Spring Transactions

CS544: Enterprise Architecture

Spring Transactions

- In this module we will first define what a Transaction is (constant), and what our general configuration options are for transactions (variable).
- After which we will look at how to configure transactions using Spring annotations or XML.

Transactions

- A Transaction is a unit of work that is:
  - **ATOMIC:** The transaction is considered a single unit, either the entire transaction completes, or the entire transaction fails.
  - **CONSISTENT:** A transaction transforms the database from one consistent state to another consistent state
  - **ISOLATED:** Data inside a transaction can not be changed by another concurrent processes until the transaction has been committed
  - **DURABLE:** Once committed, the changes made by a transaction are persistent

Transactional Choices

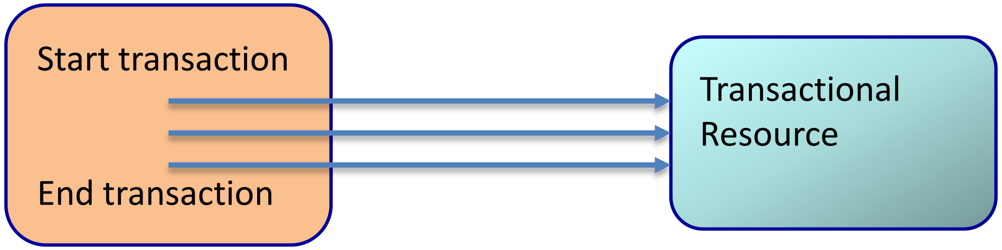
- Local or Global Transactions
- Transaction Isolation Level
- Transaction Propagation

Spring Transactions:

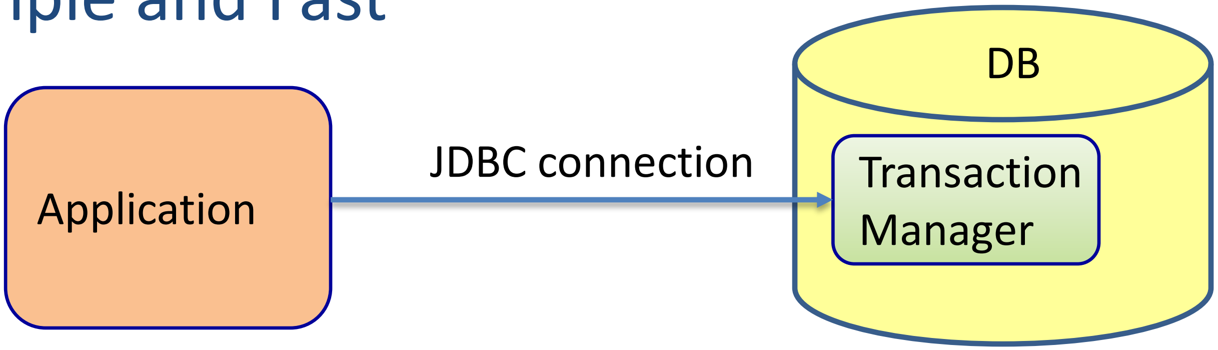
GLOBAL TRANSACTIONS

Local Transactions

- So far we've only considered local transactions
  - Transactions that use a single transactional resource (database, message bus)

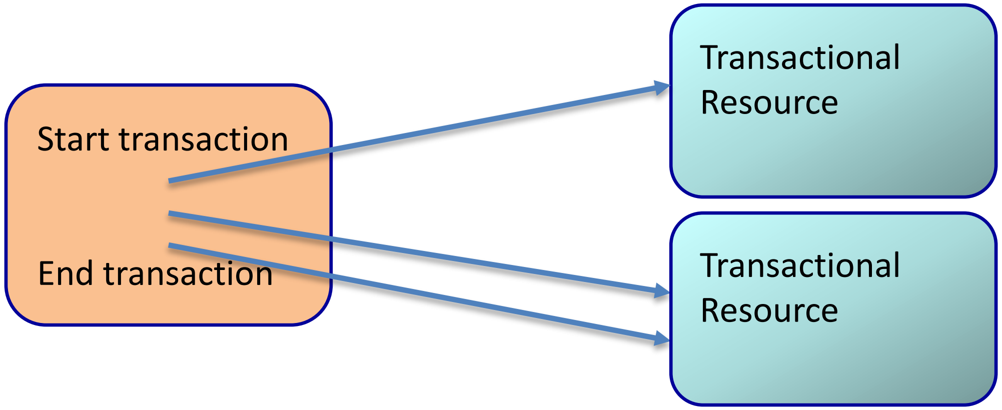


- These transactions are managed by the DB
  - Simple and Fast



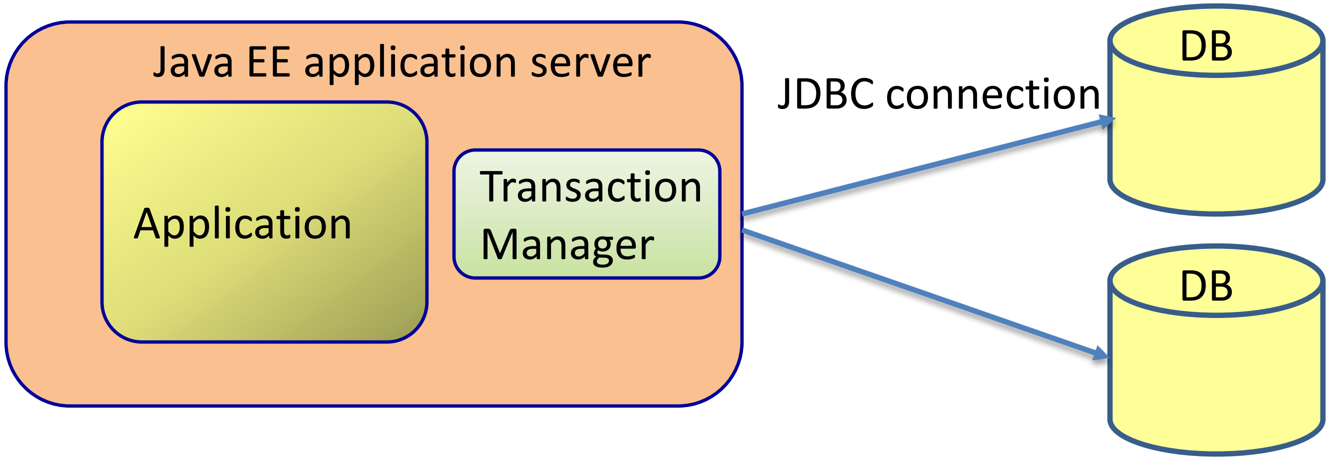
Global Transactions

- Global Transactions are transactions that span multiple transactional resources
  - Such as databases or message busses
  - More common in enterprise applications
  - Also called XA transactions



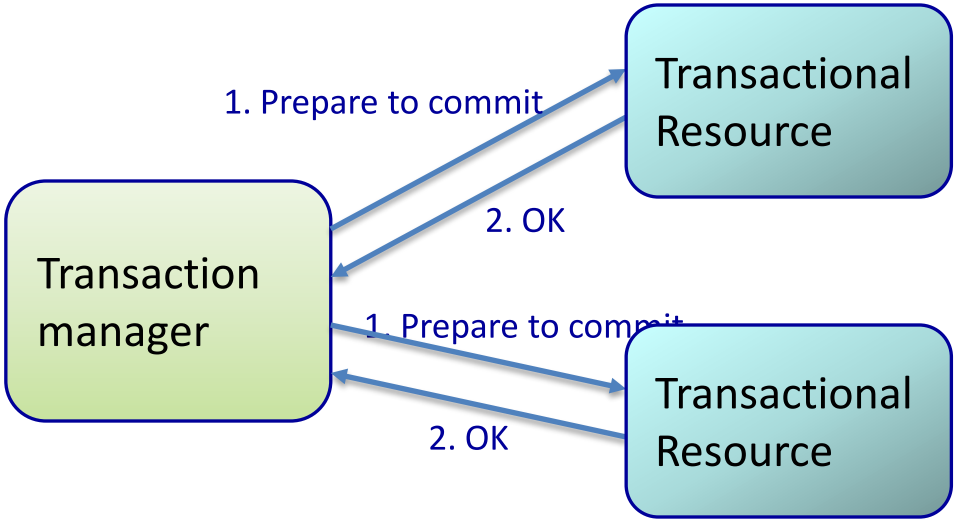
Transaction Manager

- Global Transactions have to be managed on the application side (to coordinate resources)
  - Generally done by a Transaction Manager
    - Standard Java Transaction API (JTA) interface
    - Required part of Java EE application servers
    - Stand Alone JTA implementations also exist*



2 phase commit (Success)

- Phase 1



Transaction manager

1. Prepare to commit

2. OK

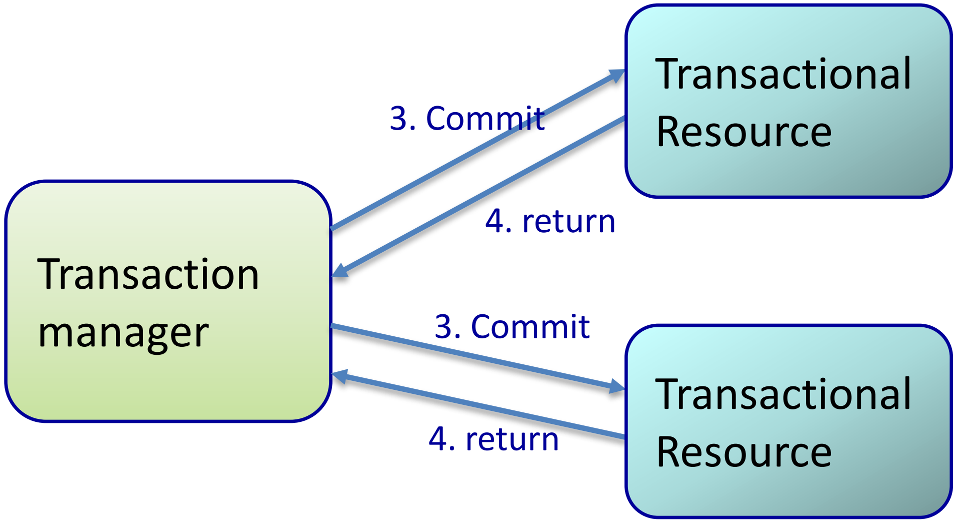
Transactional Resource

1. Prepare to commit

2. OK

Transactional Resource

- Phase 2



Transaction manager

3. Commit

4. return

Transactional Resource

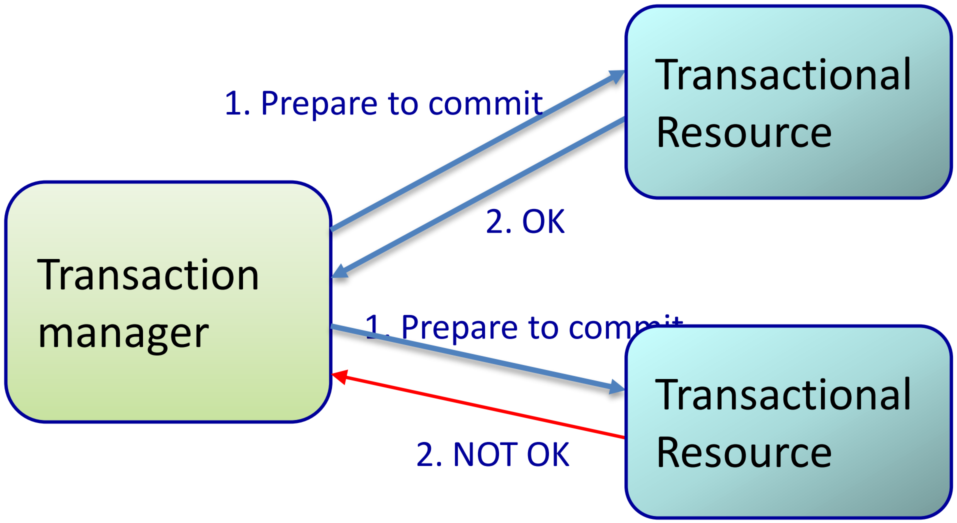
3. Commit

4. return

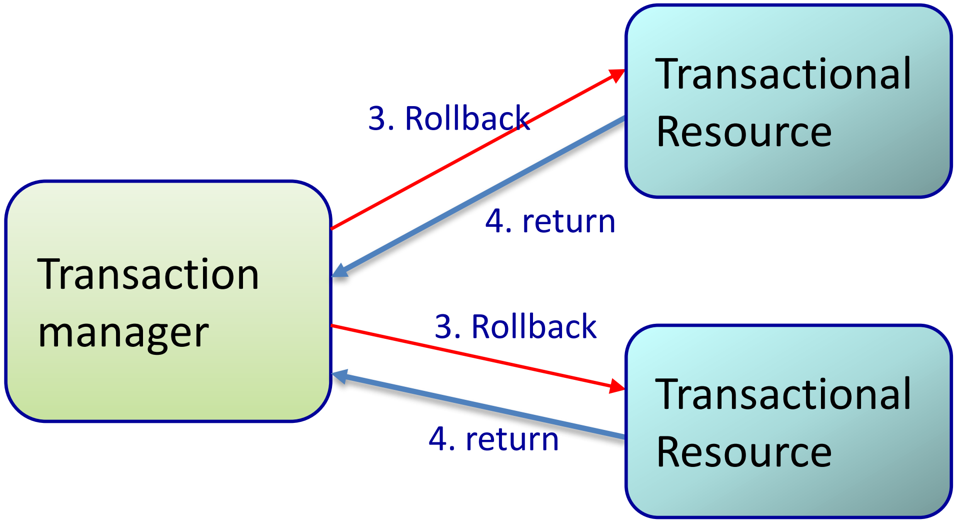
Transactional Resource

2 phase commit (Failure)

- Phase 1



- Phase 2



Spring Transactions:

ISOLATION LEVELS

Characteristics of XA Transactions

- 2 phase commit:
  - does not guarantee that nothing can go wrong
  - Is slow – multiple remote connections
- Transactional resources become dependent on each other
  - Need to keep locks until ALL resources are finished
  - Thereby also decreasing performance
- Price you pay for coordinating multiple resources

Isolation Levels

- Proper full isolation is expensive to produce in a multi user environment
  - Isolation is often relaxed to increase db speed
  - ANSI SQL defines four isolation levels

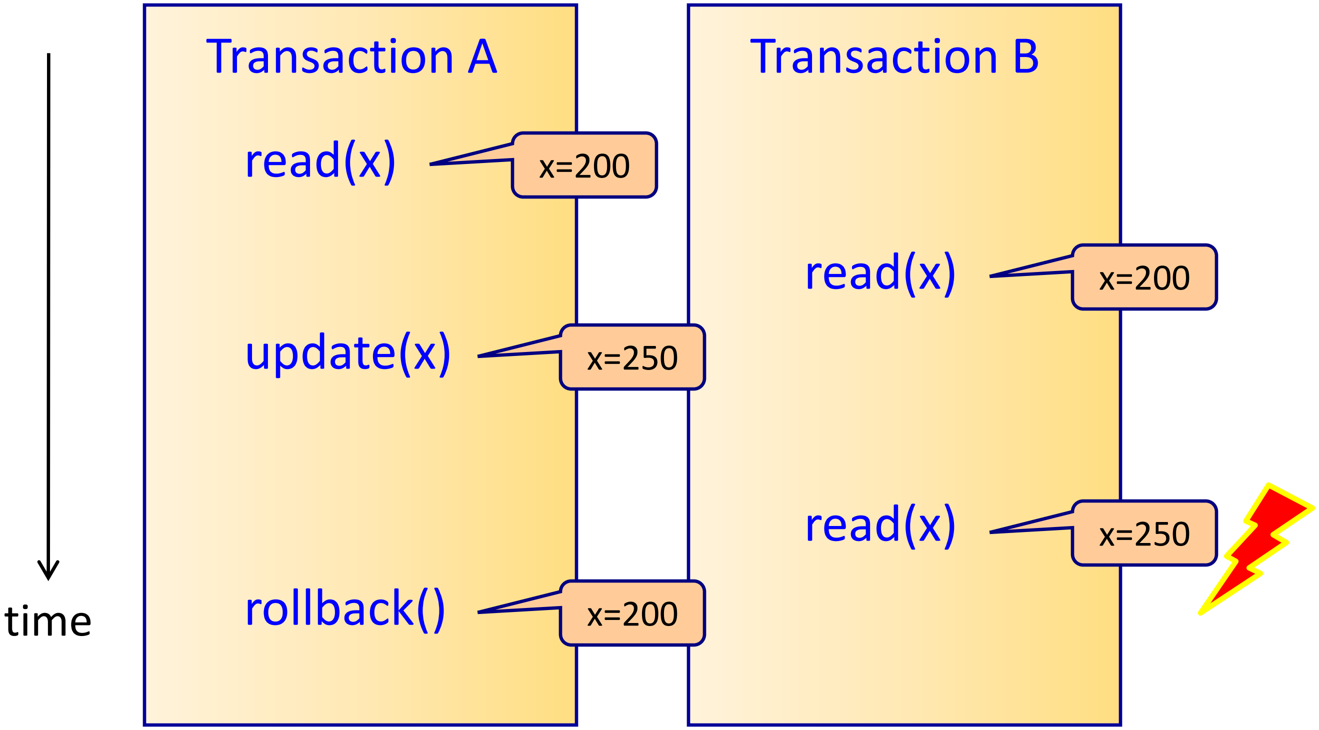
Read Uncommitted, Read Committed, Repeatable Read, and Serializable

Weaker, faster to Stronger, slower

- Most dbs default to Read Committed isolation
  - Only Serializable fully isolates a transaction from concurrency issues

Read Uncommitted

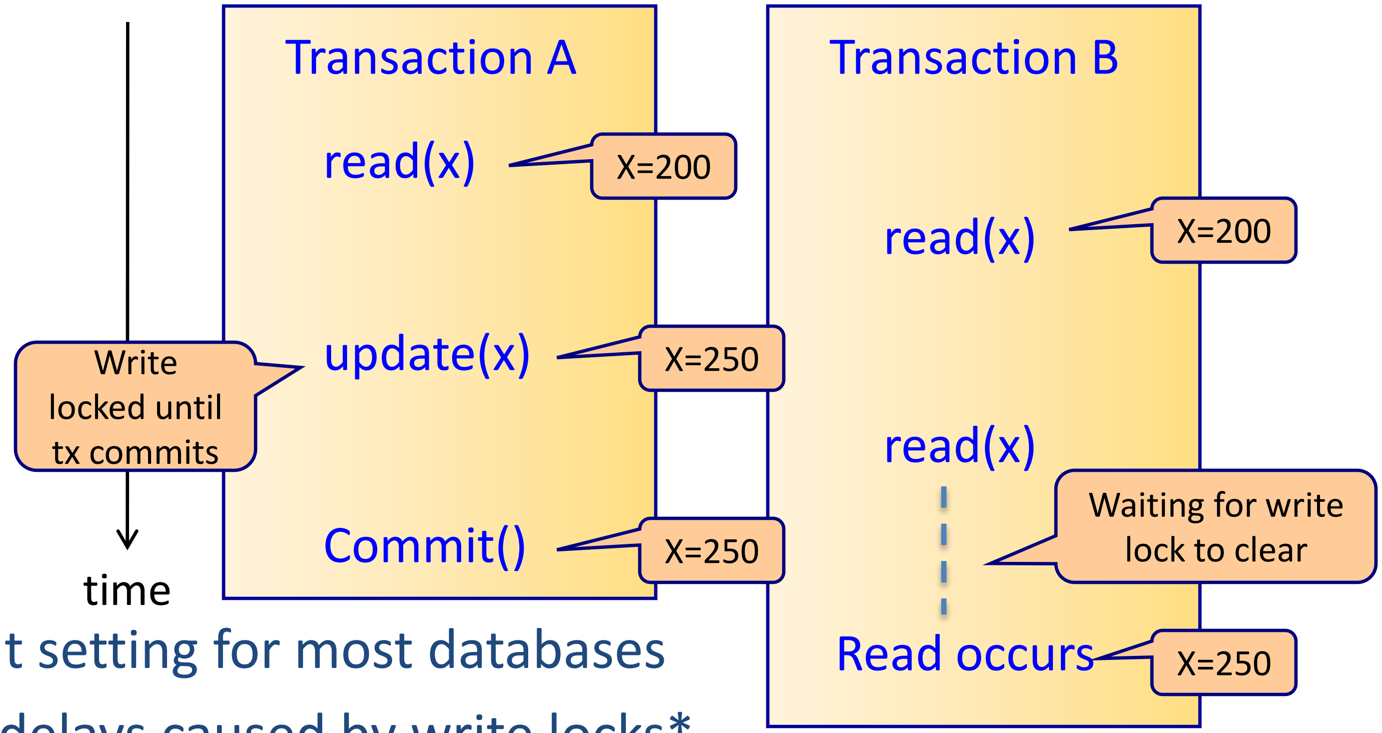
- Transactions can read uncommitted updates made by other transactions



- Violates the ACID properties
- Not supported by many database vendors (Oracle)
- Do not use this level of isolation in a multithreaded system

Read Committed

- Allows multiple transactions to access the same data, but hides non-committed data from other transactions



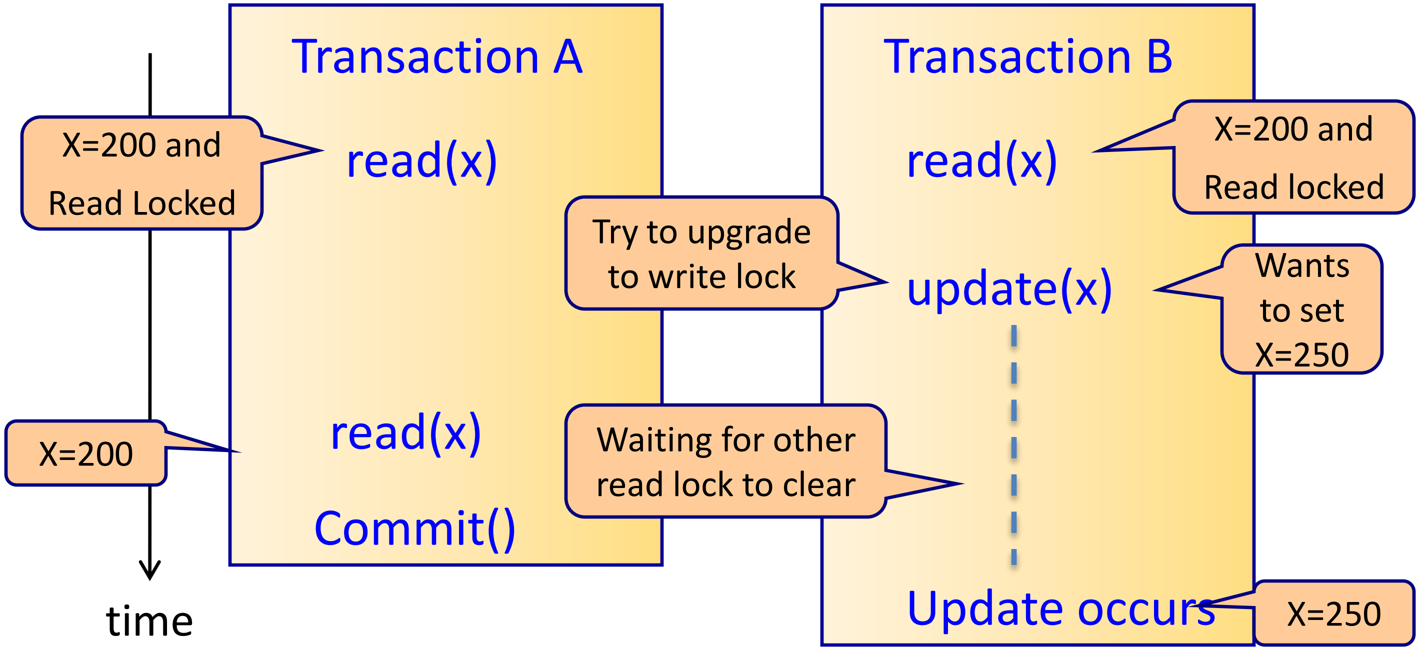
- Default setting for most databases
- Some delays caused by write locks*
- Unrepeatable reads problem

Isolation Loss

- Read Committed can read data that is newly committed during the transaction - can cause:
  - Unrepeatable Read: If a row is read twice during a transaction, the second read might give different data because of a concurrent update
- Repeatable read (see next slide) solves this problem but can still have:
  - Phantom Read: If the same select is executed twice during a transaction the second result set might include more rows than the first due to concurrent inserts

Repeatable Read

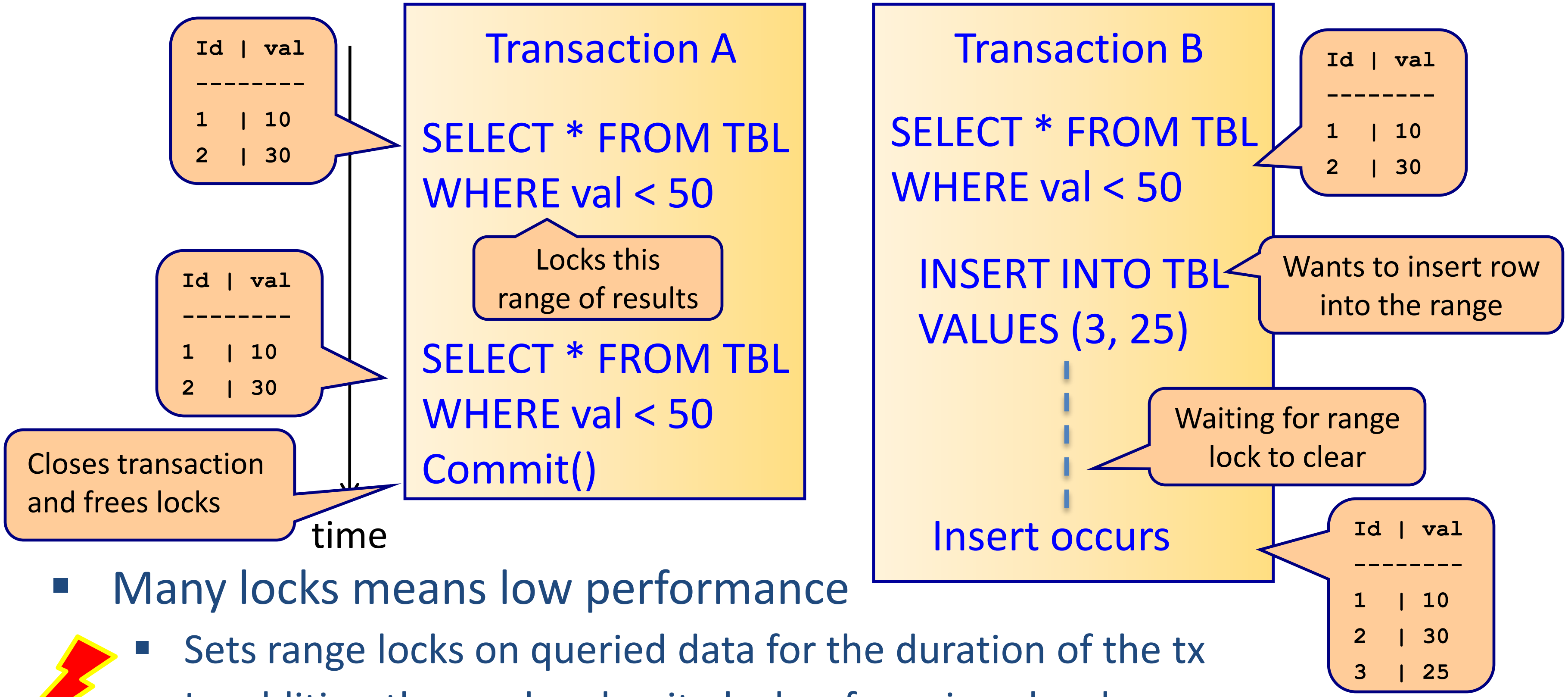
- Once a value is read within a transaction, all subsequent reads will return the same result



- Uses both read and write locks for the duration of the tx
- Still able to get phantom read (insertion) problems

Serializable

- Ensures complete isolation from change, as if all transactions were executed in a serial manner.
  - Concurrent reads are still possible



```
Id | val
--------
1  | 10
2  | 30
```

Transaction A

SELECT * FROM TBL WHERE val < 50

Locks this range of results

```
Id | val
--------
1  | 10
2  | 30
```

SELECT * FROM TBL WHERE val < 50

Commit()

Closes transaction and frees locks

time

Transaction B

SELECT * FROM TBL WHERE val < 50

```
Id | val
--------
1  | 10
2  | 30
```

INSERT INTO TBL VALUES (3, 25)

Wants to insert row into the range

Waiting for range lock to clear

Insert occurs

```
Id | val
--------
1  | 10
2  | 30
3  | 25
```

- Many locks means low performance
  - Sets range locks on queried data for the duration of the tx
  - In addition the read and write locks of previous levels

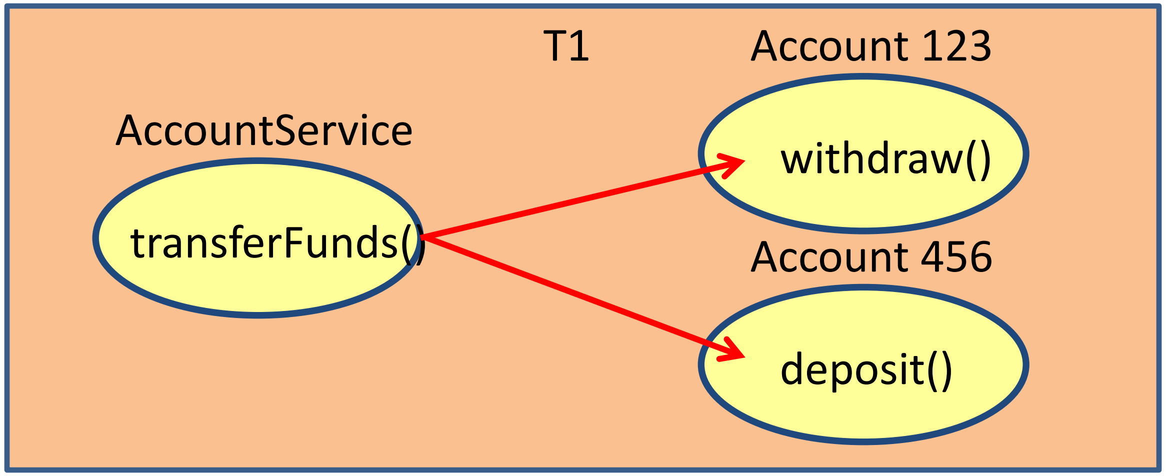
Reminder: Snapshot isolation

Spring Transactions:

TRANSACTION PROPAGATION

Transaction Propagation

- Transaction propagation defines the interaction between transactions and method calls
  - Normally any method called between a beginTransaction() and commit() is part of the TX
  - A TX created for transferFunds() will automatically propagates to both withdraw() and deposit()



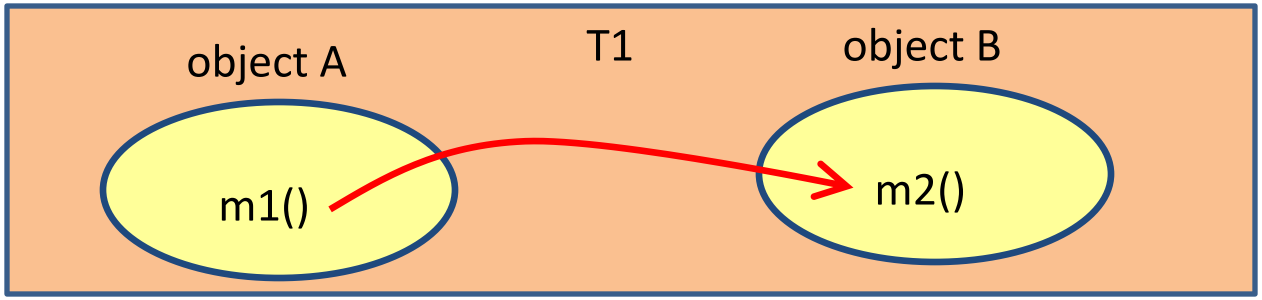
Propagation

- There are seven propagation options:
  - REQUIRED join - creates new
  - REQUIRES_NEW created new - created new
  - MANDATORY join - thrown exception
  - NESTED nested - created new
  - SUPPORTS join - no creates
  - NOT_SUPPORTED no created new - no creates new
  - NEVER thrown exception - no created one

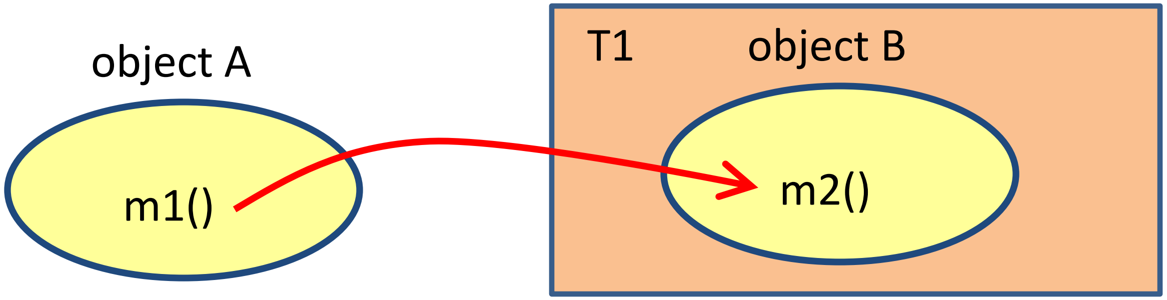
- You can also specify isolation, timeout, rollback and read-only requirements

propagation: REQUIRED

- If the calling method m1() runs in a transaction T1, then method m2() joins the same transaction T1

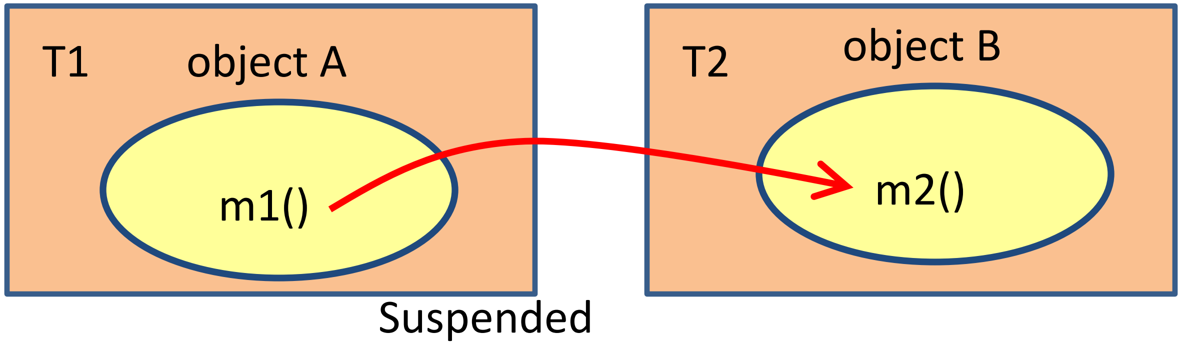


- If the calling method m1() does not run in a transaction , then method m2() runs in a new created transaction T1

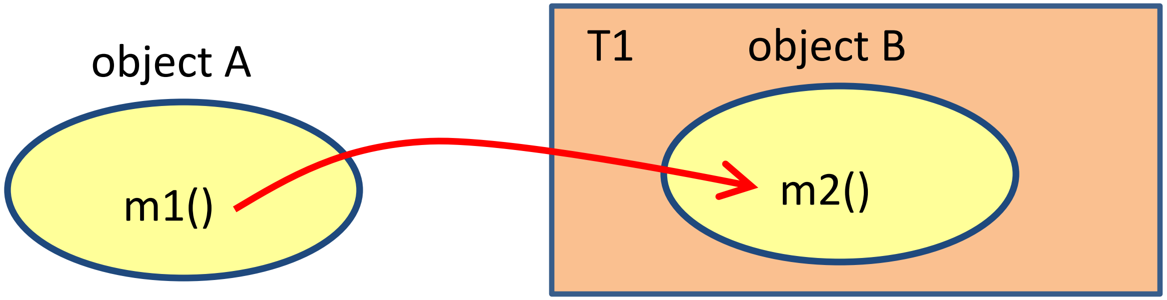


propagation: REQUIRES_NEW

- If the calling method m1() runs in a transaction T1, then method m2() runs in a new created transaction T2

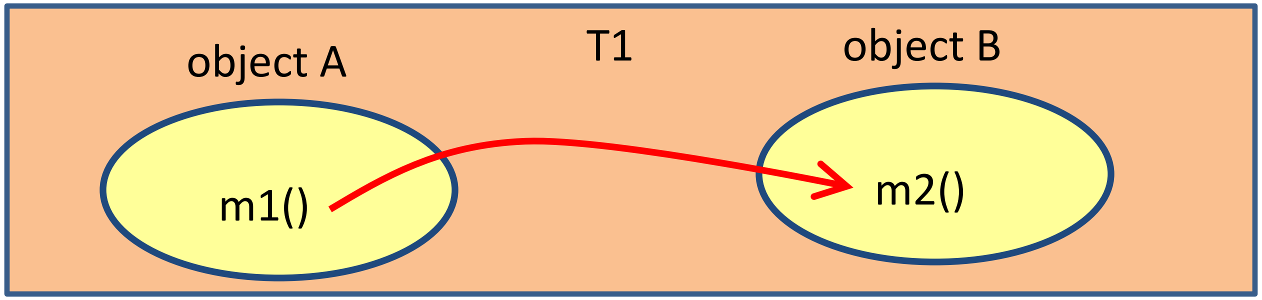


- If the calling method m1() does not run in a transaction , then method m2() runs in a new created transaction T1

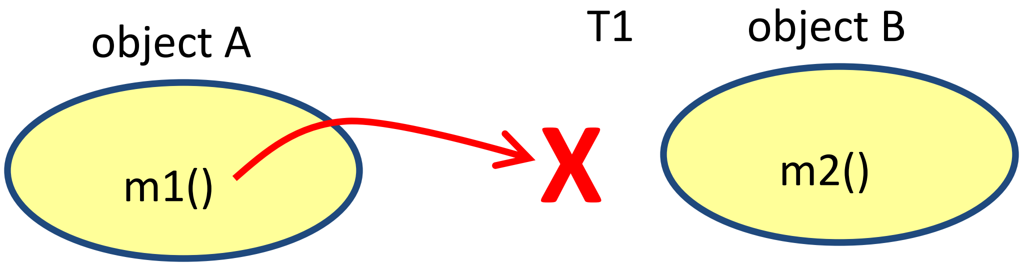


propagation: MANDATORY

- If the calling method m1() runs in a transaction T1, then method m2() joins the same transaction T1

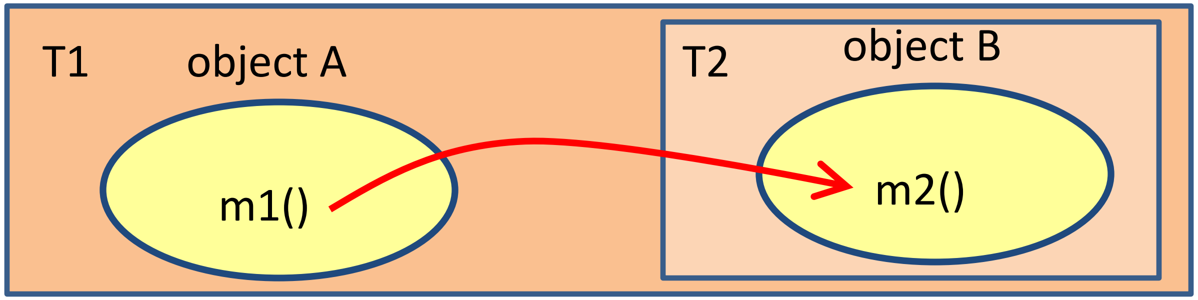


- If the calling method m1() does not run in a transaction , an exception is thrown

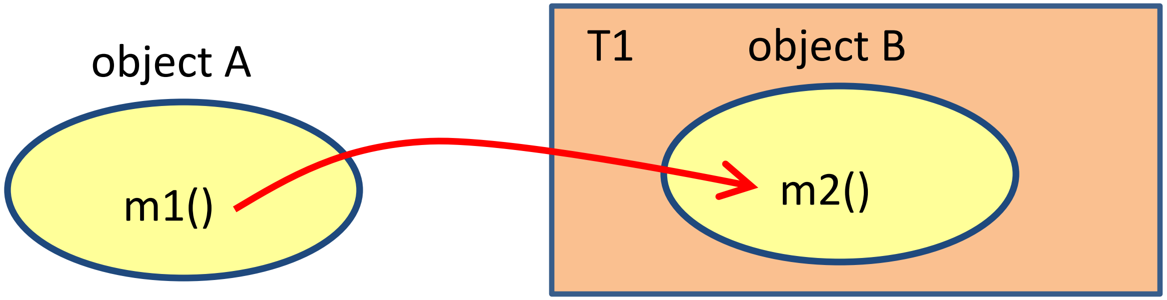


propagation: NESTED

- If the calling method m1() runs in a transaction T1, then method m2() runs in a nested transaction T2

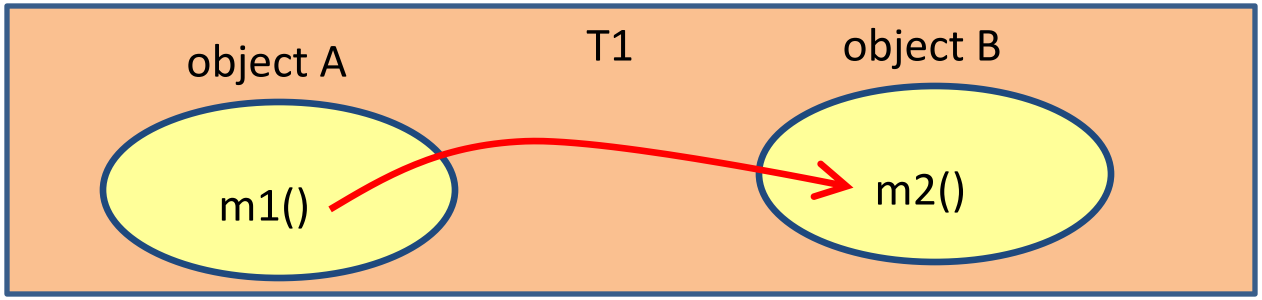


- If the calling method m1() does not run in a transaction , then method m2() runs in a new created transaction T1

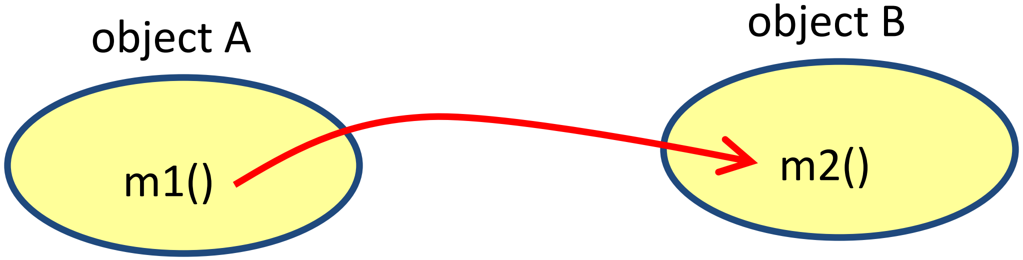


propagation: SUPPORTS

- If the calling method m1() runs in a transaction T1, then method m2() joins the same transaction T1

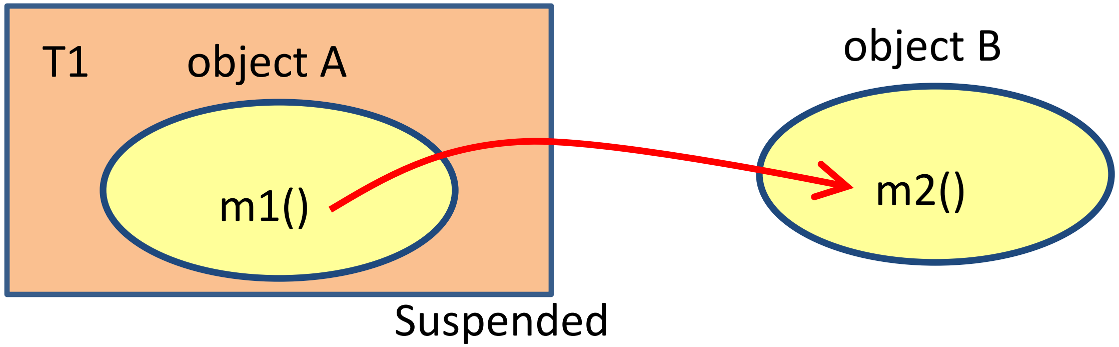


- If the calling method m1() does not run in a transaction , then method m2() also does not run within a transaction

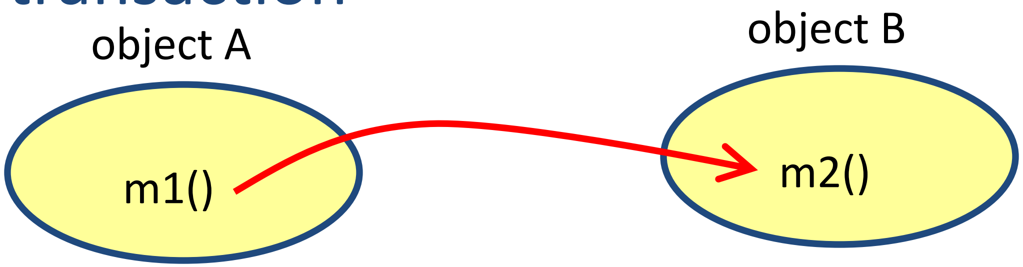


propagation: NOT_SUPPORTED

- If the calling method m1() runs in a transaction T1, then method m2() does not run within a transaction.

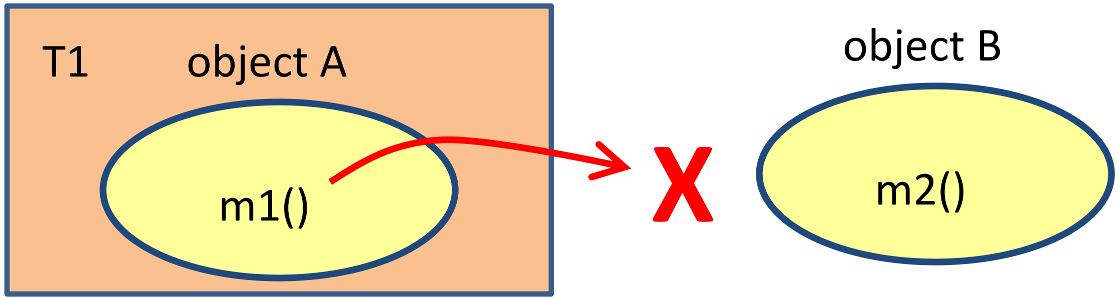


- If the calling method m1() does not run in a transaction , then method m2() also does not run within a transaction

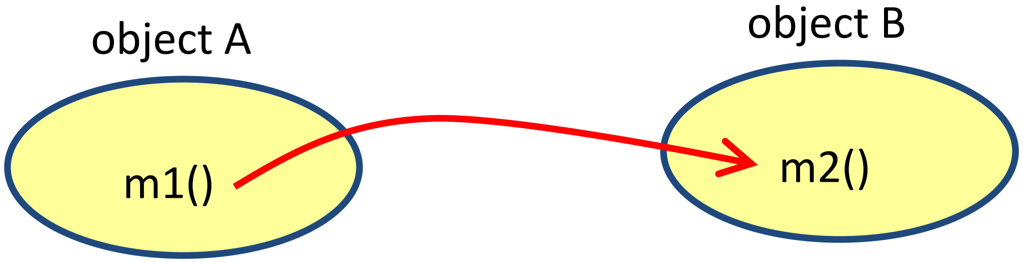


propagation: NEVER

- If the calling method m1() runs in a transaction T1, an exception is thrown



- If the calling method m1() does not run in a transaction , then method m2() also does not run within a transaction



Transaction Propagation

- What propagation options you have are very dependent on your transaction manager.
  - The default REQUIRED propagation, is of supported by every transaction manager
  - Propagation options that require transaction suspension or nesting are more problematic

Spring Transactions:

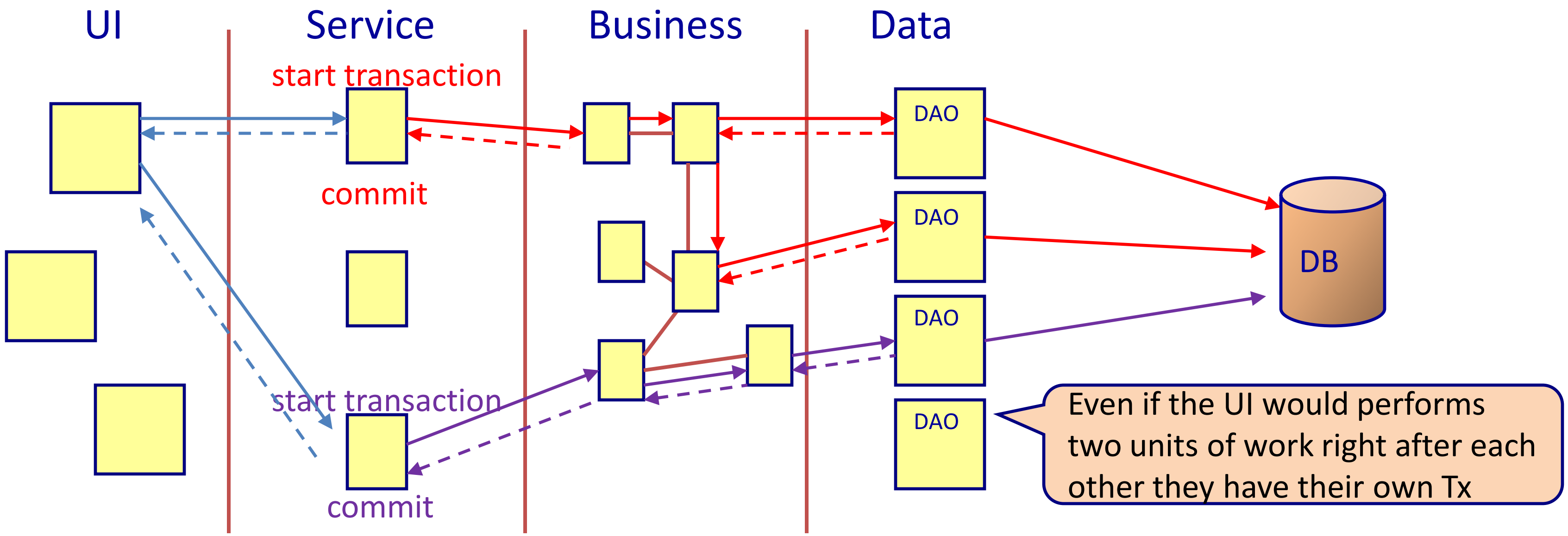
SPRING TRANSACTION SUPPORT

Spring Transaction Support

- Spring is not a transaction manager
  - We still need a transaction manager
    - JDBC transaction manager
    - Hibernate transaction manager
    - XA transaction manger (JTA)
- Spring provides an abstraction for transaction management
  - You declare how the transactions should be managed
  - Spring works with the underlying transaction manger

Transaction Demarcation

- The transactional demarcation is the specification of the transactional boundaries
- This is typical at the service level
  - Multiple DAO’s can be involved in one transaction
  - Creating a transaction per unit of work



Even if the UI would performs two units of work right after each other they have their own Tx

Programmatic Demarcation

- Hibernate programmatic transaction demarcation

```
public class CustomerService {
  private CustomerDAO customerDao = new CustomerDAO();
  private AddressDAO addressDao = new AddressDAO();
  private CreditCardDAO ccDao = new CreditCardDAO();
  private SessionFactory sf = HibernateUtil.getSessionFactory();

  public void addNewCustomer(Customer cust, Address shipAddr, CreditCard cc,
                             Address billAddr) {
    cc.setAddress(billAddr);
    cust.setShipAddress(shipAddr);
    cust.setCreditCard(cc);

    Transaction tx = sf.getCurrentSession().beginTransaction();
    addressDao.create(shipAddr);
    addressDao.create(billAddr);
    ccDao.create(cc);
    customerDao.create(cust);
    tx.commit();
  }

  ...
```

Programmatically beings the transaction

Transaction is automatically propagated to the enclosed methods

Programmatically ends the transaction

Spring Declarative Demarcation

```
public class CustomerService {
  private CustomerDAO customerDao;
  private AddressDAO addressDao;
  private CreditCardDAO ccDao;

  public CustomerService() {}
  public void setCustomerDAO(CustomerDAO customerDao) { this.customerDao = customerDao; }
  public void setAddressDAO(AddressDAO addressDao) { this.addressDao = addressDao; }
  public void setCredit                     cDao) { this.ccDao = ccDao; }


  @Transactional(propagation=Propagation.REQUIRED)
  public void addNewCustomer(Customer cust, Address shipAddr, CreditCard cc,
          Address billAddr) {

    cc.setAddress(billAddr);
    cust.setShipAddress(shipAddr);
    cust.setCreditCard(cc);

    addressDao.create(shipAddr);
    addressDao.create(billAddr);
    ccDao.create(cc);
    customerDao.create(cust);
  }

  ...
```

REQUIRED is the default, and therefore optional

Simply declare that a transaction is needed for this method

Spring takes care of opening and closing the transaction

Transaction propagates to called methods as normal

Using Annotation Configuration

- Configuring Spring to use annotations for transaction demarcation

```
@Transactional(propagation=Propagation.REQUIRED)
public class AddressDAO {
  private SessionFactory sf;

  @Transactional(propagation=Propagation.SUPPORTS)
  public void setSessionFactory(SessionFactory sf) {
    this.sf = sf;
  }

  ...
```

1) Annotate the desired classes and or methods

```
<beans ...>
  ...

  <tx:annotation-driven transaction-manager="txManager" proxy-target-class="true"/>

  <bean id="txManager" class="org.springframework.orm.hibernate4.HibernateTransactionManager">
    <property name="sessionFactory" ref="sessionFactory" />
  </bean>

</beans>
```

2) Tell spring to look for @Transactional annotations

3) Specify the transaction manager

Class Annotations

```
@Transactional(propagation=Propagation.REQUIRED)
public class AddressDAO {
  private SessionFactory sf;

  @Transactional(propagation=Propagation.SUPPORTS)
  public void setSessionFactory(SessionFactory sf) {
    this.sf = sf;
  }

  public void create(Address addr) {
    sf.getCurrentSession().persist(addr);
  }

  public Address get(int id) {
    return (Address) sf.getCurrentSession().get(Address.class, id);
  }

  public void update(Address addr) {
    sf.getCurrentSession().saveOrUpdate(addr);
  }

  public void delete(Address addr) {
    sf.getCurrentSession().delete(addr);
  }
}
```

Annotating a class specifies that all its methods should be transactional

You can also add method level annotations to specify exceptions

Now require a transaction to be executed

Isolation

- You can also specify isolation requirements with the isolation property

```
@Transactional(propagation=Propagation.REQUIRED, isolation=Isolation.READ_COMMITTED)
public class AddressDAO {
  private SessionFactory sf;

  ...
```

Both with annotations

```
  <tx:advice id="daoTxAdvice" transaction-manager="txManager">
    <tx:attributes>
      <tx:method name="set*" propagation="SUPPORTS"/>
      <tx:method name="*" propagation="REQUIRED" isolation="READ_COMMITTED"/>
    </tx:attributes>
  </tx:advice>
```

And with XML

Read-only

- Or read-only transaction mode requirement

With annotations

```java
@Transactional(readOnly=true)
public Customer getCust(int custId) {
  Customer cust = customerDao.get(custId);
  Hibernate.initialize(cust.getShipAddress());
  Hibernate.initialize(cust.getCreditCard());
  Hibernate.initialize(cust.getCreditCard().getAddress());

  return cust;
}
```

```xml
<tx:advice id="serviceTxAdvice" transaction-manager="txManager">
  <tx:attributes>
    <tx:method name="get*" propagation="REQUIRED" read-only="true"/>
    <tx:method name="add*" propagation="REQUIRED"/>
    <tx:method name="update*" propagation="REQUIRED"/>
  </tx:attributes>
</tx:advice>
```

And with XML

Timeout

- Note that timeout settings have to also be supported by the transaction manager*

Timeout value in seconds

```
@Transactional(timeout=10)
public void updCustomer(Customer cust, Address shipAddr, CreditCard cc,
        Address billAddr) {
  cc.setAddress(billAddr);
  cust.setShipAddress(shipAddr);
  cust.setCreditCard(cc);

  addressDao.update(billAddr);
  addressDao.update(shipAddr);
  ccDao.update(cc);
  customerDao.update(cust);
}
```

Timeout value in seconds

```
<tx:advice id="serviceTxAdvice" transaction-manager="txManager">
  <tx:attributes>
    <tx:method name="get*" propagation="REQUIRED"/>
    <tx:method name="add*" propagation="REQUIRED"/>
    <tx:method name="update*" propagation="REQUIRED" timeout="10"/>
  </tx:attributes>
</tx:advice>
```

Rollback

- By default Spring will rollback for checked exceptions but not for un-checked exceptions*
  - Spring allows you to configure this behavior

```
@Transactional(
  rollbackFor={MyCheckedException.class},
  noRollbackFor={MyRuntimeException.class}
)
public List<Customer> getAll() {
  List<Customer> customers = customerDao.getAll();
  return customers;
}
```

Do rollback for MyCheckedException and don’t rollback for MyRuntimeException

```
  <tx:advice id="serviceTxAdvice" transaction-manager="txManager">
    <tx:attributes>
      <tx:method name="get*" propagation="REQUIRED"
            rollback-for="MyCheckedException"
            no-rollback-for="MyRuntimeException" />
      <tx:method name="add*" propagation="REQUIRED" />
      <tx:method name="update*" propagation="REQUIRED“ />
    </tx:attributes>
</tx:advice>
```

Do rollback for MyCheckedException and don’t rollback for MyRuntimeException

XML Configuration

Specify the Transaction Manager

```
<beans ...>
  ...

  <bean id="txManager" class="org.springframework.orm.hibernate3.HibernateTransactionManager">
    <property name="sessionFactory" ref="sessionFactory" />
  </bean>
```

Use an AOP pointcut to specify the methods that you want to make transactional

```
  <aop:config>
    <aop:pointcut expression="execution(* example.service.*.*(..))" id="serviceTx"/>
    <aop:advisor advice-ref="serviceTxAdvice" pointcut-ref="serviceTx"/>
  </aop:config>

  <tx:advice id="serviceTxAdvice" transaction-manager="txManager">
    <tx:attributes>
      <tx:method name="get*" propagation="REQUIRED"/>
      <tx:method name="add*" propagation="REQUIRED"/>
      <tx:method name="update*" propagation="REQUIRED"/>
    </tx:attributes>
  </tx:advice>
```

Only service methods starting with get, add, or update will really require a TX

Optionally create an additional configuration for DAO methods

```
  <aop:config>
    <aop:pointcut expression="execution(* example.dao.*.*(..))" id="daoTx"/>
    <aop:advisor advice-ref="daoTxAdvice" pointcut-ref="daoTx"/>
  </aop:config>

  <tx:advice id="daoTxAdvice" transaction-manager="txManager">
    <tx:attributes>
      <tx:method name="set*" propagation="SUPPORTS"/>
      <tx:method name="*" propagation="REQUIRED"/>
    </tx:attributes>
  </tx:advice>
</beans>
```

DAO Methods starting with set support a TX, all others require a TX

Active Learning

- Define: transaction demarcation

- Define: Transaction propagation

Summary

- There are a number of issues we should think about when using transactions
  - Global or local transactions
  - Transaction Isolation
  - Transaction Propagation
- With Spring we can specify transactional methods with the @Transactional annotation
- With Spring it becomes easy to make service level methods transactional

Main Point

- Spring Transactions annotations allow you to declaratively specify how transactions should happen, using AOP to accomplish its goals
- *Science of Consciousness*: Do Less and Accomplish More, the transactions are automatically applied in an additional AOP layer