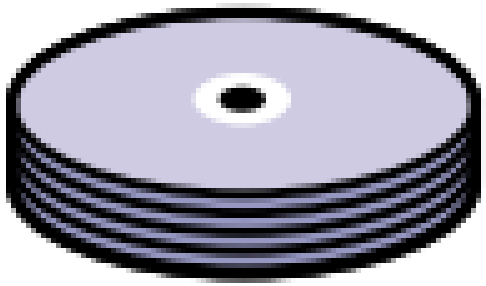
Implementing a Transaction Policy

Modul 12
Motivation: Datenbankzugriff JDBC

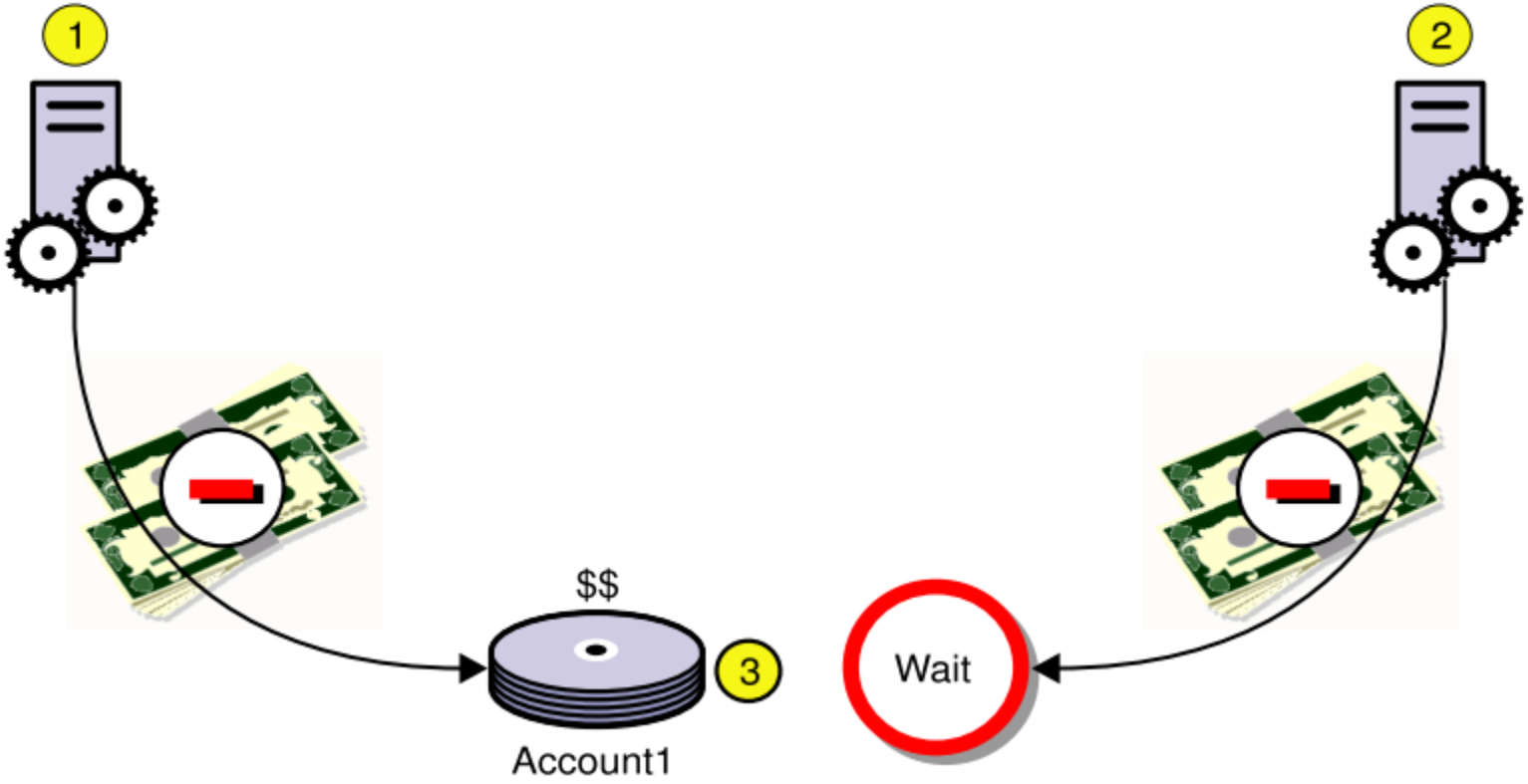
```
commit;
insert …
update …
delete …
commit;
```

Eine Transaktion (atomar)
innerhalb einer DB



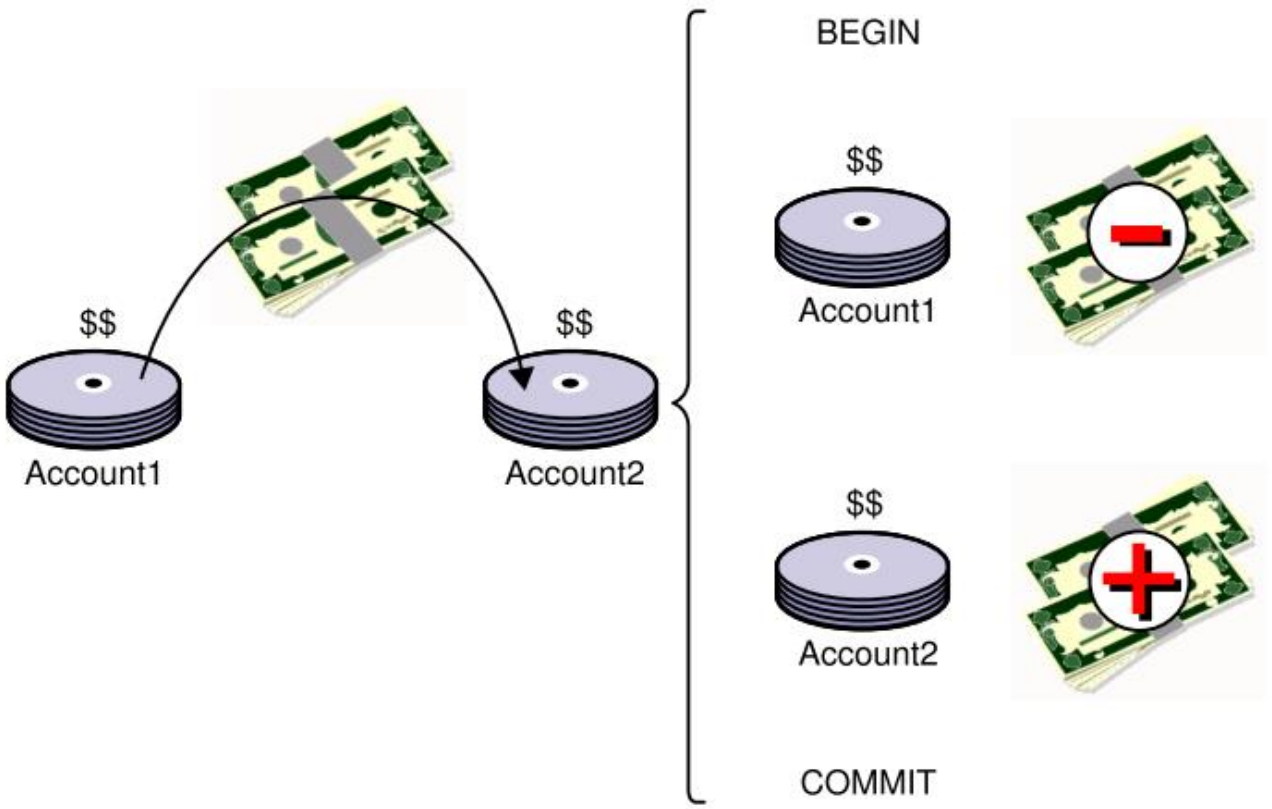
Modul 12
Motivation: Datenbankzugriff JDBC

Eine Transaktion innerhalb einer DBs: Isolation & Locking



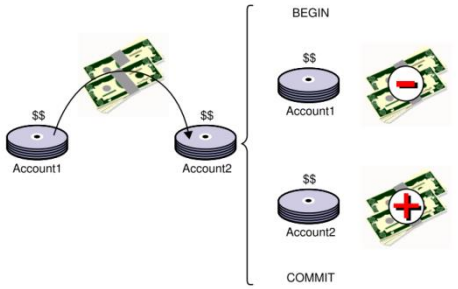
Modul 12
Motivation: Transaktionen über mehrere DBs hinweg

Mehrere Datenbanken sind Teil einer Transaktion

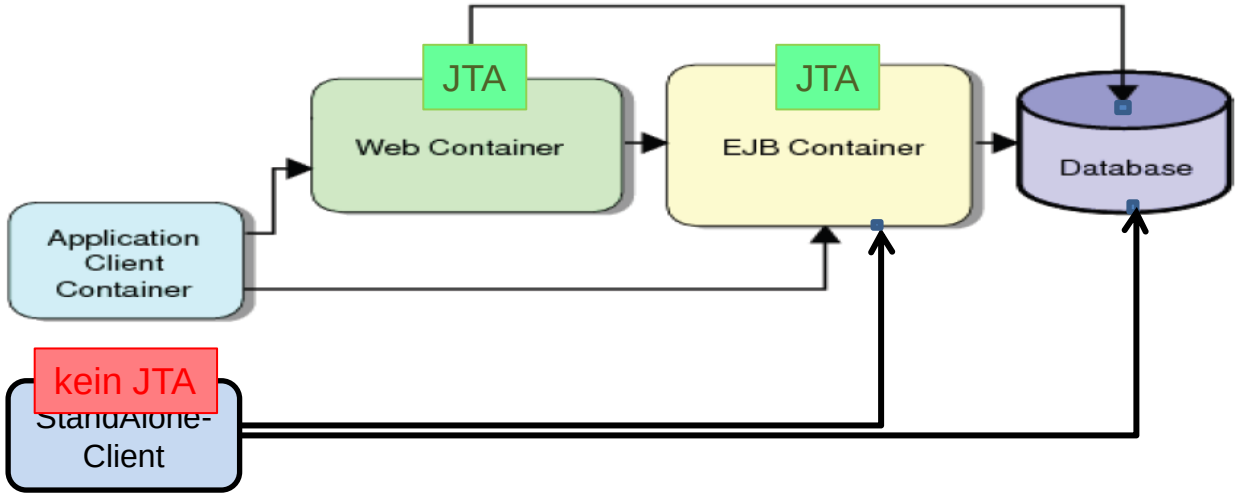


Modul 12
Motivation: Transaktionen über mehrere DBs hinweg

Mehrere Datenbanken sind Teil einer Transaktion



Lösung:
Java Transaction API (JTA)



Modul 12
Motivation: Bean-Managed vs. Container-Managed

```
try {
   utx.begin();
      em.persist( entity1 );
   utx.commit();
} catch( Exception ex ) {
   utx.rollback();
}
```

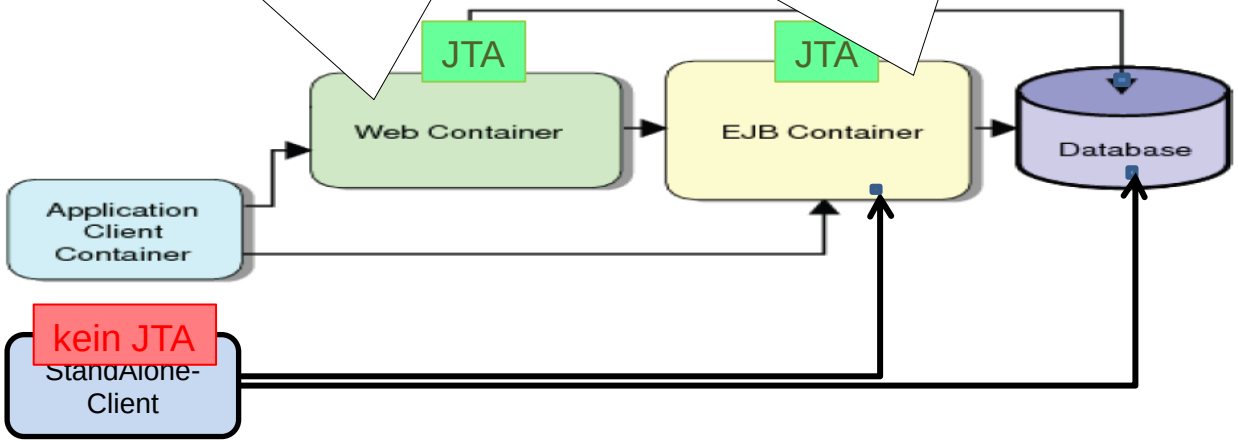
„programmatic“ („bean managed“)

```
@PersistenceContext
private EntityManager em;
...
   em.persist( entity1 );
   em.merge( entity2 );
   em.find( MyEntity.class, id );
```

„declarative“ („container managed“)

```
try {
   utx.begin();
      em.persist( entity1 );
   utx.commit();
} catch( Exception ex ) {
   utx.rollback();
}
```

„programmatic“ („bean managed“)



Modul 12
Comparison of Programmatic and Declarative Transactions

The Java EE application developer's concern is with *scoping* transactions. Scoping can be programmatic (bean-managed transactions) or declarative (container-managed transactions).

	Programmatic	**Declarative**
Operation	Use JTA calls to specify which operations form a single transaction	Specify the transaction properties in the deployment descriptor
Available In	Servlets, JSP components, session beans, and message-driven beans, but not in entity beans	Session beans, message-driven beans, and entity beans, but not in servlets or JSP components

There are three types of transaction management models:

Transaction Model Type	Description
Nested	A transaction consists of subtransactions that run in parallel.
Chained	A transaction consists of subtransactions that run in sequence.
Flat	A transaction cannot be made up of subtransactions. The flat transaction model: • Is the only transaction model supported by the Java EE specification • Is supported by all database vendors • Leads to a very simple API

Using JTA to Scope Transactions Programmatically

The JTA specification is relevant to developers of application servers and database drivers. The code is independent of the transaction infrastructure.

- Obtain a reference to the `UserTransaction` object from the container
- Scope the transactions using the `begin` and `commit` methods
- Fail a transaction using the `rollback` method

Modul 12
UserTransaction Interface with **`begin`**, **`commit`**, and **`rollback`**

Most uses of JTA to scope a transaction have the following basic structure:

```
@Stateful
@TransactionManagement(BEAN)
public class MySessionBean implements MySession {
@Resource UserTransaction ut;

public void method() {
     try {
       ut.begin(); // Transaction starts here
       //... transaction operation 1, 2, 3
       ut.commit(); // Transaction finishes here

     } catch (Exception e) {
       ut.rollback(); // Oops: roll back
     }
```

Declarative or container-managed transaction scoping is the technique of choice in EJB components.

- No coding is necessary, but the transaction scope is affected by the number and sequence of method calls.
- In many cases, you are required to do nothing. All of the methods of CMT EJB components have a default of the `REQUIRED` transaction attribute.

```
import javax.ejb.*;

@TransactionAttribute(TransactionAttributeType.REQUIRED)
public void someMethod () {...}
```

Modul 12
Implementing a Container-Managed Transaction Policy

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

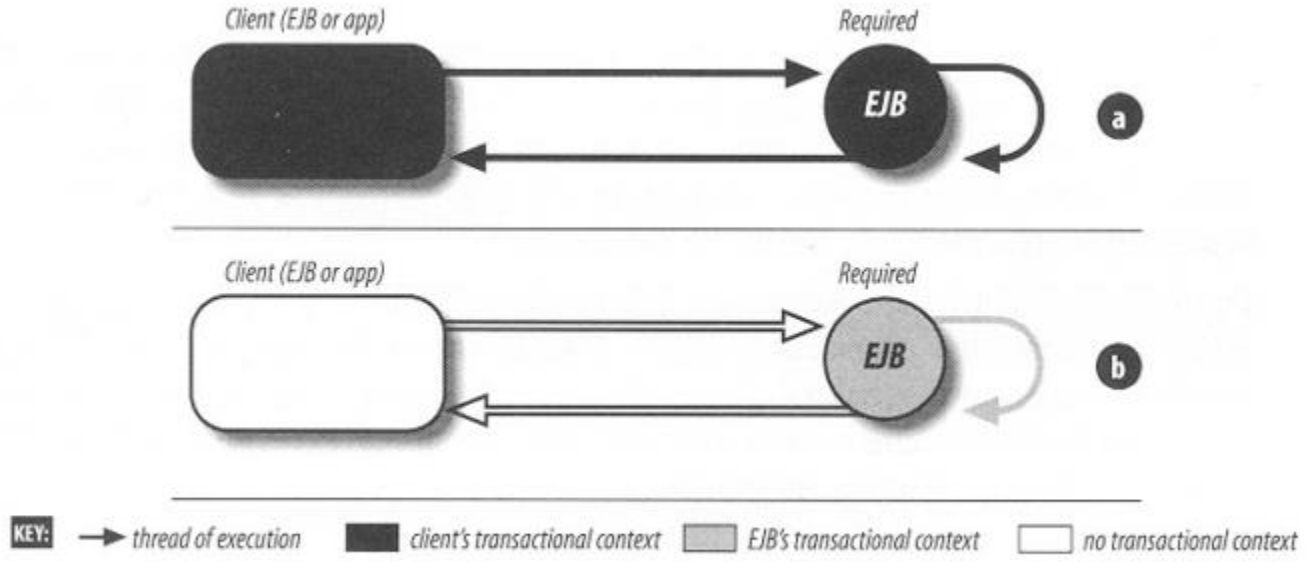
```
@TransactionAttribute(NOT_SUPPORTED)
@Stateful
public class TransactionBean implements Transaction {
...
    @TransactionAttribute(REQUIRES_NEW)
    public void firstMethod() {...}

    @TransactionAttribute(REQUIRED)
    public void secondMethod() {...}

    public void thirdMethod() {...}

    public void fourthMethod() {...}
}
```

Modul 12
Control the Container's Behavior Using Transaction Attributes

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

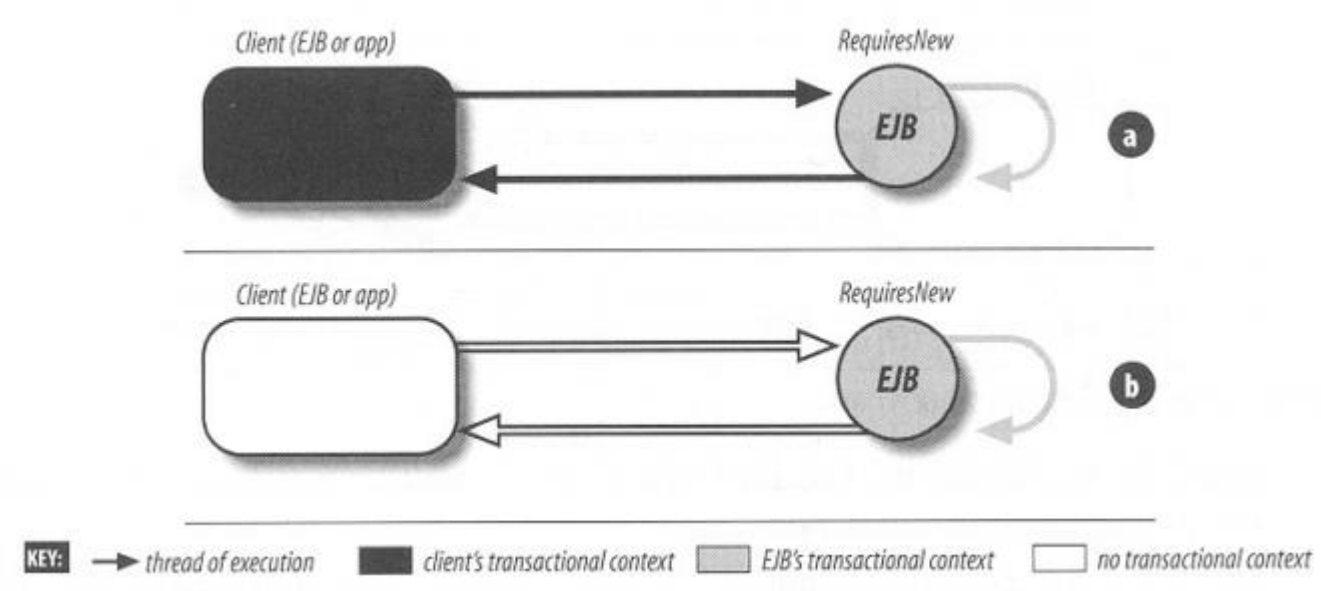
Attribute	Effect
`Required`	The method becomes part of the caller's transaction. If the caller does not have a transaction, the method runs in its own transaction.
`RequiresNew`	The method always runs in its own transaction. Any existing transaction is suspended.
`NotSupported`	The method never runs in a transaction. Any existing transaction is suspended.
`Supports`	The method becomes part of the caller's transaction if there is one. If the caller does not have a transaction, the method does not run in a transaction.
`Mandatory`	It is an error to call this method outside of a transaction.
`Never`	It is an error to call this method in a transaction.

Modul 12
Transaction Attribute: Required



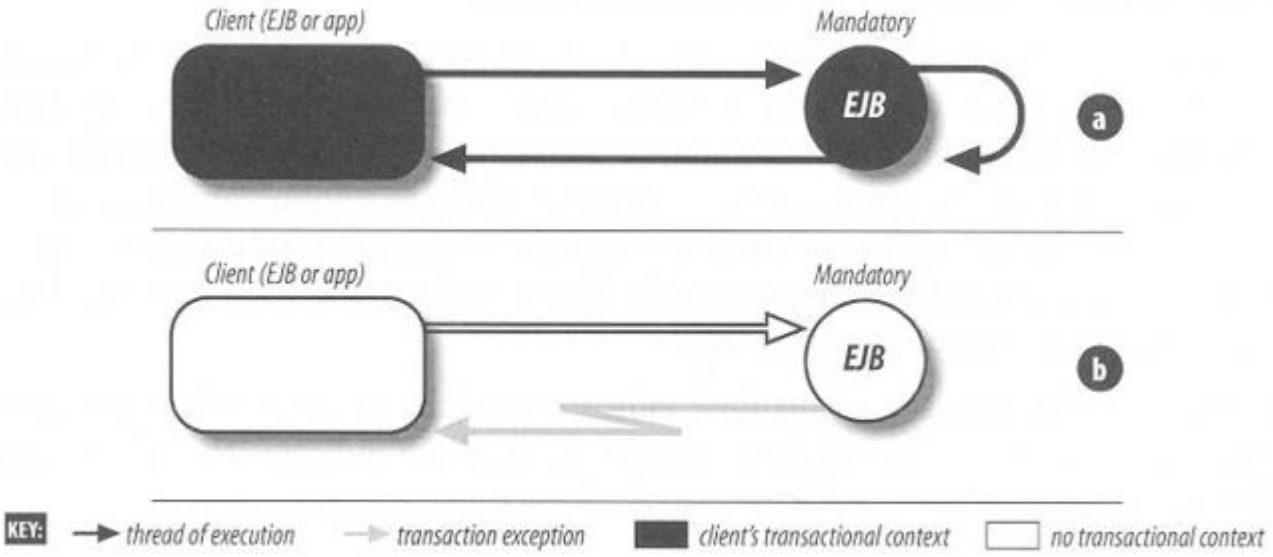
Quelle: "Enterprise JavaBeans 3.0" Bill Burke and Richard Monson-Haefel, 5th edition, O'Reilly, 2006

Modul 12
Transaction Attribute: Requires_New



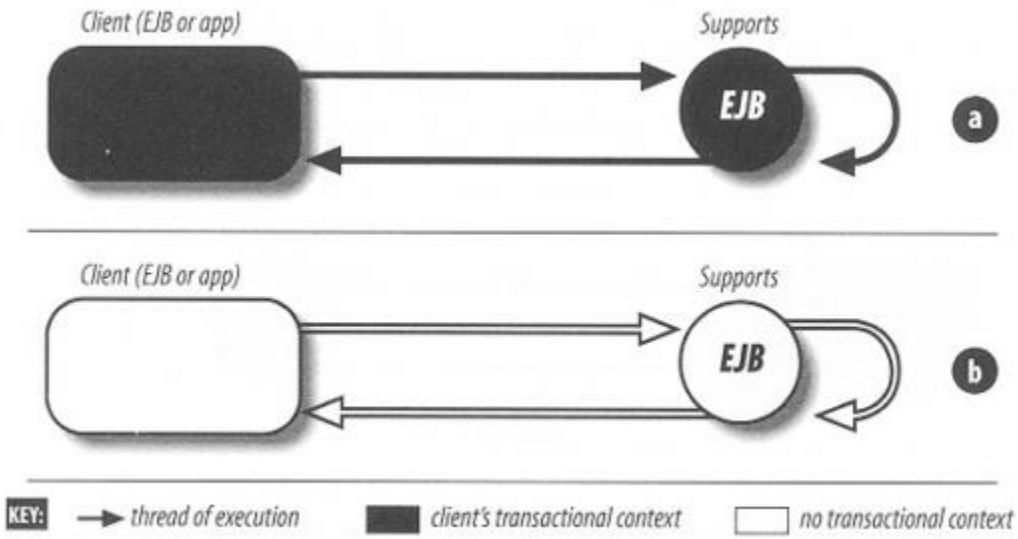
Quelle: "Enterprise JavaBeans 3.0" Bill Burke and Richard Monson-Haefel, 5th edition, O'Reilly, 2006

Modul 12
Transaction Attribute: Mandatory



Quelle: "Enterprise JavaBeans 3.0" Bill Burke and Richard Monson-Haefel, 5th edition, O'Reilly, 2006

Modul 12
Transaction Attribute: Supports

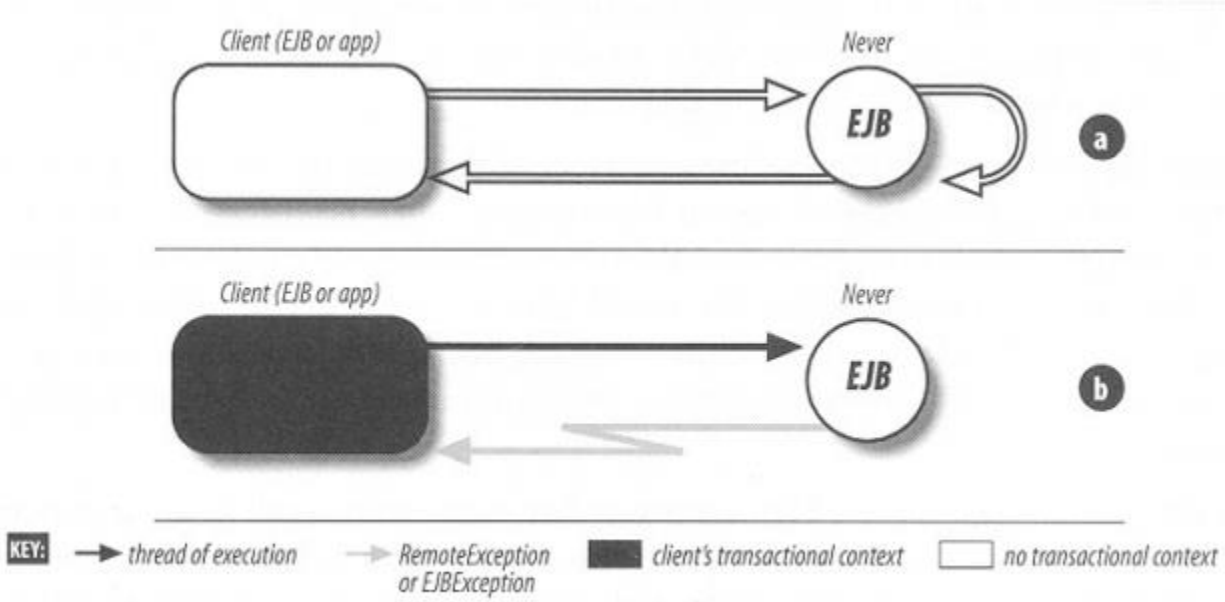


Quelle: "Enterprise JavaBeans 3.0" Bill Burke and Richard Monson-Haefel, 5th edition, O'Reilly, 2006

Modul 12
Transaction Attribute: Not_Supported

Quelle: "Enterprise JavaBeans 3.0" Bill Burke and Richard Monson-Haefel, 5th edition, O'Reilly, 2006

Modul 12
Transaction Attribute: Not_Supported

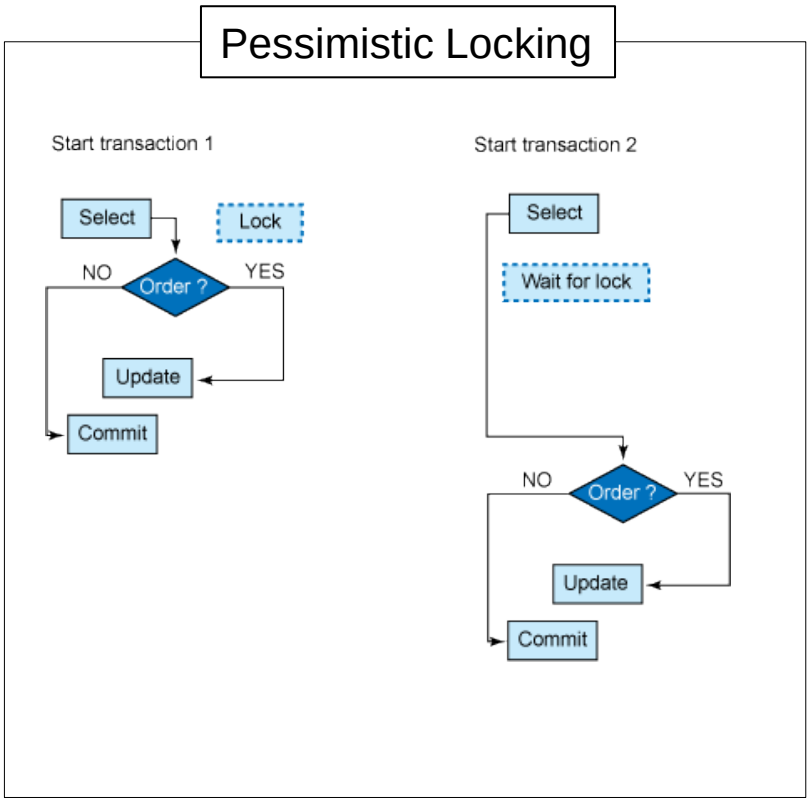


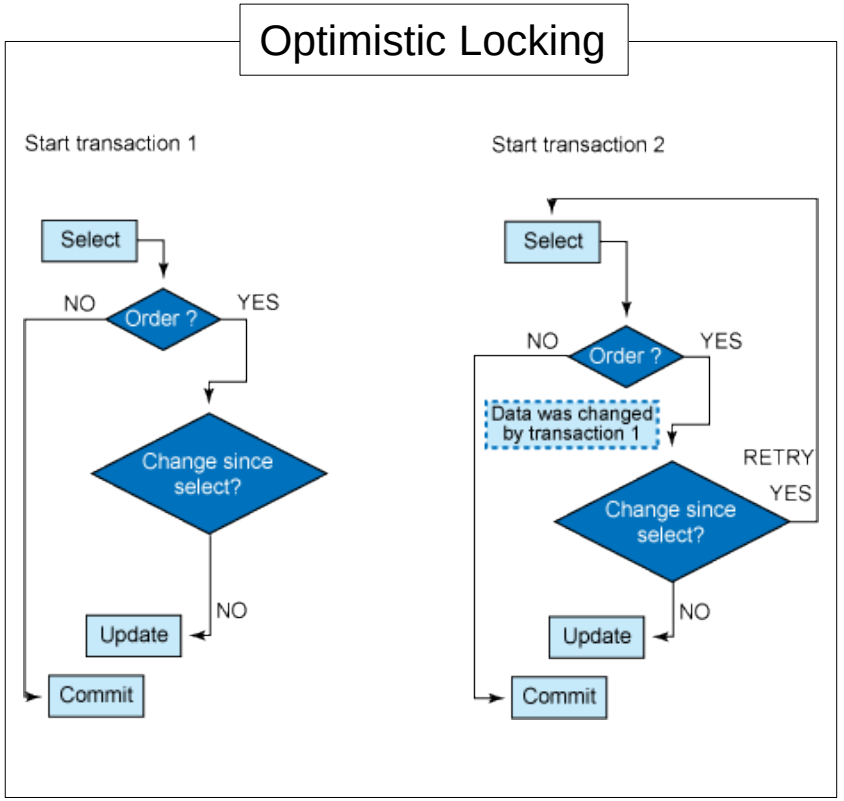
Quelle: "Enterprise JavaBeans 3.0" Bill Burke and Richard Monson-Haefel, 5th edition, O'Reilly, 2006

Entity components must be synchronized with the underlying database at least once per transaction.

- Entity classes do not declare transactions, they inherit the transaction of the calling component.
- Entity classes can use optimistic locking. When a transaction commits entity data, the transaction can fail to commit because the data was modified by a concurrently running transaction.

Modul 12
Optimistic Locking and Versioning





Quelle: http://www.ibm.com/developerworks/data/library/techarticle/dm-0801schuetz/

Modul 12
Implementing Versioning

A basic implementation of versioning can use an integer database column in the same table used for entity data. The field or property used for the version should not be written to.

```
@Version
@Column(name = "VERSION")
private int version;

public int getVersion() {
    return version;
}
```

When an entity is merged and its version does not match the stored version, the merge fails with a `javax.persistence.OptimisticLockException`.

Effect of Exceptions on Transaction State

Exception handling differs in programmatic and declarative transaction scoping.

- With programmatic transaction scoping, you are responsible for catching exceptions and failing transactions.
- With declarative transaction scoping:
  - The container automatically fails the current transaction if it catches a runtime exception. `OptimisticLockException` is a runtime exception.
  - An EJB component can use the `EJBContext` object or subclass to check whether the container has failed a transaction.

Using the **`EJBContext`** Object to Check and Control Transaction State

The `EJBContext` object that is injected in EJB components at runtime provides two methods that are useful for transaction management and testing:

- `getRollbackOnly()`
- `setRollbackOnly()` (only runtime-exceptions call rollback automatically)

```
@Stateful public class BankBean
    implements Bank, Serializable {
    @Resource private SessionContext context;
    ...
    @TransactionAttribute
    (TransactionAttributeType.REQUIRED)
    public void transfer(...) throws BankException {
        try {
            ...
        }
        catch (BankException e) {
            context.setRollbackOnly();
            throw e;
    }
}
```

„application“ exception

Modul 12
Klausur

Modul 12
Klausur

Im folgenden Szenario wird ein Aufruf-Stack dargestellt. Dabei besagt eine Einrückung, dass die Methode von der übergeordneten Methode aufgerufen wird. Beispiel:

```
methodA()
        methodB()
        methodC()
```

In diesem Beispiel ruft methodA die Methode methodB und danach die Methode methodC auf.

Aufruf-Stack:

```
1   No transaction          Controller.addNewOrder( ... )
2   No transaction          Order newOrder = new Order();
3   Required                CRM_Mgr.addNewOrder(newOrder);
4   RequiresNew                   newOrder.Log();
5   Not Supported                 CRM_Mgr.upDateAll(NewOrder);
6                                       CRM_Mgr.out();
7                                 newOrder.insert();
```

Welche Methoden werden zurückgesetzt (rollback), wenn eine System-Exception auftritt. Wenn keine Methode zurückgesetzt wird, schreiben Sie "-".

System-Exception in	Rollback der Methoden in Zeile
Zeile 4	**4**
Zeile 5	**-**
Zeile 6	**6**
Zeile 7	**7, 3**

Modul 12
Klausur

```
1   No transaction  Controller.transferMoney()
2   Required           BankMgr.transferMoney()
3                         Customer cust1 = em.find(Customer.class, id1)
4                         Customer cust2 = em.find(Customer.class, id2)
5                         cust1.setBalance()
6                         cust2.setBalance()
```

1. Which methods would be rolled back if the method started on line 2 threw a system exception *after* running lines 3, 4, 5, and 6 all successfully?

Lines 2 – 6
(Die Methoden 2-6 sind Teil einer einzigen Transaktion)