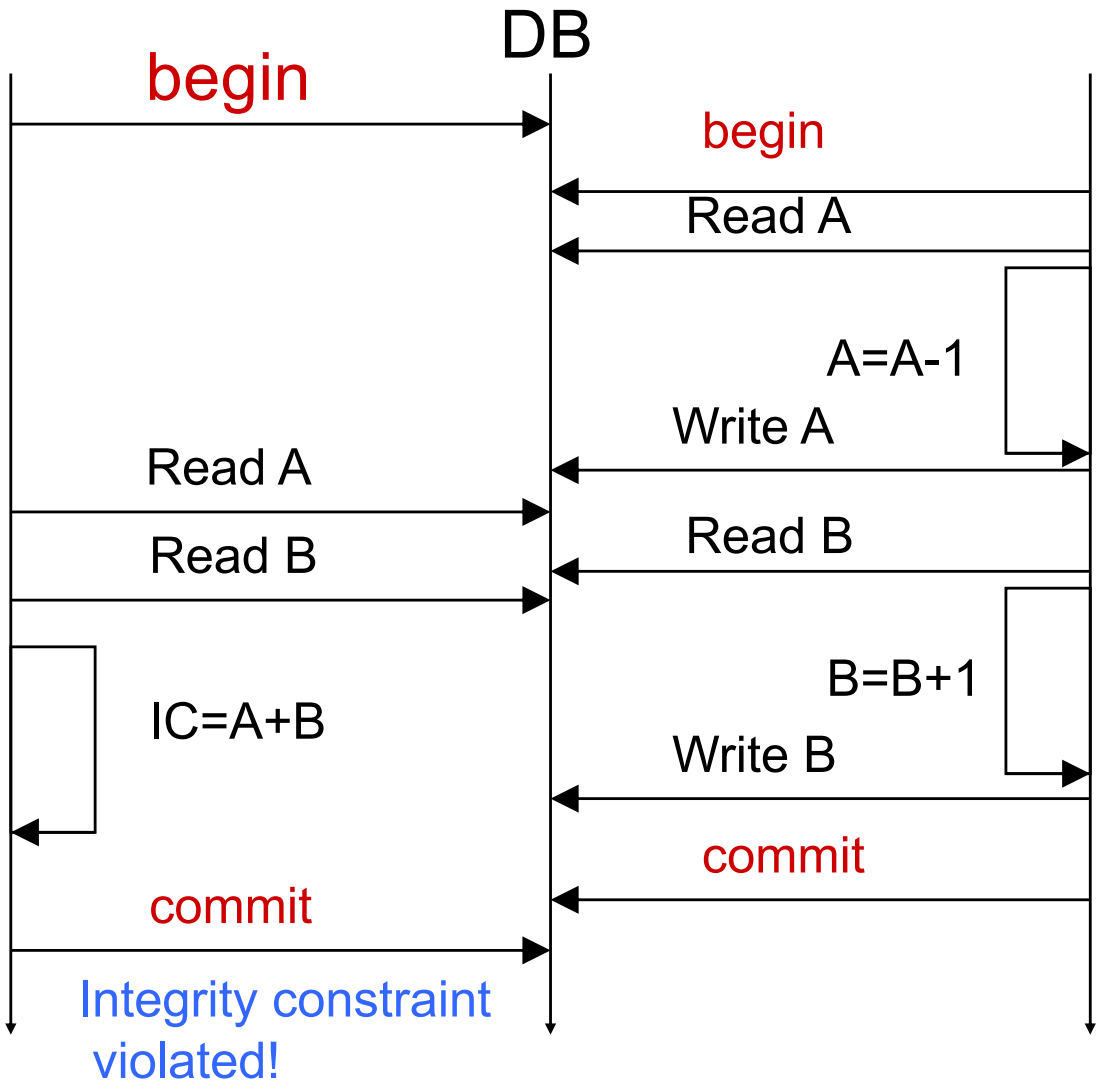
Dirty Read


DB
begin
Read A
Increm. A
begin
Write A
Read A
rollback
Increm. A
Write A
commit

Unrepeatable Read



Phantom Read (ghost update)



Integrity Constraint:
A+B=100

Isolation levels

ISOLATION LEVEL	Dirty Read	Unrepeatable Read	Phantom Read
READ UNCOMMITTED	YES	YES	YES
READ COMMITTED	NO	YES	YES
REPEATABLE READ	NO	NO	YES
SERIALIZABLE	NO	NO	NO

Default level for many DBMS

Pessimistic and Optimistic Concurrency Control Strategies

TIPO	Dimension	Concurrency	Problems
Pessimistic —Your EJB locks the source data for the entire time it needs data, not allowing anything else to potentially update the data until it completes its transaction.	Small Systems	Low	Does not scale well
Optimistic - Your EJB implements a strategy to detect whether a change has occurred to the source data between the time it was read and the time it now needs to be updated. Locks are placed on the data only for the small periods of time the EJB interacts with the database.	Large Systems	High	Complexity of the collision detection code

ACID vs BASE

Classic distributed systems: focused on ACID semantics

- Atomic
- Consistent
- Isolated
- Durable

Modern Internet systems: focused on BASE

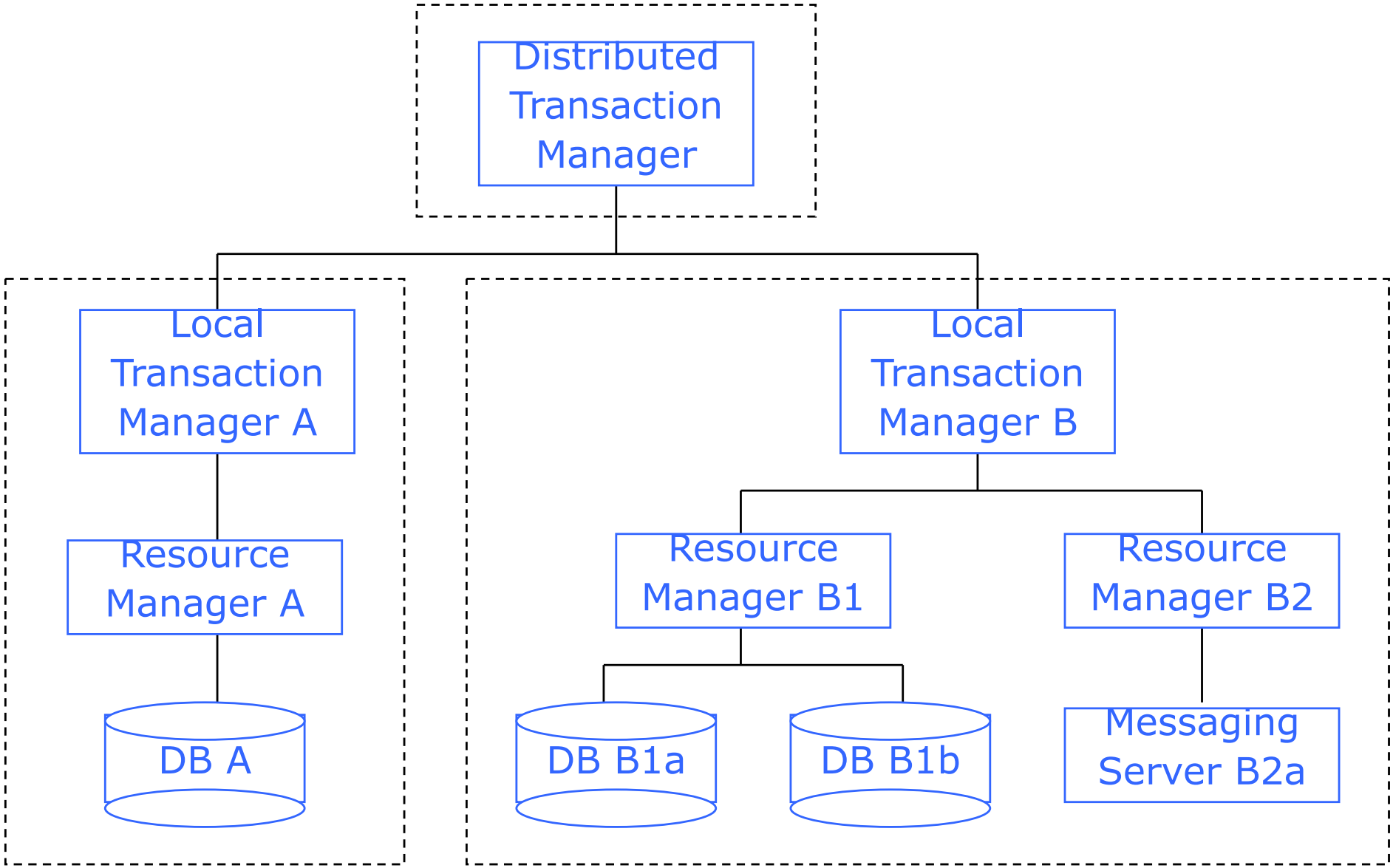
- Basically Available
- Soft-state (or scalable)
- Eventually consistent

Distributed Transactions

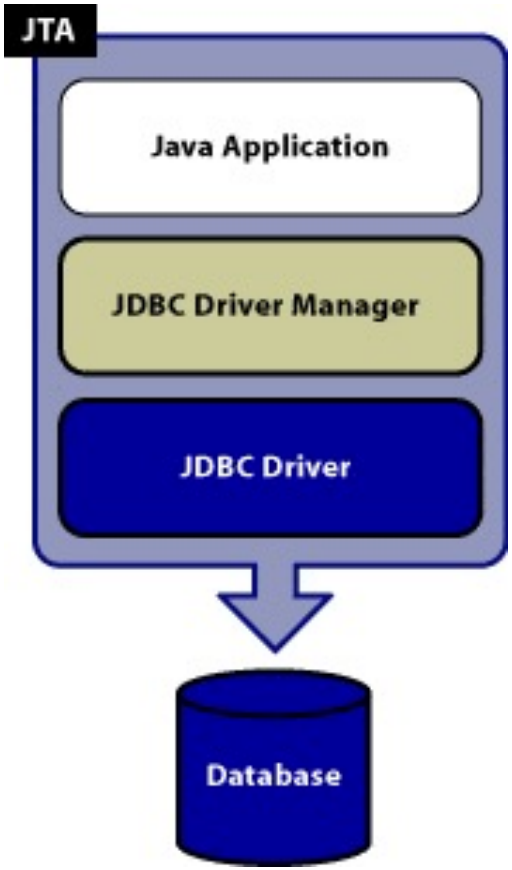
Actors

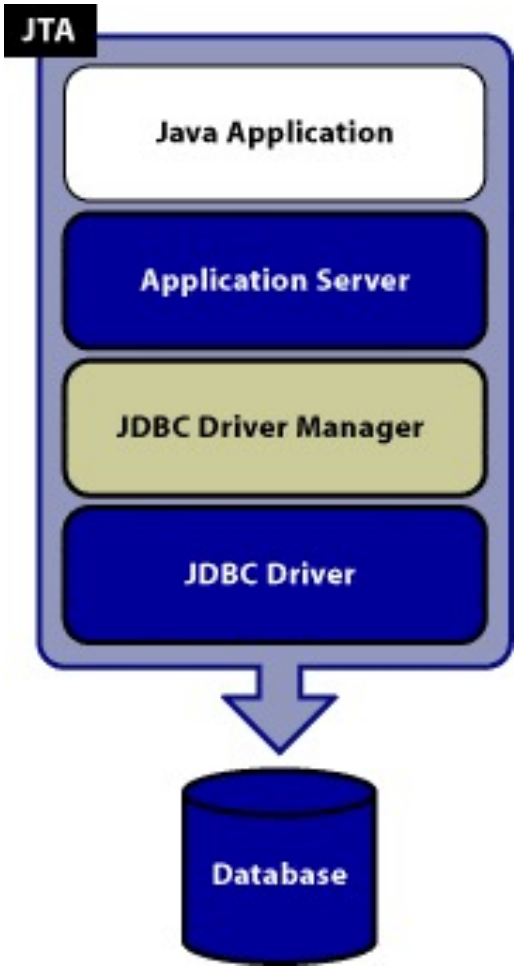
A *transactional object* (or *transactional component*) is an application component that is involved in a transaction.
A *transaction manager* is responsible for managing the transactional operations of the transactional components.
A *resource* is a persistent storage from which you read or write.
A *resource manager* manages a resource. Resource managers are responsible for managing all state that is permanent.
The most popular interface for resource managers is the *X/Open XA* resource manager interface (a de facto standard): a deployment with heterogeneous resource managers from different vendors can interoperate.

Distributed Systems



Local vs. Distributed transactions – part 1





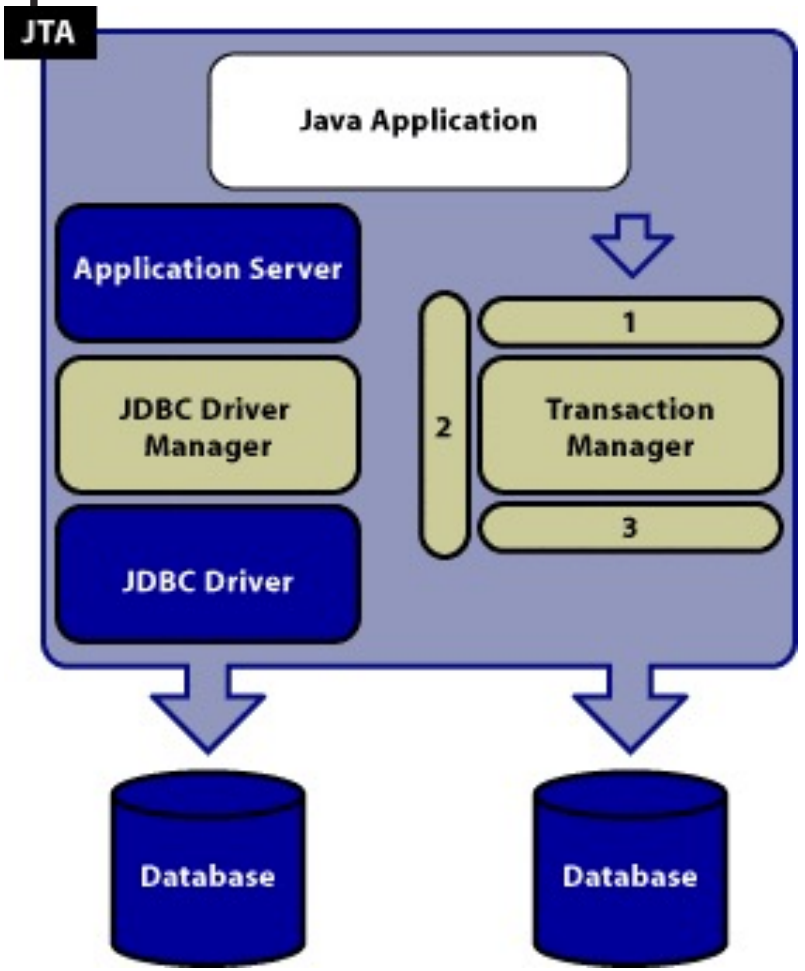
see https://www.progress.com/tutorials/jdbc/understanding-jta

Local vs. Distributed transactions – part 2

1—UserTransaction—an interface that provides the application the ability to control transaction boundaries programmatically. It starts a global transaction and associates the transaction with the calling thread.

2—Transaction Manager—an interfacethat allows the application server to control transaction boundaries on behalf of the application being managed.

3—XAResource—an interface mapping the industry standard XA (Distributed Transaction Processing: The XA Specification).



for details see https://www.progress.com/tutorials/jdbc/understanding-jta

A word of warning

Transactions over replicated data introduce extra challenges.

What goals might you want from a shared-data system?

Strong Consistency: all clients see the same view, even in the presence of updates

High Availability: all clients can find some replica of the data, even in the presence of failures

Partition-tolerance: the system properties hold even when the system is partitioned

Brewer's Theorem: that's impossible! you can have only two of them at the same time.

This and the following three slides are adpted from lectures by Prof. Ion Stoica & Scott Shenker (UC, Berkeley)

Transactions

Java Transactions

Java Transaction API

JTA consists of two sets of interfaces:

- *one for X/Open XA resource managers (which you don't need to worry about)*

- *one that we will use to support programmatic transaction control: javax.transaction.UserTransaction .*

javax.transaction.UserTransaction

Methods for Transactional Boundary Interaction

begin()

Begins a new transaction.This transaction becomes associated with the current thread.

commit()

Runs the two-phase commit protocol on an existing transaction associated with the current thread. Each resource manager will make its updates durable

getStatus()

Retrieves the status of the transaction associated with this thread.

rollback()

Forces a rollback of the transaction associated with the current thread.

javax.transaction.UserTransaction

Methods for Transactional Boundary Interaction

setRollbackOnly()

Calls this to force the current transaction to roll back.
This will eventually force the transaction to abort.

setTransactionTimeout(int)

The transaction timeout is the maximum amount of time that a transaction can run before it's aborted.This is useful to avoid deadlock situations,when precious resources are being held by a transaction that is currently running.

The *javax.transaction.Status* Constants

STATUS_ACTIVE
A transaction is currently happening and is active.
STATUS_NO_TRANSACTION
No transaction is currently happening.
STATUS_MARKED_ROLLBACK ***The current transaction will eventually abort because it 's been marked for rollback.This could be because some party called UserTransaction.setRollbackOnly().***
STATUS_ROLLING_BACK ***The current transaction is in the process of rolling back.***
STATUS_ROLLEDBACK ***The current transaction has been rolled back.***
STATUS_UNKNOWN ***The status of the current transaction cannot be determined.***

The *javax.transaction.Status* Constants

STATUS_PREPARING ***The current transaction is preparing to be committed (during Phase One of the two-phase commit protocol).***

STATUS_PREPARED ***The current transaction has been prepared to be committed (Phase One is complete).***

STATUS_COMMITTING ***The current transaction is in the process of being com mitted right now (during Phase Two).***

STATUS_COMMITTED ***The current transaction has been committed (Phase Two is complete).***

Without annotation

```
import javax.transaction.UserTransaction;


…
try {
   java.util.Properties env =...
   // Get the JNDI initial context
   Context ctx =new InitialContext(env);
      userTran=(javax.transaction.UserTransaction)
      ctx.lookup("java:comp/UserTransaction");
// Execute the transaction
   userTran.begin();
   /* perform business operations */
   userTran.commit();
}
catch (Exception e){
    //deal with any exceptions}
```

Set environment up.You must set the JNDI InitialContext factory,the Provider URL,and any login names or passwords necessary to access JNDI.

Look up the JTA UserTransaction interface via JNDI.The container is required to make the JTA available at the location java:comp/UserTransaction

With annotation

```
import javax.transaction.UserTransaction;

@PersistenceContext private EntytiManage em
@Resource private javax.transaction.UserTransactin userTran
…
try {
  java.util.Properties env =...
// Execute the transaction
  userTran.begin();
  /* perform business operations */
  userTran.commit();
}
catch (Exception e){
   //deal with any exceptions}
```

Transactions in EJB

Declarative Transactions

Who begins a transaction?

Who begins a transaction? *Who* issues either a commit or abort? This is called *demarcating transactional boundaries* .

There are three ways to demarcate transactions:

- *programmatically:*
  you are responsible for issuing a *begin* statement and either a *commit* or an *abort* statement.
- *declaratively,*
  the EJB container *intercepts* the request and starts up a transaction automatically on behalf of your bean.
- *client-initiated.*
  write code to start and end the transaction from the client code outside of your bean.

Programmatic vs. declarative

programmatic transactions:

your bean has full control over transactional boundaries.*For instance,you can use programmatic transactions to run a series of minitransactions within a bean method.*

When using programmatic transactions,always try to complete your transactions in the same method that you began them.Doing otherwise results in spaghetti code where it is difficult to track the transactions;the performance decreases because the transaction is held open longer.

declarative transactions:

your entire bean method must either run under a transaction or not run under a transaction*.*

Transactions are simpler! (just declare them in the descriptor)

Client-initiated

Client initiated transactions:

A non-transactional remote client calls an enterprise bean that performs its own transactions The bean succeeds in the transaction,but the network or application server crashes before the result is returned to a remote client.The remote client would receive a Java RMI RemoteException indicating a network error,but would not know whether the transaction that took place in the enterprise bean was a success or a failure.

With client-controlled transactions, if anything goes wrong,the client will know about it.

The downside to client-controlled transactions is that if the client is located far from the server, ***transactions are likely to take a longer time and the efficiency will suffer****.*

Transactional Models

A *flat transaction* is the simplest transactional model to understand.A flat transaction is a series of operations that are performed atomically as a single *unit of work* .

A *nested transaction* allows you to embed atomic units of work within other units of work.The unit of work that is nested within another unit of work can roll back without forcing the entire transaction to roll back. (*subtransactions can independently roll back without affecting higher transactions in the tree*)
(Not currently mandated by the EJB specification)

Other models: *chained transactions* and *sagas*.
(Not supported by the EJB specification)

EJB Transaction Attribute Values

Required

You want your method to always run in a transaction.

If a transaction is already running,your bean joins in on that transaction. If no transaction is running,the EJB container starts one for you.

Never

Your bean cannot be involved in a transaction.

If the client calls your bean in a transaction,the container throws an exception back to the client (java.rmi.RemoteException if remote, javax.ejb.EJBException if local).

EJB Transaction Attribute Values

Supports

The method runs only in a transaction if the client had one running already *—it joins that transaction.*
If the client does not have a transaction,the bean runs with no transaction at all.

Mandatory

a transaction must be already running when your bean method is called*. If a transaction isn´t running, javax.ejb.TransactionRequiredException is thrown back to the caller (or javax.ejb.TransactionRequiredLocalException if the client is local).*

EJB Transaction Attribute Values

NotSupported

your bean cannot be involved in a transaction at all.

For example,assume we have two enterprise beans,A and B.Let ′s assume bean A begins a transaction and then calls bean B. If bean B is using the NotSupported attribute,the transaction that A started is suspended. None of B′s operations are transactional,such as reads/writes to databases. When B completes,A ′s transaction is resumed.

EJB Transaction Attribute Values

RequiresNew

*You should use the RequiresNew attribute if you **always want a new transaction to begin** when your bean is called. If a transaction is already underway when your bean is called,that transaction is suspended during the bean invocation.*

The container then launches a new transaction and delegates the call to the bean.The bean performs its operations and eventually completes.The container then commits or aborts the transaction and finally resumes the old transaction. If no transaction is running when your bean is called,there is nothing to suspend or resume.

EJB Transaction Attribute Values

TYPE	PRECONDITION	POSTCONDITION
Required	NO transaction	NEW
	PRE-EXISTING	PRE-EXISTING
RequiresNew	NO transaction	NEW
	PRE-ESISTENTE	NEW
Supports	NO transaction	NO transaction
	PRE-EXISTING	PRE-EXISTING
Mandatory	NO transaction	error
	PRE-EXISTING	PRE-EXISTING
NotSupported	Nessuna transazione	NO transaction
	PRE-ESISTENTE	NO transaction
Never	NO transaction	NO transaction
	PRE-EXISTING	error

Annotations

```
@Stateless
@TransactionManagement(javax.ejb.TransactonManagementType.CONTAINER)
public class Mybean implements Myinterface{
@PersistencyContext private EntityManager em;
@Resource private SessionContext ctx;
…
@TransactionAttribute(javax.ejb.TransactonManagementType.REQUIRED)
   public void myTransactedMethod(){…
        if (…) ctx.setRollbackOnly;
   }
}
```

EJB Transaction Attribute Values

```
<assembly-descriptor>
    <container-transaction>
      <method>
            <ejb-name>Employee</ejb-name>
            <method-name>*</method-name>
      </method>
      <trans-attribute>Mandatory</trans-attribute>
    </container-transaction>
    <container-transaction>
      <method>
            <ejb-name>Employee</ejb-name>
            <method-name>setName</method-name>
            <method-param>String</method-param>
      </method>
      <trans-attribute>Required</trans-attribute>
    </container-transaction>
</assembly-descriptor>
```

Dooming container-managed transactions

call *setRollbackOnly()* on your *EJB context* object.
If the transaction participant is not an Container Managed EJB component, you can doom a transaction by looking up the JTA and calling the JTA 's *setRollbackOnly()* method,

Container-managed transactional beans can detect doomed transactions by calling the *getRollbackOnly()*method on the EJB context object. If this method returns *true* ,the transaction is doomed.

Isolation levels in EJB

BMT:

you specify isolation levels with your resource manager API (such as JDBC).

For example,you could call java.sql.Connection.SetTransactionIsolation(...).

CMT:

there is no way to specify isolation levels in the deployment descriptor.

You need to either use resource manager APIs (such as JDBC),or rely on your container ′s tools or database ′s tools to specify isolation.

Isolation portability problems

Unfortunately, there is no way to specify isolation for container-managed transactional beans in a portable way—you are reliant on container and database tools.

This means if you have written an application, you cannot ship that application with built-in isolation. The deployer now needs to know about transaction isolation when he uses the container's tools, and the deployer might not know a whole lot about your application's transactional behavior.