Spring Databases

Module 3
Spring Transactions

Contents

- ACID
- Transaction types
- Overview of API
- Isolation Levels
- Transactions Propagation
- Using AOP in Transaction Management
- Programmatic Transaction Management
- Examples
- Exercises

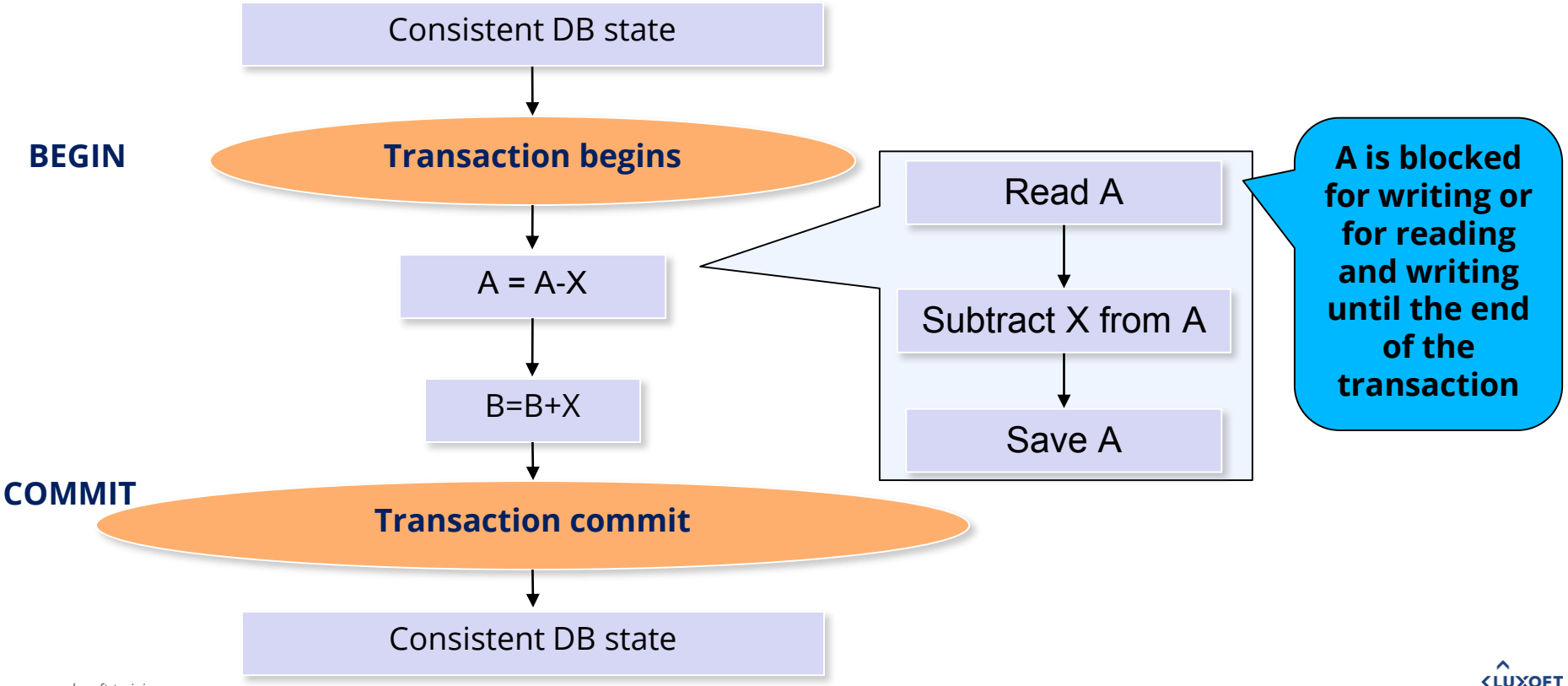
Spring :: Tx :: ACID

Transaction is a group of operations that have ACID properties:

- **Atomicity**: either all actions occur, or nothing occurs
- **Consistency:** once a transaction has completed (successfully or not), the data is in consistent state
- **Isolation:** each transaction should be isolated from others to prevent data corruption
- **Durability:** once a transaction has completed, its result should be durable

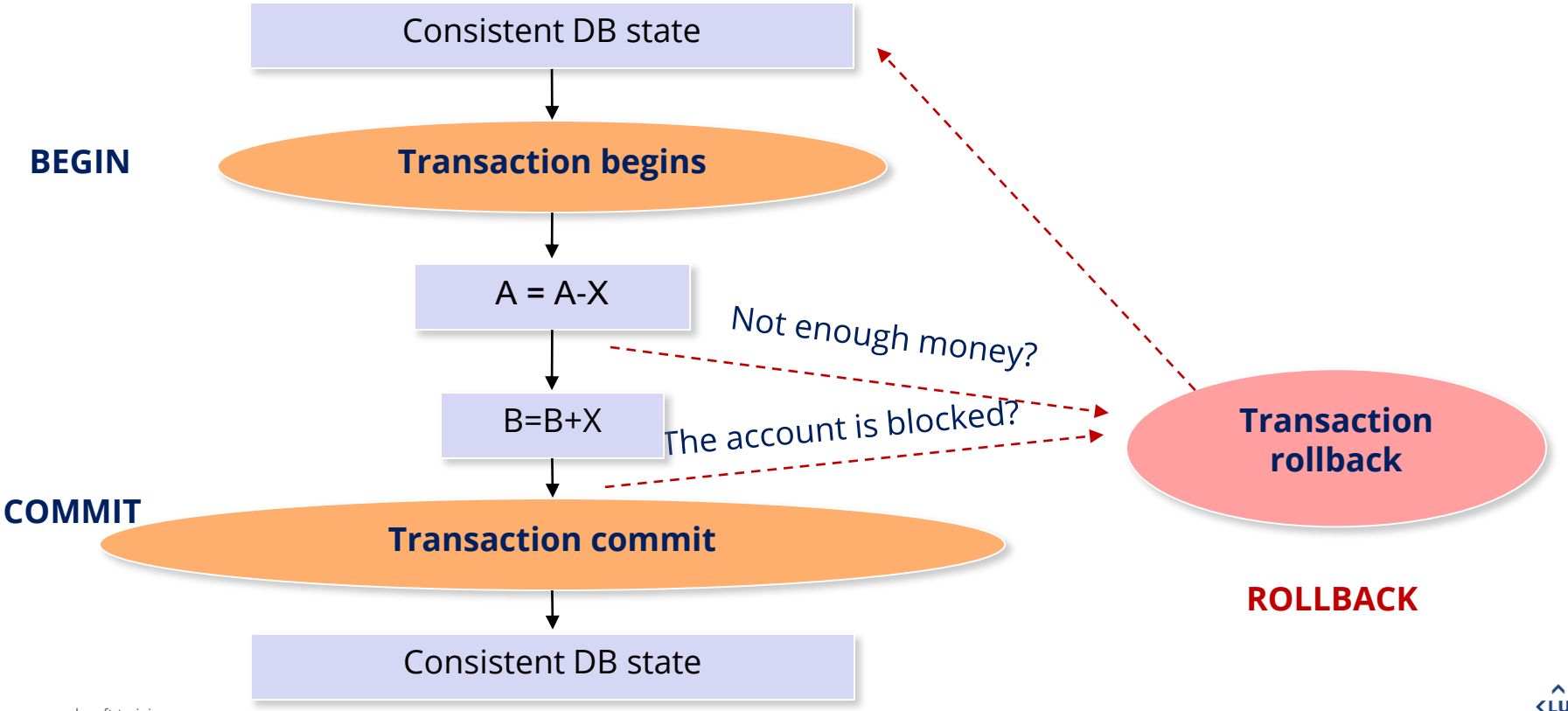
Spring :: Tx :: Transaction

Transfer X dollars from A account to B account



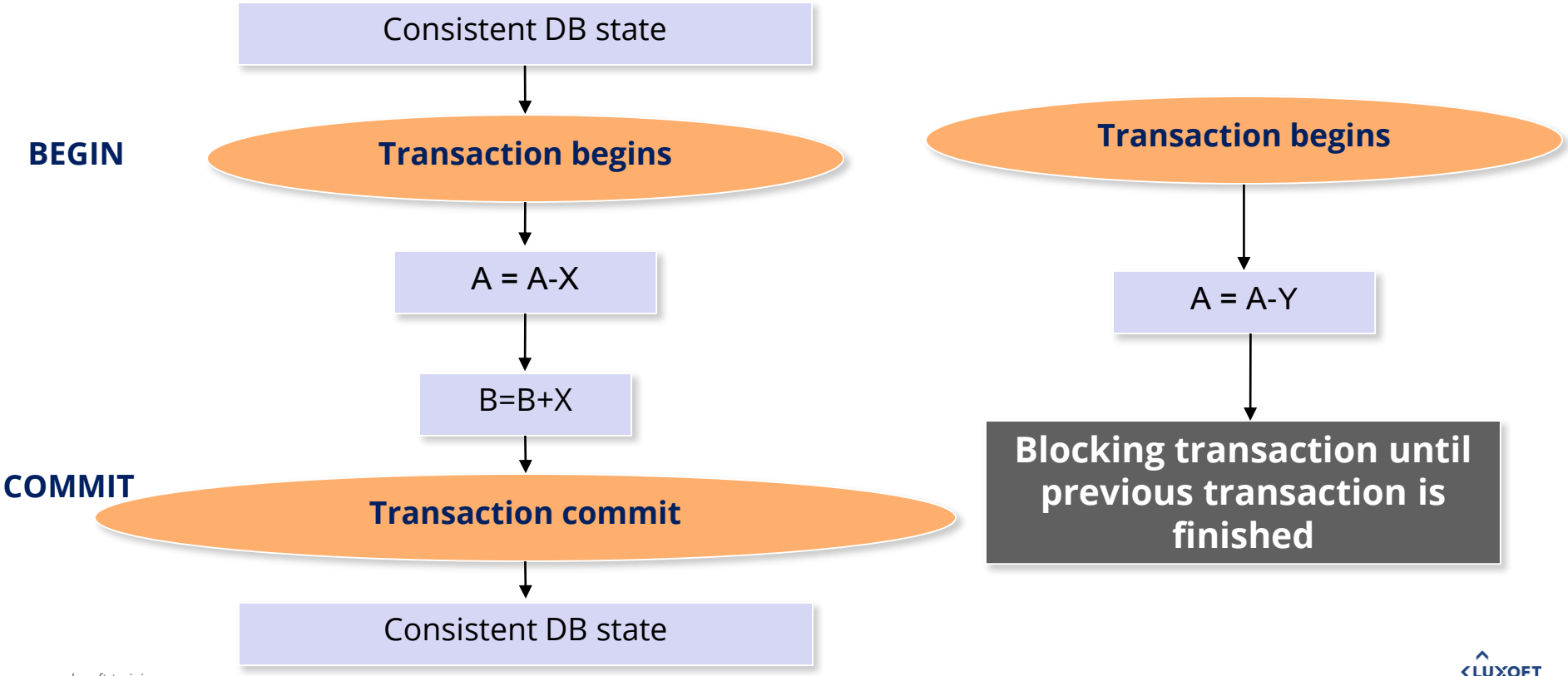
Spring :: ORM :: Example

Transfer X dollars from A account to B account



Spring :: Tx :: ACID

Transfer X dollars from A account to B account

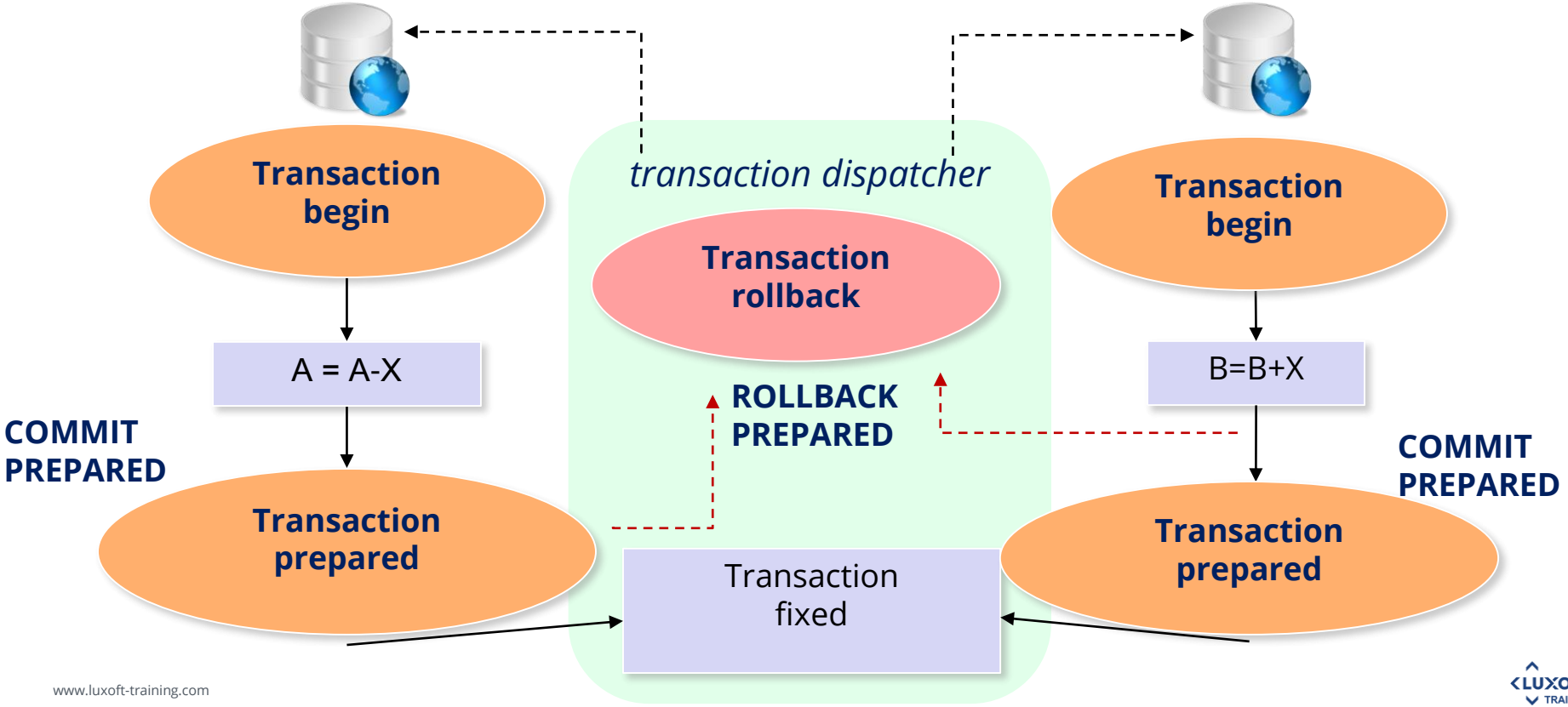


Spring :: Tx :: Transactions types

- **Local**: transaction into the single data source
- **Global**: distributed transactions in more than one data source; managed by the application server, using JTA (Java Transaction API)

Spring :: Tx :: Distributed transactions

Transfer X dollars from A account in one bank to B account in another bank



Spring :: Tx

- Transactions support model used in Spring Framework is applicable to different transaction APIs such as JDBC, Hibernate, JPA, JDO, etc.

- Benefits:
  - Application server is NOT required
  - Declarative transaction management

Spring :: Tx API

- The key Spring transaction abstraction is defined by an interface:

```
org.springframework.transaction.PlatformTransactionManager

public interface PlatformTransactionManager {
	// create transaction by definition
	TransactionStatus getTransaction
		(TransactionDefinition definition) throws TransactionException;
	void commit(TransactionStatus status) throws TransactionException;
	void rollback(TransactionStatus status) throws TransactionException;
}
```

Spring :: Tx API

- TransactionDefinition is used to set parameters of the transaction:

`org.springframework.transaction.TransactionDefinition`

- **TransactionDefinition parameters**:
  - **Isolation** – isolation level of the transaction
  - **Propagation** – transaction propagation – how transactions are propagated from one method to another
  - **Timeout**
  - **Read-only** status

Spring :: Tx API

In case of declarative transactions, **TransactionDefinition** is created implicitly through annotations:

```
@Transactional(readOnly=true)
public class DefaultFooService implements FooService {

   public Foo getFoo(String fooName) {
       // do something
   }

   @Transactional(readOnly=false, propagation = Propagation.REQUIRES_NEW)
  public void updateFoo(Foo foo) {
       // do something
   }
}
```

Spring :: Tx API

- You need to include this directive into the application context:

```
<tx:annotation-driven/>
```

- And add one of the transaction managers to the context:

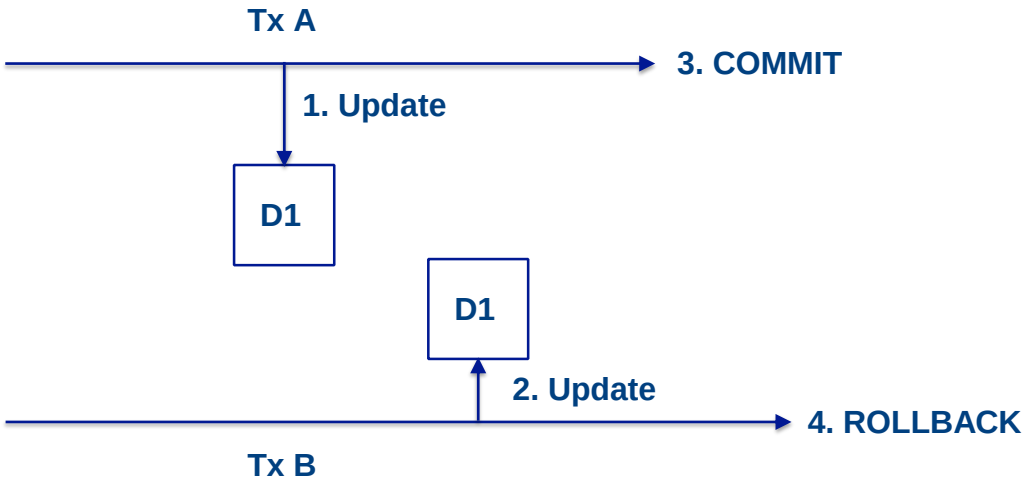
```
<bean id="transactionManager"
    class="org.springframework.orm.hibernate5.HibernateTransactionManager">
```

Spring :: Tx :: Transaction isolation levels

Isolation level	Phantom reading	Unrepe-atable read	«Dirty» read	Lost update
ISOLATION_READ_UNCOMMITTED				
READ UNCOMMITTED	–	–	–	+
ISOLATION_READ_COMMITTED				
READ COMMITTED	–	–	+	+
ISOLATION_REPEATABLE_READ				
REPEATABLE READ	–	+	+	+
ISOLATION_SERIALIZABLE				
SERIALIZABLE	+	+	+	+

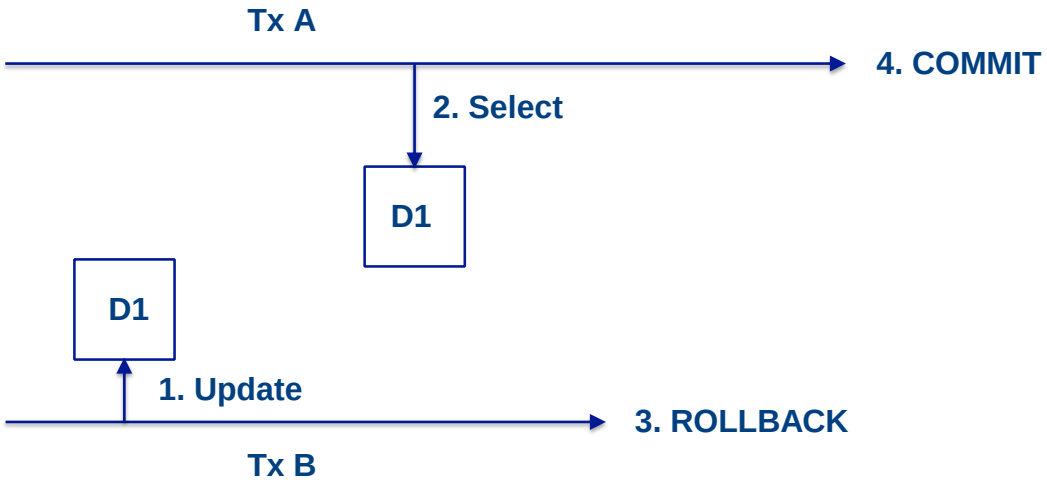
Spring :: Tx :: Lost Update

Two transactions update the same data without isolation.



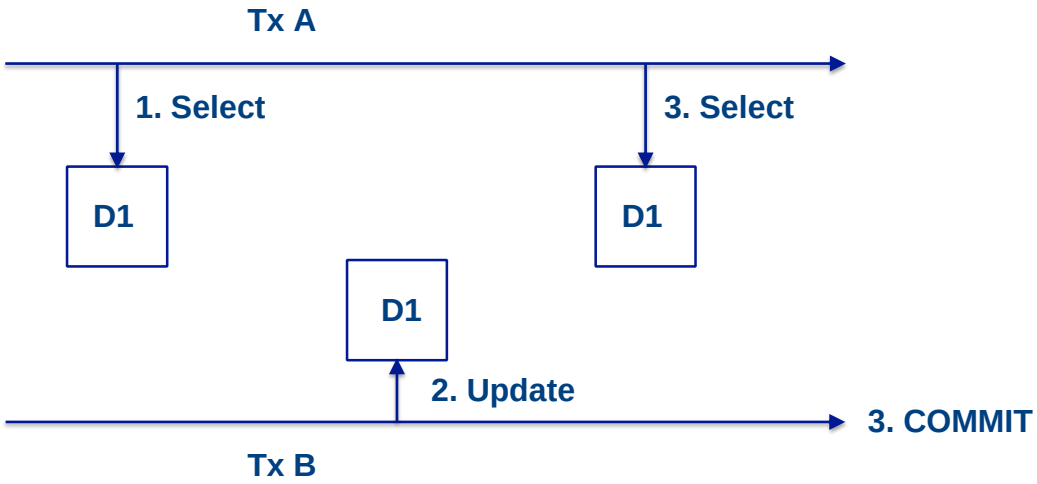
Spring :: Tx :: Dirty Read

Transaction A reads uncommitted data from transaction B.



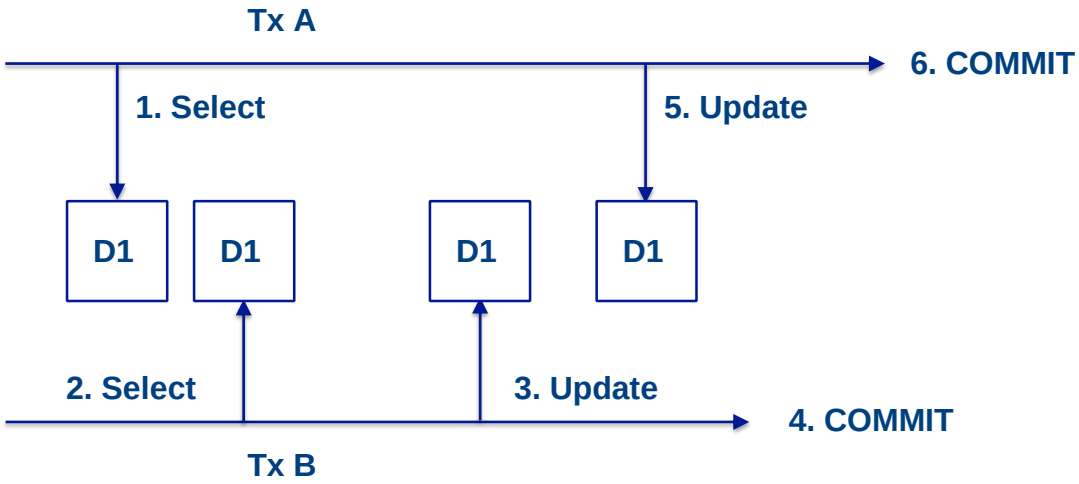
Spring :: Tx :: Unrepeatable read

Transaction A executes two non-repeatable reads.



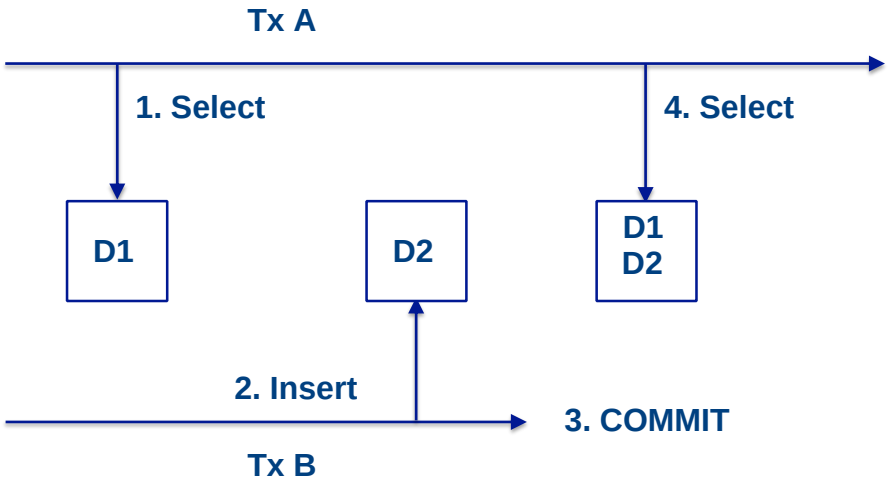
Spring :: Tx :: Unrepeatable read

Last commit wins problem. Tx A overrides changes of Tx B.



Spring :: Tx :: Phantom read

Tx A reads new data in the second SELECT.



Spring :: Tx :: Isolation levels

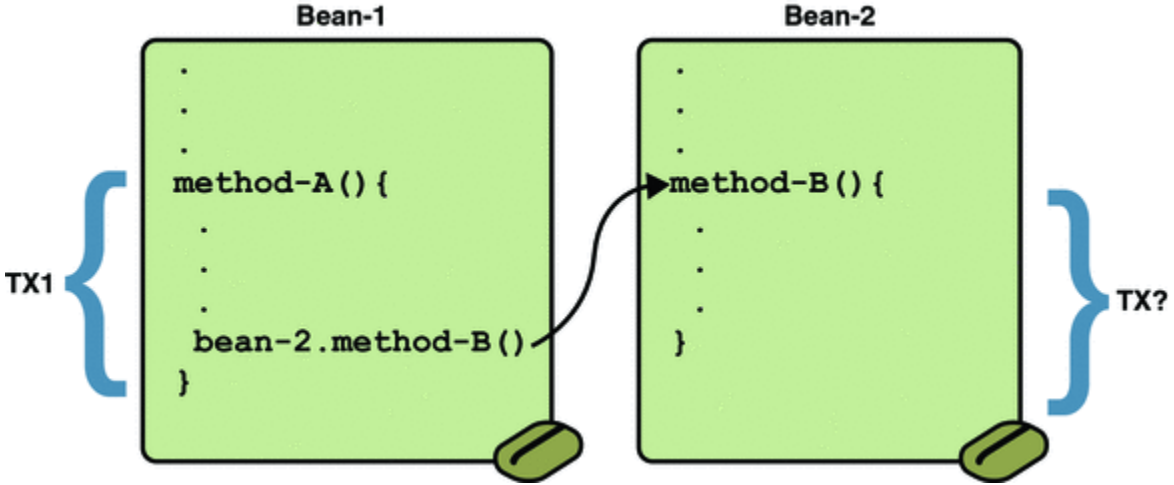
Enum **org.springframework.transaction.annotation.Isolation**

Isolation level	Phantom reading	Unrepe-atable read	«Dirty» read	Lost update
ISOLATION_READ_UNCOMMITTED				
READ UNCOMMITTED	–	–	–	+
ISOLATION_READ_COMMITTED				
READ COMMITTED	–	–	+	+
ISOLATION_REPEATABLE_READ				
REPEATABLE READ	–	+	+	+
ISOLATION_SERIALIZABLE				
SERIALIZABLE	+	+	+	+

Spring :: Tx :: Propagation

Transaction propagation: what happens to transaction when method is called?
Spring supports the follow list of propagation:

Enum **`org.springframework.transaction.annotation.Propagation`**:
Required, RequiresNew, Mandatory, NotSupported, Supports, Never



```
public interface PlatformTransactionManager
{

    TransactionStatus getTransaction(
        TransactionDefinition definition);

    void commit(TransactionStatus status);

    void rollback(TransactionStatus status);

}
```

***all methods throws TransactionException**

LUXOFT TRAINING

public interface PlatformTransactionManager {…}

Most widely-used implementations:

- DataSourceTransactionManager
- HibernateTransactionManager
- JmsTransactionManager
- JmsTransactionManager102
- JpaTransactionManager
- OC4JJtaTransactionManager
- WebLogicJtaTransactionManager
- WebSphereUowTransactionManager

```
public interface TransactionDefinition
{
    int getPropagationBehavior();

    int getIsolationLevel();

    int getTimeout();

    boolean isReadOnly();

    String getName();

}
```

```
public interface TransactionStatus
{
    boolean isNewTransaction();

    boolean hasSavepoint();

    void setRollbackOnly();

    boolean isRollbackOnly();

    void flush();

    boolean isCompleted();
}
```

Spring :: Tx :: Propagation

`org.springframework.transaction.annotation.Propagation`:

Transaction attribute	Client transaction	Transaction in business method
Required – If transaction was started – it uses it. If not – start the new transaction	No	T2
	T1	T1
RequiresNew – If method () is invoked within the context of another transaction, it will be suspended and a new transaction will be started. Once method() ends, new transaction is either committed or rolled back, and first transaction resumes.	No	T2
	T1	T2
Mandatory – Check for the opened transaction. If transaction was not started, throws TransactionRequiredException.	No	Error
	T1	T1

Spring :: Tx :: Propagation

`org.springframework.transaction.annotation.Propagation`:

Transaction attribute	Client transaction	Transaction in business method
NotSupported – Suspend current transaction if it was started.	No	No
	T1	No
Supports –Use current transaction. If there was no transaction, it works without transaction.	No	No
	T1	T1
Never – If there was a transaction, generate exception RemoteException.	No	No
	T1	Error

Spring :: Tx :: Propagation

Transaction propagation use cases:
If the **Required** transaction attribute is the most commonly used transaction attribute and is the default for both EJB 3.0 and Spring.

The **RequiresNew** This transaction attribute should only be used for database operations (such as auditing or logging) that are independent of the underlying transaction. This attribute actually violates ACID criteria (specifically the atomicity property). In other words, all database updates are no longer contained within a single unit of work. If the first transaction was to be rolled back, the changes committed by new transaction remain committed.

If the **Mandatory** transaction, if method() is invoked under an existing transaction scope, the existing transaction scope will be used. However, if method() is invoked without a transaction context, then a **`TransactionRequiredException`** will be thrown, indicating that a transaction must be present before method() is invoked.

Spring :: Tx :: Propagation

Transaction propagation use cases:

NotSupported specifies that the method being called will not use or start a transaction, regardless if one is present. If the **NotSupported** transaction attribute is specified for method() and method() is invoked in context of a transaction, that transaction is suspended until method() ends. When method() ends, the original transaction is then resumed..

Supports transaction attribute. If the Supports transaction attribute is specified for method() and method() is invoked within the scope of an existing transaction, method() will execute under the scope of that transaction. However, if method() is invoked without a transaction context, then no transaction will be started. This attribute is primarily used for read-only operations to the database. If that is the case, why not specify the **NotSupported** transaction attribute instead? After all, that attribute guarantees that the method will run without a transaction. Invoking the query operation in the context of an existing transaction will cause data to be read from the database transaction log (in other words, updated data), whereas running without a transaction scope will cause the query to read unchanged data from the table.

Spring :: Tx :: Propagation

Transaction propagation use cases:

Never transaction attribute. The only use case to come up with this transaction attribute may be for testing. It provides a quick and easy way of verifying that a transaction exists when you invoke a particular method. If you use the **Never** transaction attribute and receive an exception when invoking the method in question, you know a transaction was present. If the method is allowed to execute, you know a transaction was not present.

Spring :: Tx :: Examples

- Let’s study some examples of specific transaction management implementations in application context

Spring :: Tx :: Examples

DataSourceTransactionManager:

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource"
destroy-method="close">
    <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver" />
    <property name="url" value="jdbc:oracle:thin:@rj-t42:1521:elvis" />
    <property name="username" value="scott" />
    <property name="password" value="tiger" />
</bean>

<bean id="transactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource" />
</bean>
```

Spring :: Tx :: Examples

Default Rollback rules:

- They enable us to specify which exceptions should cause automatic roll back;
- By default, transactions are rolled back only with **`RuntimeException`** ;

This behavior can be redefined:

```
@Transactional(rollbackFor = IOException.class,
               noRollbackFor = RuntimeException.class)
public void doSomething() {
…
}
```

Spring :: Tx :: @Transactional

@Transactional annotation properties:

Property	Type	Description
propagation	enum: Propagation	Optional propagation setting.
isolation	enum: Isolation	Optional isolation level.
readOnly	Boolean	Read/write vs. read-only transaction. Read-only transaction may work faster and block less resources.
timeout	int (seconds)	Transaction timeout after which transaction will automatically rollout.

Spring :: Tx :: @Transactional

@Transactional annotation properties:

Property	Type	Description
rollbackFor	Array of Class objects, which must be derived from Throwable.	Optional array of exception classes that must cause rollback.
rollbackForClassname	Array of class names. Classes must be derived from Throwable.	Optional array of names of exception classes that must cause rollback.
noRollbackFor	Array of Class objects, which must be derived from Throwable.	Optional array of exception classes that must not cause rollback.
noRollbackForClassname	Array of String class names, which must be derived from Throwable.	Optional array of names of exception classes that must not cause rollback.

Spring :: Tx :: Rollback

@Transactional applies to:

- Interfaces
- Classes
- Interface methods
- Class public methods

It is better to apply **@Transactional** to specific classes and their methods, not to interfaces

Spring :: Tx :: Programmatic management

- Generally, declarative transaction management is used - especially if there are many transactions in application
- Programmatic management is used in case:
  - There are few transactions in the application. **`TransactionTemplate`** can be used, but it is not advisable,
  - Transaction name has to be specified explicitly.

Spring :: Tx :: Programmatic management

One possible option is using **`TransactionTemplate`**:

```
public class SimpleService {
  private TransactionTemplate transactionTemplate;

  public Object someServiceMethod() {
     return transactionTemplate.execute(new TransactionCallback() {
        public Object doInTransaction(TransactionStatus status) {
           updateOperation1();
           return resultOfUpdateOperation2();
        }
     });
  }
}
```

Spring :: Tx :: Programmatic management

In this case all properties can be defined programmatically:

```
public void nonCallbackService() {
  transactionTemplate.setIsolationLevel(
                      TransactionDefinition.ISOLATION_READ_COMMITTED);
  transactionTemplate.setReadOnly(false);
  transactionTemplate.setTimeout(100);
  transactionTemplate.setPropagationBehavior(
                      TransactionDefinition.PROPAGATION_REQUIRED);
}
```

Spring :: Tx :: Programmatic management

- In this case all properties can be defined programmatically :
  - **TransactionTemplate** supports callback approach;
  - Implement **TransactionCallback** using **doInTransaction()** method;
  - Pass it to **execute()** method exposed on the **TransactionTemplate**;

```
public void callbackService() {
  transactionTemplate.execute(new TransactionCallback() {
     public Object doInTransaction(TransactionStatus status) {
        updateOperation1();
        return resultOfUpdateOperation2();
     }
  });
}
```

Exercise

Lab guide:

- Exercise 3