Spring Core – TX

by Andrés Arcia

Spring TX

- PlatformTransactionManager
- @Transactional(…)
- @EnableTransactionManagement / <tx:annotation-driven/>

What is a transaction?

- A set of tasks which take place as a single, atomic, consistent, isolated and durable operation.

ACID Principles

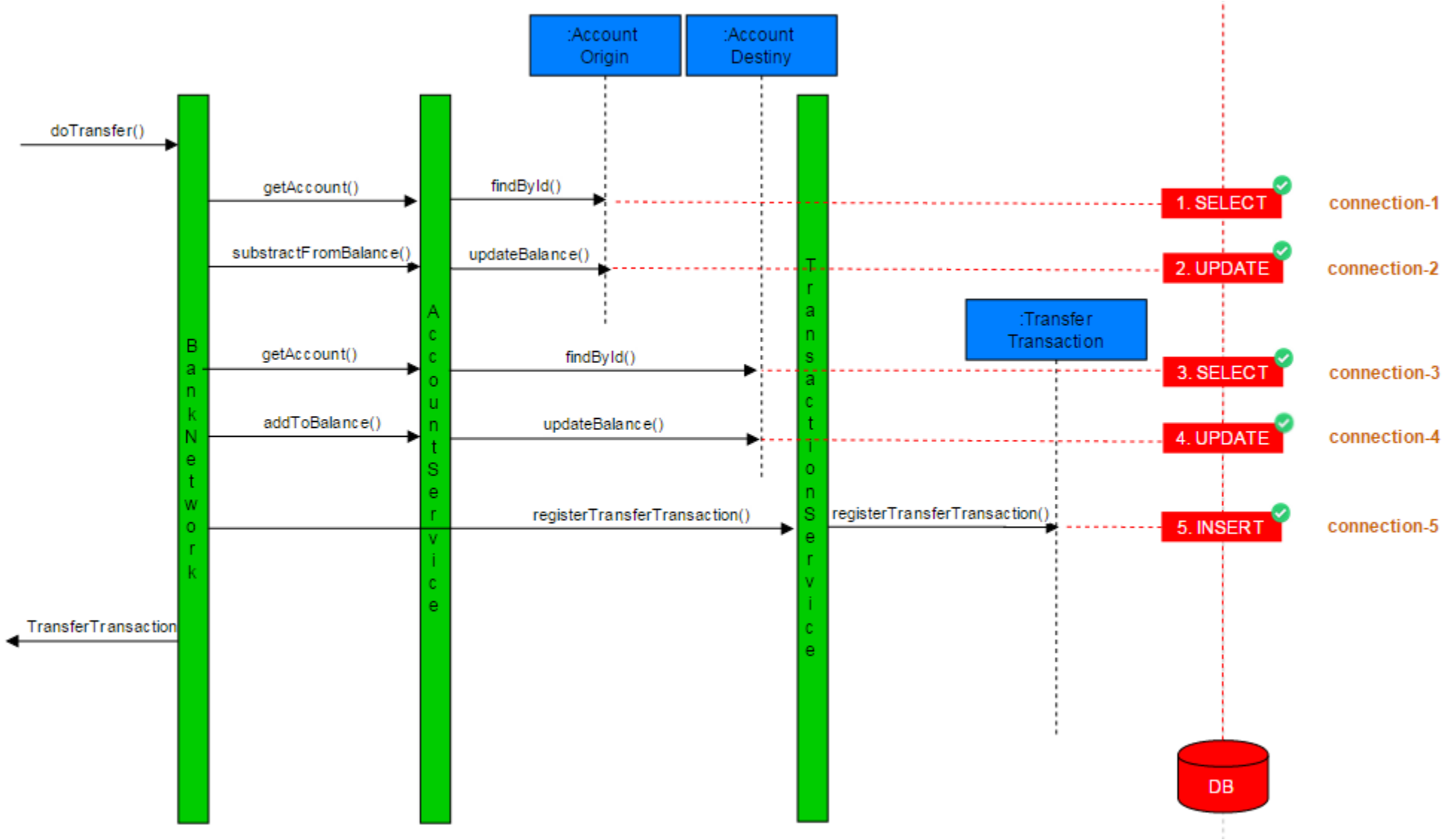
Atomic: Each unit of work is an all-or-nothing operation

Consistent: Database integrity constraints are never violated

Isolated: Isolating transactions from each other

Durable: Committed changes are permanent

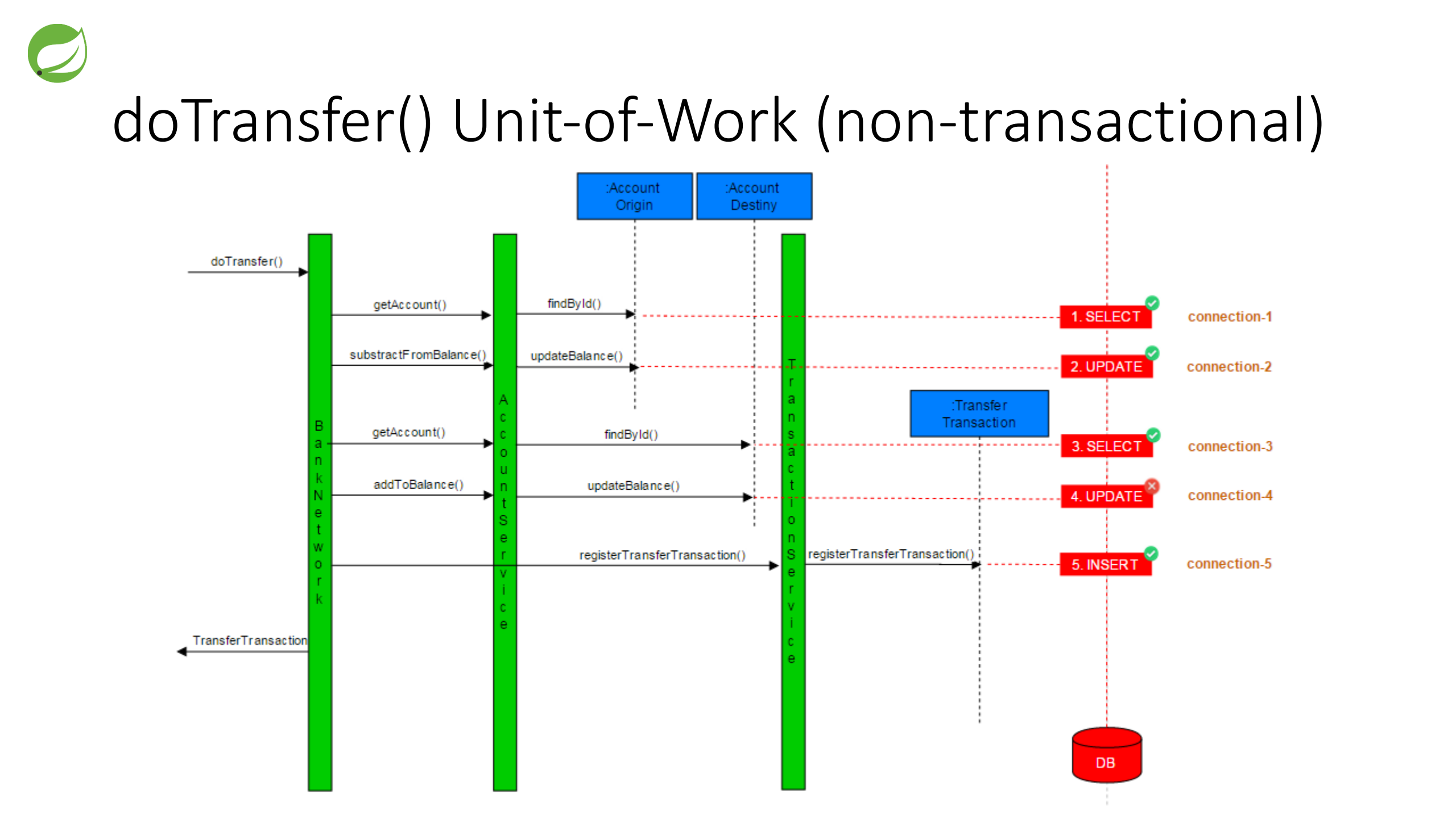
doTransfer() Unit-of-Work (non-transactional)



doTransfer() Unit-of-Work (non-transactional)

- This Unit-of-Work contains 5 data access operations (*each one acquires, uses and releases a distinct connection*)

- The Unit-of-Work is **non-transactional**

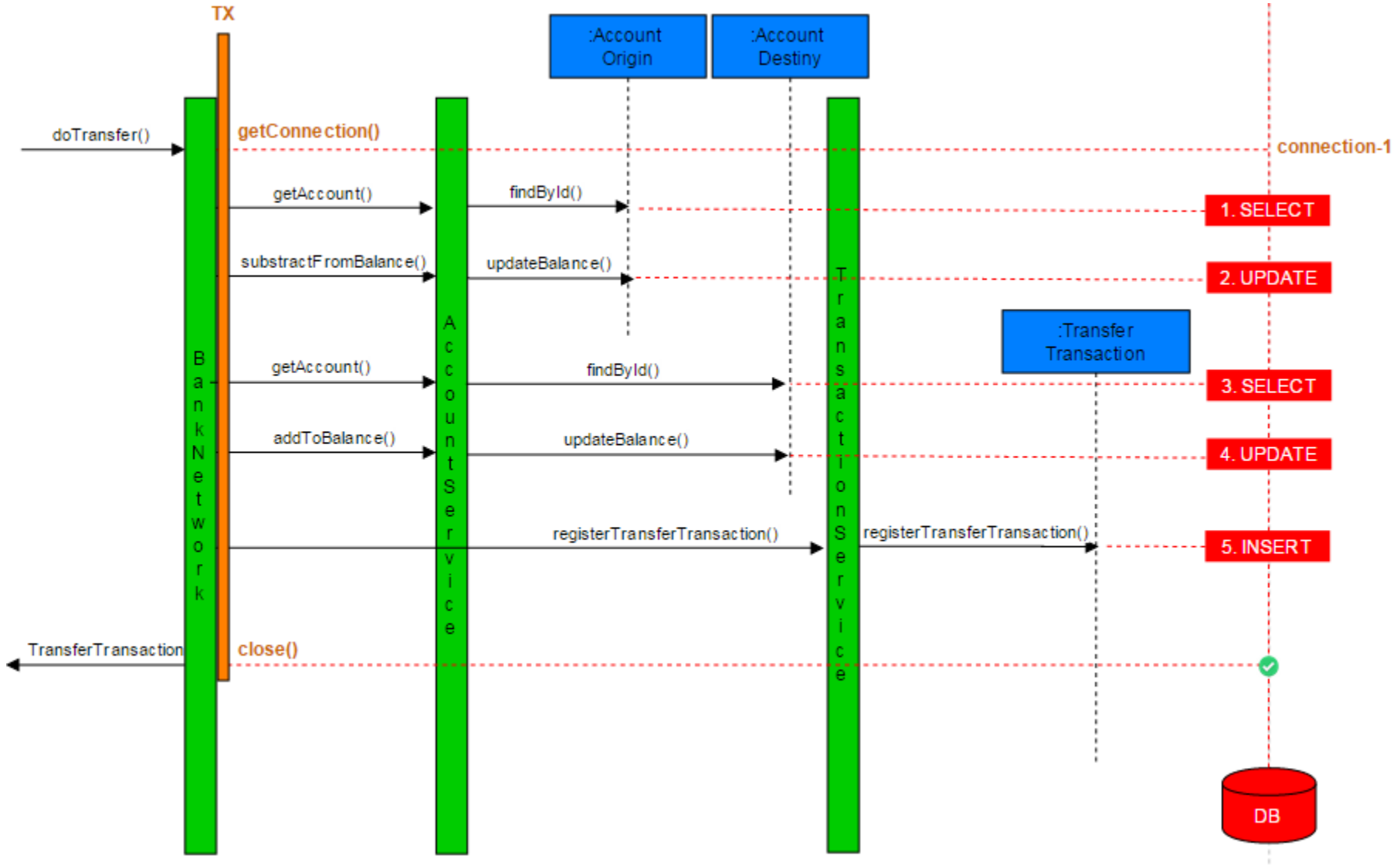


doTransfer() Unit-of-Work (non-transactional)

doTransfer() Unit-of-Work (Transactional)

- Operations must act as an atomic unit (either all succeed or all fail)
- The Unit-of-Work can run in as **transactional**

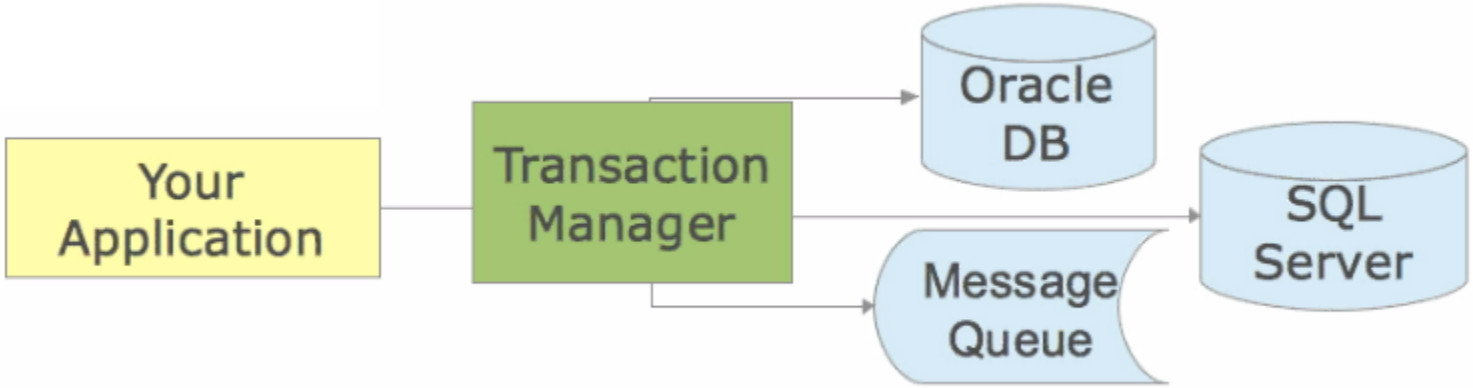
doTransfer() Unit-of-Work (Transactional)



Local and Global Transaction Management

- Local Transactions – Single Resource

- Global (distributed) Transactions – Multiple Resources



JDBC Transaction Management example (No Spring)

```
try {
    conn = dataSource.getConnection();
    conn.setAutoCommit(false);


    ...


    conn.commit();
} catch (Exception e) {
    conn.rollback();
    ...
}
```

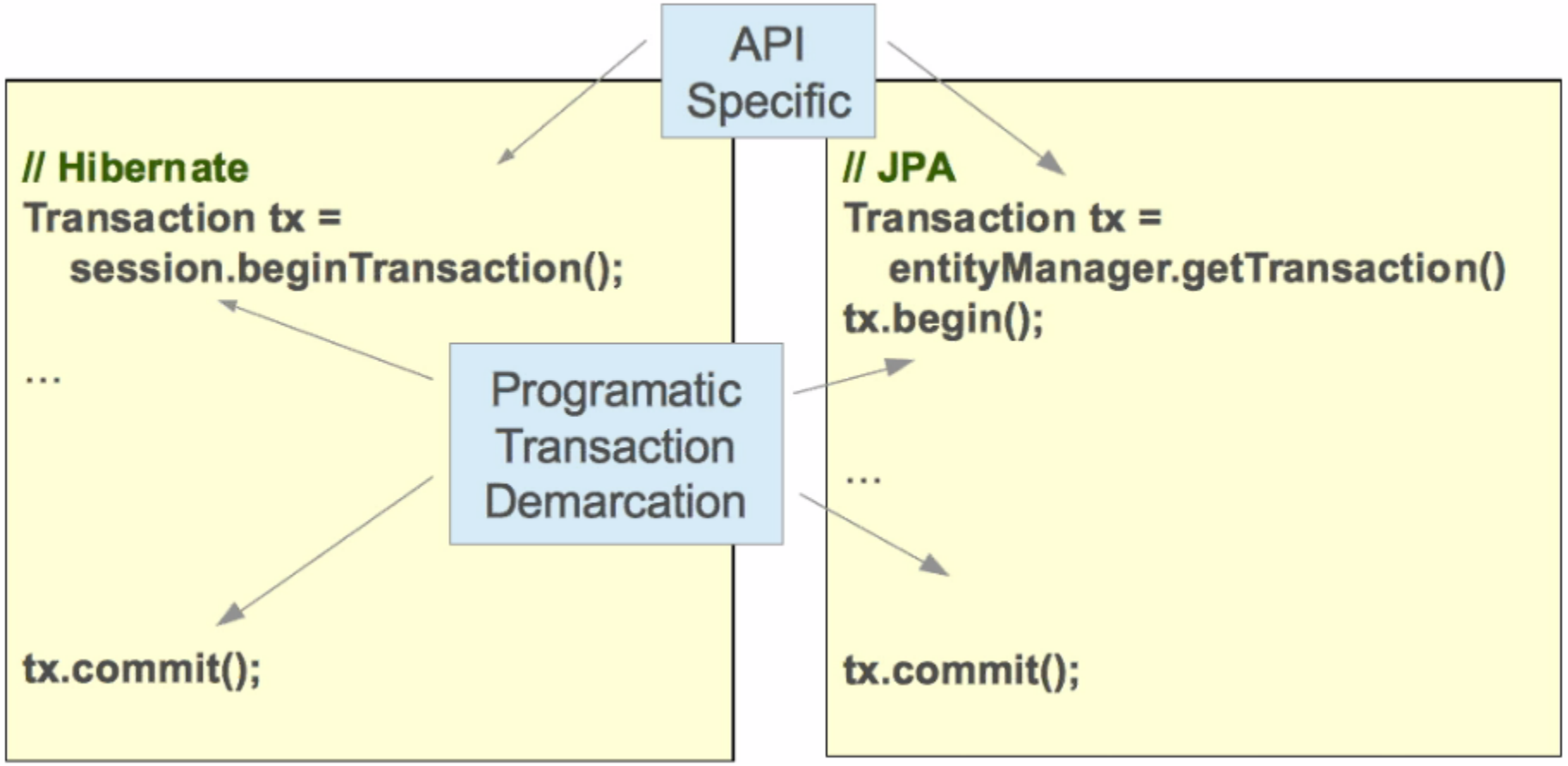
Specific To JDBC API

Programatic Transaction Demarcation

Checked Exceptions

Code cannot 'join' a transaction already in progress
Code cannot be used with global transaction

JPA/Hibernate Transaction Management example (No Spring)



Java Transaction API (JTA) example (No Spring)

```
try {
      UserTransaction ut =
         (UserTransaction) new InitialContext()
               .lookup("java:comp/UserTransaction");
      ut.begin();
      …

      ut.commit();
   } catch (Exception e) {
      ut.rollback();
      ...
   }
```

Programatic Transaction Demarcation

Checked Exceptions

Problems with Java Transaction Management

- Multiple API's for different local resources
- Programmatic transaction demarcation
  - Usually located in the repository layer
  - Usually repeated (cross-cutting concern)
  - Service layer more appropriate (Multiple data access methods may be called within a transaction)
- Orthogonal concerns
  - Transaction demarcation should be independent of transaction implementation.

Spring Transaction Management

- Spring separates transaction demarcation from transaction implementation
  - Demarcation expressed declaratively via AOP (programmatic approach also available)
  - “**PlatformTransactionManage**r” abstraction hides implementation details (several implementations available)

- Spring uses the same API for Global and Local
  - Change from Local to Global is minor

How to use Spring Transaction Management?

- Only 2 steps:
  - Declare a “PlatformTransactionManager” bean
  - Declare the transactional methods (@nnotations, XML, programatic)

- Spring “PlatformTransactionManager” is the base interface for the abstraction: (implementation examples)
  - DataSourceTransactionManager
  - HibernateTransactionManager
  - JpaTransactionManager
  - JtaTransactionManager
  - more…

How to use Spring Transaction Management?

Declare a “PlatformTransactionManager” bean:

- @nnotations

```
@Bean
public PlatformTransactionManager  transactionManager() {
  return new DataSourceTransactionManager(dataSource);
}
```

A dataSource must be defined elsewhere

- XML

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

Bean id “transactionManager” is default name. Can change it but must specify alternative name everywhere – easier not to!

@Transactional(…) – Method Level

```
public class RewardNetworkImpl implements RewardNetwork {
   @Transactional
   public RewardConfirmation rewardAccountFor(Dining d) {
      // atomic unit-of-work
   }
}
```

@Transactional(...) – Class Level

```
@Transactional
public class RewardNetworkImpl implements RewardNetwork {

   public RewardConfirmation rewardAccountFor(Dining d) {
      // atomic unit-of-work
   }

   public RewardConfirmation updateConfirmation(RewardConfirmantion rc) {
      // atomic unit-of-work
   }
}
```

@Transactional(...) – Class and Method Levels

default settings

```
@Transactional(timeout=60)
public class RewardNetworkImpl implements RewardNetwork {

   public RewardConfirmation rewardAccountFor(Dining d) {
      // atomic unit-of-work
   }
   @Transactional(timeout=45)
   public RewardConfirmation updateConfirmation(RewardConfirmantion rc) {
      // atomic unit-of-work
   }
}
```

overriding attributes at the method level

@EnableTransactionManagement - JavaConfig

Defines a Bean Post-Processor
– proxies @Transactional beans

```
@Configuration
@EnableTransactionManagement
public class TxnConfig {
  @Bean
  public PlatformTransactionManager transactionManager(DataSource ds);
     return new DataSourceTransactionManager(ds) {
  }
```

<tx:annotation-driven/> - XML

Defines a Bean Post-Processor
– proxies @Transactional beans

```
<tx:annotation-driven/>

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

<jdbc:embedded-database id="dataSource"> … </jdbc:embedded-database>
```

@Transactional: what happens exactly?

- Target object wrapped in a proxy
  - Uses an “Around” advice
- Proxy implements the following behavior:
  - Transaction started before entering the method
  - Commit at the end of the method
  - Rollback if method throws a RuntimeException (this behavior can be overridden)
- Transaction context bound to current thread
- All controlled by configuration

XML based Spring Transactions

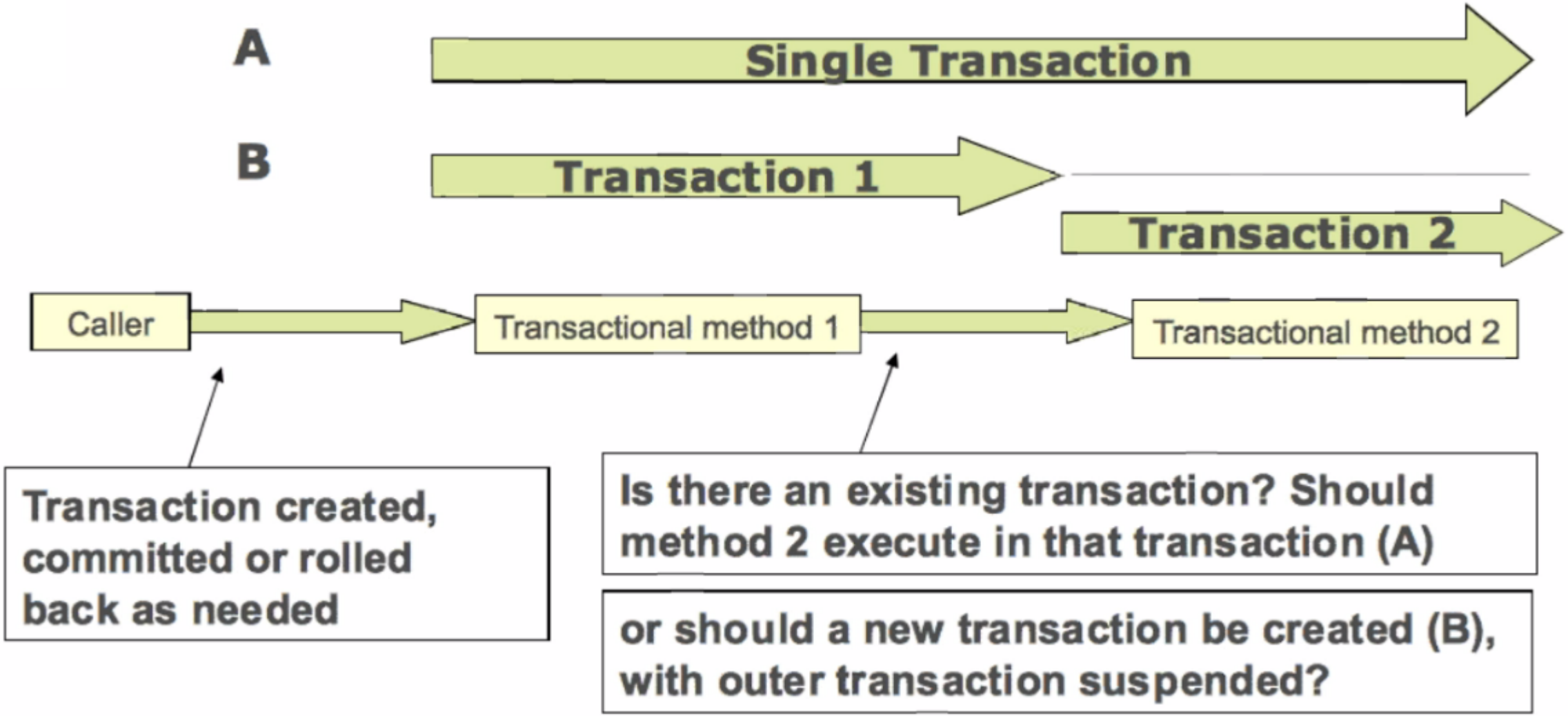
AspectJ *named* pointcut expression

```
<aop:config>
  <aop:pointcut id="rewardNetworkMethods"
          expression="execution(* rewards.RewardNetwork.*(..))"/>
  <aop:advisor pointcut-ref="rewardNetworkMethods" advice-ref="txAdvice"/>
</aop:config>

<tx:advice id="txAdvice">
  <tx:attributes>
    <tx:method name="get*" read-only="true" timeout="10"/>
    <tx:method name="find*" read-only="true" timeout="10"/>
    <tx:method name="*" timeout="30"/>
  </tx:attributes>
</tx:advice>

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

Method-level configuration for transactional advice

Includes rewardAccountFor(..) and updateConfirmation(..)

Understanding Transaction Propagation



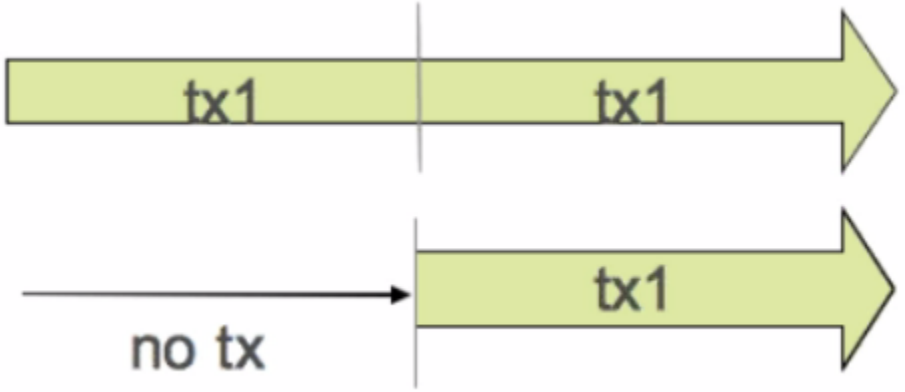
Transaction Propagation with Spring

Propagation Type	If NO current transaction	If there is a current transaction
MANDATORY	throw exception	use current transaction
NEVER	don't create a transaction, run method outside any transaction	throw exception
NOT_SUPPORTED	don't create a transaction, run method outside any transaction	suspend current transaction, run method outside any transaction
SUPPORTS	don't create a transaction, run method outside any transaction	use current transaction
REQUIRED(default)	create a new transaction	use current transaction
REQUIRES_NEW	create a new transaction	suspend current transaction, create a new independent transaction
NESTED	create a new transaction	create a new nested transaction

Transaction Propagation with Spring

- REQUIRED
  - Default value
  - Execute within a current transaction, creates a new one if none exists

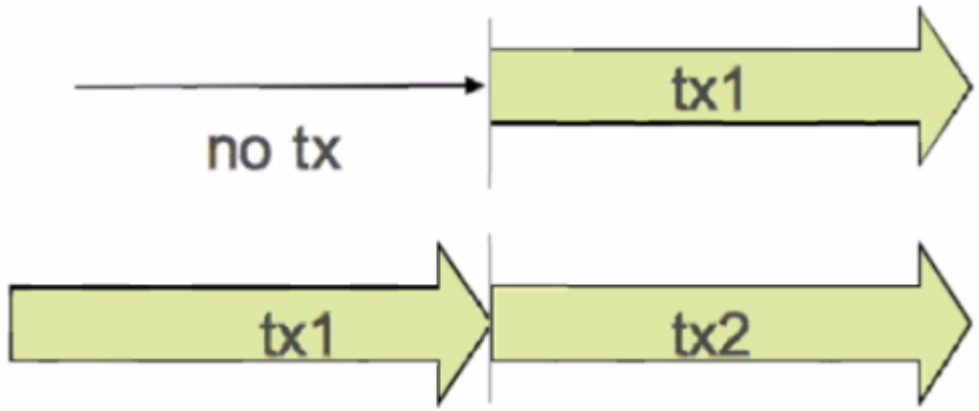
```
@Transactional(propagation=Propagation.REQUIRED)
```



Transaction Propagation with Spring

- REQUIRES_NEW
  - Create a new transaction, suspending the current transaction if one exists

```
@Transactional( propagation=Propagation.REQUIRES_NEW )
```



Rollback Rules

- By default, a transaction is rolled back if a RuntimeException (or subclass) has been thrown.
- Rollback is not executed with any checked exception defined. (see next slide)

```
public class RewardNetworkImpl implements RewardNetwork {
  @Transactional
  public RewardConfirmation rewardAccountFor(Dining d) {
    // ...
    throw new RuntimeException();

  }
}
```

Triggers a rollback

rollbackFor / noRollbackFor

- Default settings can be overriden

```
public class RewardNetworkImpl implements RewardNetwork {


  @Transactional(rollbackFor=MyCheckedException.class)
  public void updateConfirmation(Confirmation c) throws MyCheckedException {
    // ...
  }

  @Transactional(noRollbackFor={JmxException.class, MailException.class})
  public RewardConfirmation rewardAccountFor(Dining d) {
    // ...
  }
```