Transaction Management with Spring

Transactional Proxies and @Transactional

1.18.5

Objectives

After completing this lesson, you should be able to do the following

- Explain why Transactions are used
- Describe and use Spring Transaction Management
- Configure Transaction Propagation
- Setup Rollback rules
- Use Transactions in Tests

Agenda

- **Why use Transactions?**
- Java Transaction Management
- Spring Transaction Management
- Transaction Propagation
- Rollback rules
- Testing

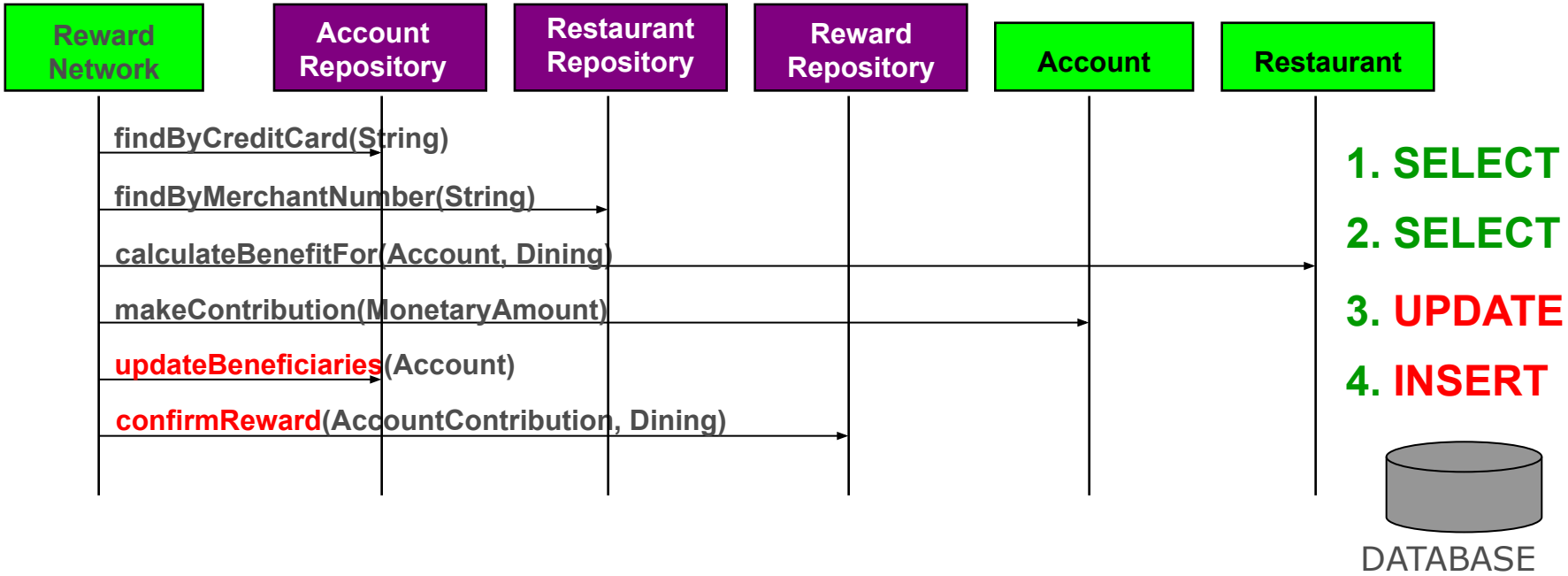
What is a Transaction?

Enable ***concurrent*** access to a ***shared*** resource

- A set of tasks which take place as a single, indivisible action
  - **A**tomic
    - Each unit of work is an all-or-nothing operation
  - **C**onsistent
    - Database integrity constraints are never violated
  - **I**solated
    - Isolating transactions from each other
  - **D**urable
    - Committed changes are permanent

Transactions in the RewardNetwork

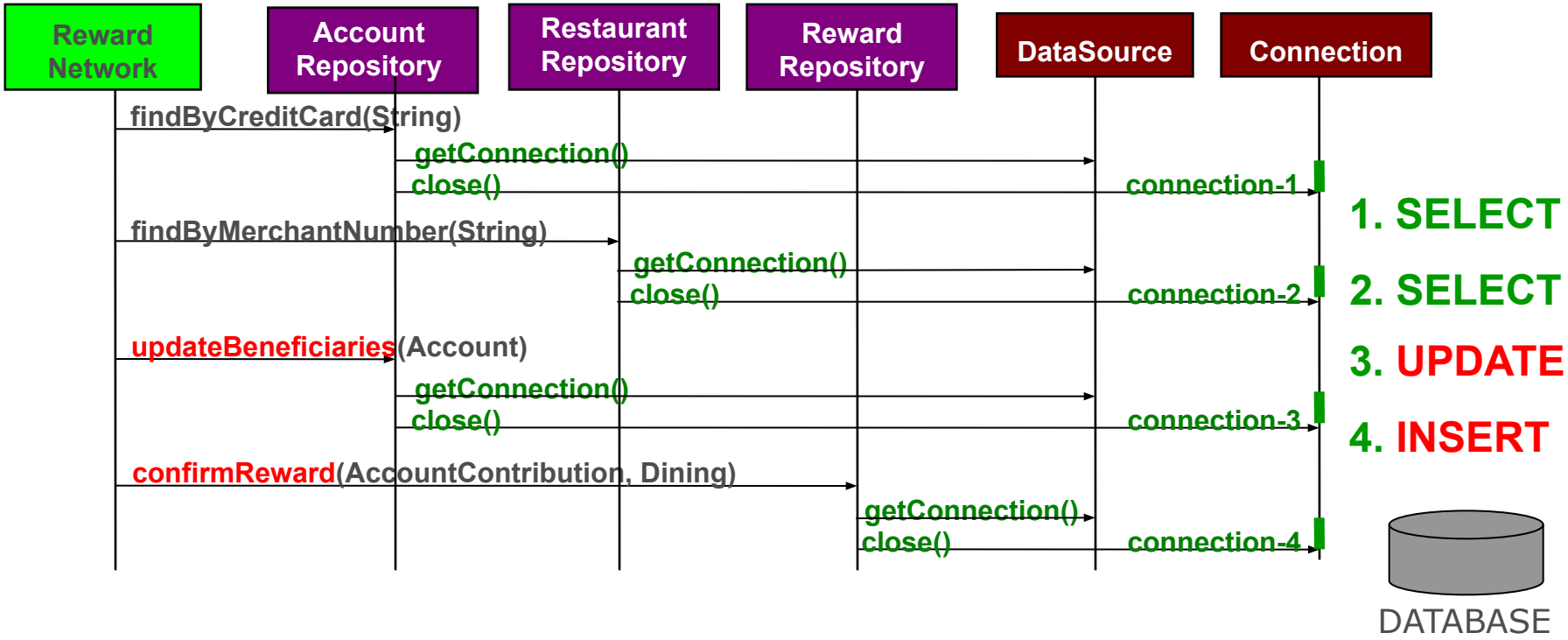
- The *rewardAccountFor(Dining)* method represents a unit-of-work that should be atomic



Naive Approach

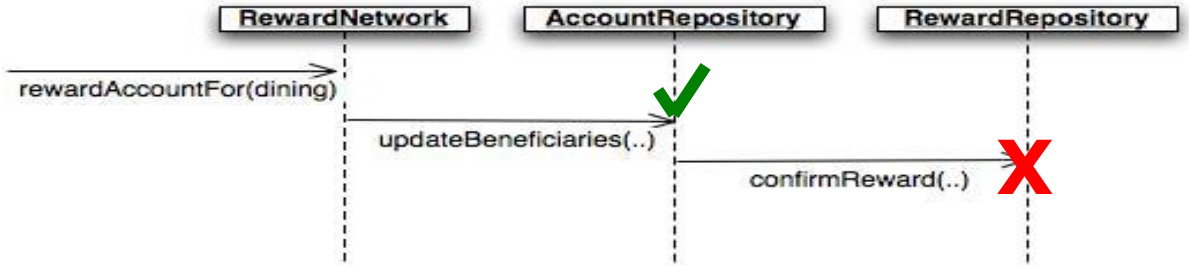
- Connection per data access operation
  - This unit-of-work contains 4 data access operations
    - Each acquires, uses, and releases a distinct Connection
  - The unit-of-work is *non-transactional*

Running non-Transactionally - Multiple Connections Used



Partial Failures (in non-Transactional operation)

- Suppose an Account is being rewarded



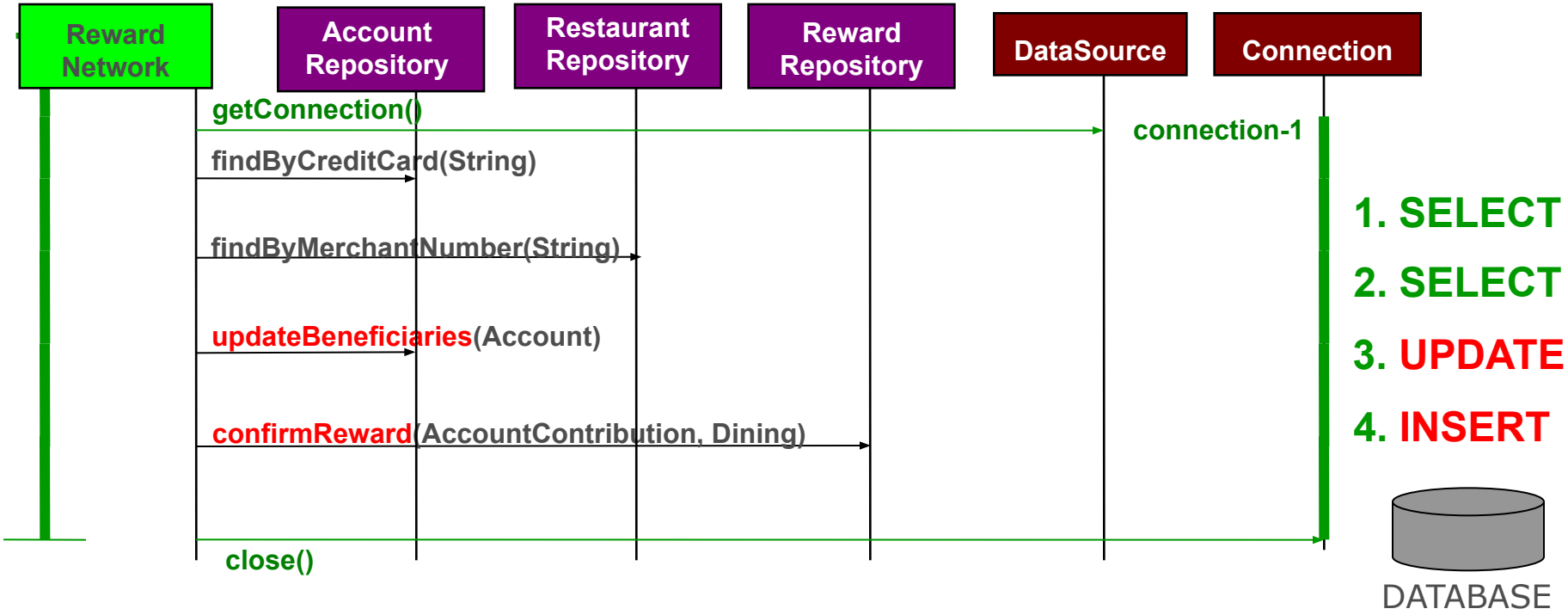
- If the beneficiaries are updated…
- But the reward confirmation fails…
- There will be no record of the reward!

The unit-of-work is **not** *atomic*

Correct Approach

- Connection per Unit-of-Work
  - More efficient
    - Same Connection reused for each operation
  - Operations complete as an atomic unit
    - Either all succeed or all fail
  - The unit-of-work can run in a *transaction*

Running in a Transaction - A single connection used



Agenda

- Why use Transactions?
- **Java Transaction Management**
- Spring Transaction Management
- Transaction Propagation
- Rollback rules
- Testing

Transactional Code Pattern

- Many different APIs, but a common pattern
  - Implemented using code
  - Classic cross-cutting concern

```
try {
    beginTransaction

    …

    commitTransaction
} catch (Exception e) {

    rollbackTransaction
}
```

Programmatic Transaction Demarcation

Typically *checked* Exceptions

Java Transaction API - Different API for different resource (Hard to use)

API	Begin Transaction	End Transaction
JDBC	conn = dataSource.getConnection() conn.setAutoCommit(false)	conn.commit() conn.rollback()
JMS	session = connection .createSession (true, 0)	session.commit() session.rollback()

JPA	Transaction tx =
	entityManager.getTransaction();
	tx.begin();
		tx.commit()
		tx.rollback()

Hibernate	Transaction tx = session.beginTransaction();	tx.commit() tx.rollback()

Agenda

- Why use Transactions?
- Java Transaction Management
- **Spring Transaction Management**
- Transaction Propagation
- Rollback rules
- Testing

Spring Transaction Management

- There are only 2 steps
  - Declare a **`PlatformTransactionManager`** bean
  - Declare the transactional methods
    - Using Annotations (recommended) or Programmatic
    - Can mix and match
  - Add **`@EnableTransactionManagement`** to a configuration class

PlatformTransactionManager Implementations

- Spring's **PlatformTransactionManager** is the base interface for the abstraction
- Several implementations are available
  - `DataSourceTransactionManager`
  - `JmsTransactionManager`
  - `JpaTransactionManager`
  - `JtaTransactionManager`
  - `WebLogicJtaTransactionManager`
  - `WebSphereUowTransactionManager`
  - *and more*

Deploying the Transaction Manager

- Create the required implementation
  - Just like any other Spring bean
    - Configure it as appropriate
  - Here is the manager for a DataSource

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

A DataSource bean must be defined elsewhere

@Transactional Configuration

In your code

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

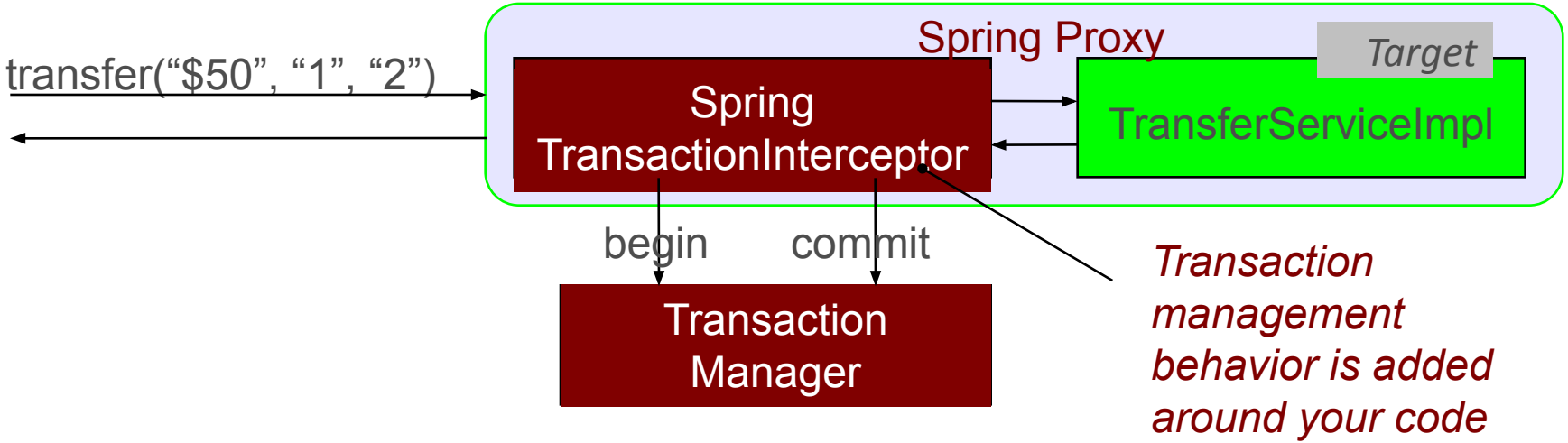
```
@Configuration
@EnableTransactionManagement
public class TxnConfig {
  @Bean
  public PlatformTransactionManager transactionManager(DataSource ds) {
    return new DataSourceTransactionManager(ds);
  }
```

Enables Spring's annotation-driven transaction management capability

In your Spring configuration

Declarative Transaction Management

- Target service wrapped in a proxy
- Caller injected with proxy reference



@Transactional: What Happens Exactly?

- Proxy implements the following behavior
  - Transaction started before entering the method
  - Commit at the end of the method
  - Rollback if method throws a **`RuntimeException`**
    - Default behavior
    - Can be overridden (see later)
- All controlled by *configuration*

@Transactional – Class Level

- Applies to all methods declared by the interface(s)

```
@Transactional
public class RewardNetworkImpl implements RewardNetwork {

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

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

Alternatively @*Transactional* can be declared on the interface instead – since Spring Framework 5.0

@Transactional – Class *and* method levels

- Combining class and method levels

default settings

override attributes at method level

```
@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
  }
}
```

Java's @Transactional

- Java also has an annotation
  - `javax.transaction.Transactional`
- Also supported by Spring
  - Fewer options
  - Not used in these examples

Agenda

- Why use Transactions?
- Java Transaction Management
- Spring Transaction Management
- **Transaction Propagation**
- Rollback rules
- Testing

Understanding Transaction Propagation

- What should happen if **`ClientServiceImpl`** calls **`AccountServiceImpl`**?
  - Single transaction?
  - Two separate transactions?

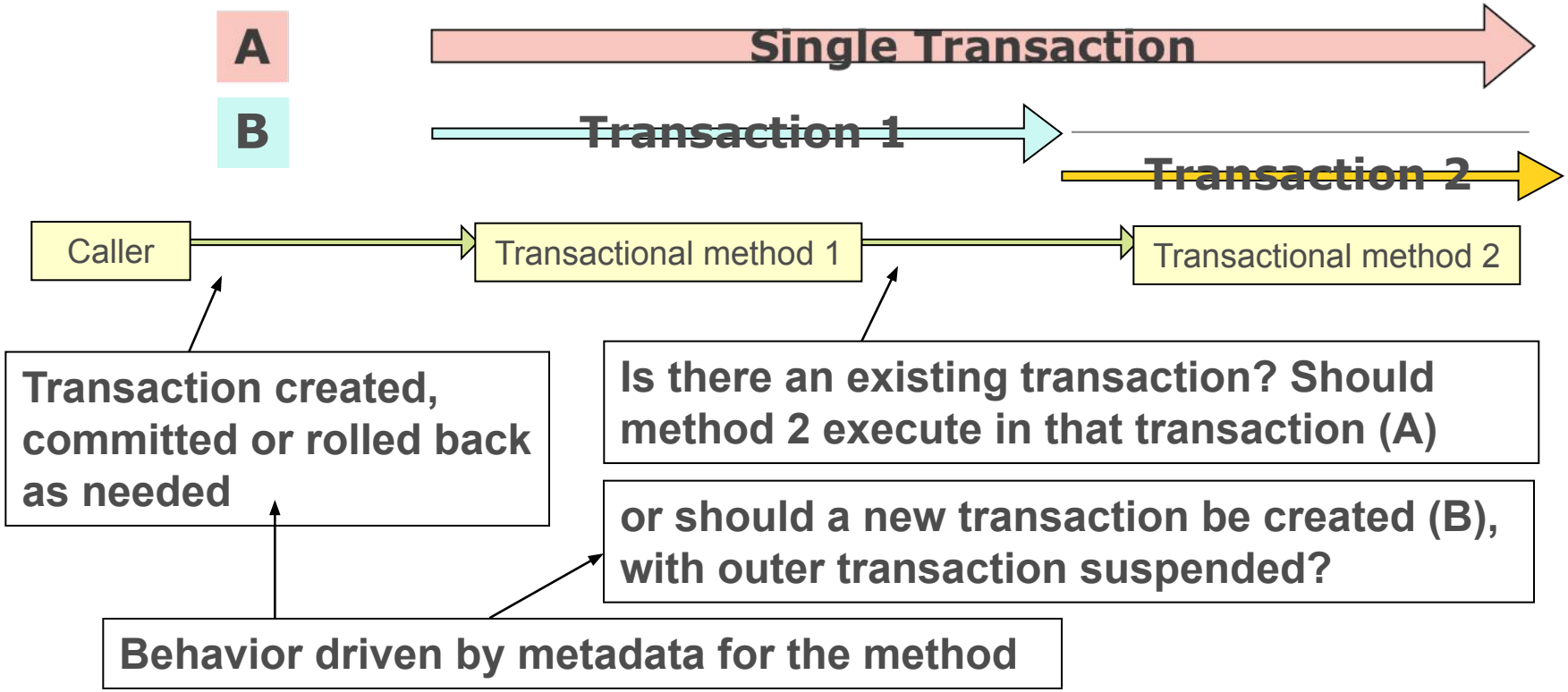
```
public class ClientServiceImpl
      implements ClientService {
  @Autowired
  private AccountService accountService;

  @Transactional
  public void updateClient(Client c) {
    // …
    this.accountService.update(c.getAccounts());
  }
}
```

```
public class AccountServiceImpl
      implements AccountService {

  @Transactional
  public void update(List <Account> accs)
   { // … }
}
```

Understanding Transaction Propagation



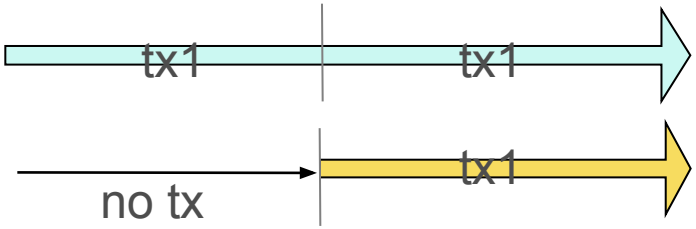
Transaction Propagation with Spring

- 7 levels of propagation
- The following examples show *REQUIRED* and *REQUIRES_NEW*
- Can be used as follows:

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

REQUIRED

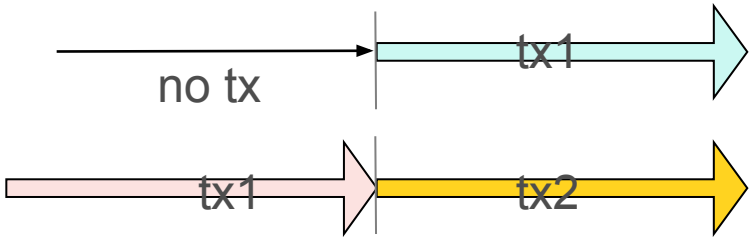
- Default value
- Execute within a current transaction, create a new one if none exists




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

REQUIRES_NEW

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




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

Propagation Rules Are Enforced by a Proxy

- In the example below, the 2nd propagation rule does not get applied because the call does not go through a proxy

```
public class ClientServiceImpl implements ClientService {
        @Transactional(propagation=Propagation.REQUIRED)
        public void update1() {
            update2();
        }
        @Transactional(propagation=Propagation.REQUIRES_NEW)
        public void update2() {
        }
    }
}
```

Does not get applied because the call is internal

Propagation Levels and their Behaviors

Propagation Type	If NO current transaction (txn) exists	If there IS a current transaction (txn)
MANDATORY	Throw exception	Use current txn
NEVER	Don't create a txn, run method without a txn	Throw exception
NOT_SUPPORTED	Don't create a txn, run method without a txn	Suspend current txn, run method without a txn
SUPPORTS	Don't create a txn, run method without a txn	Use current txn
REQUIRED (default)	Create a new txn	Use current txn
REQUIRES_NEW	Create a new txn	Suspend current txn, create a new independent txn
NESTED	Create a new txn	Create a new nested txn

Agenda

- Why use Transactions?
- Java Transaction Management
- Spring Transaction Management
- Transaction Propagation
- **Rollback Rules**
- Testing

Default Behavior

- By default, a transaction is rolled back only if a *RuntimeException* has been thrown
  - Could be any kind of *RuntimeException*: *DataAccessException*, *HibernateException* etc.

```
public class RewardNetworkImpl implements RewardNetwork {

  @Transactional
  public RewardConfirmation rewardAccountFor(Dining d) {
    // ...
    throw new RuntimeException();

  }
}
```

Triggers a rollback

rollbackFor and noRollbackFor

- Default settings can be overridden with *rollbackFor* and/or *noRollbackFor* attributes

```
public class RewardNetworkImpl implements RewardNetwork {

  @Transactional(rollbackFor=MyCheckedException.class,
                 noRollbackFor={JmxException.class, MailException.class})
  public RewardConfirmation rewardAccountFor(Dining d) throws Exception {
      // ...
  }

}
```

Agenda

- Why use Transactions?
- Java Transaction Management
- Spring Transaction Management
- Transaction Propagation
- Rollback Rules
- **Testing**

@Transactional within Integration Test

- Annotate test method (or class) with @Transactional
  - Runs test methods in a transaction
  - Transaction will be *rolled back* afterwards
    - No need to clean up your database after testing!

```
@SpringJUnitConfig(RewardsConfig.class)
public class RewardNetworkTest {
    @Test @Transactional
    public void testRewardAccountFor() {
      ...
    }
}
```

This test is now transactional

Controlling Transactional Tests

```
@SpringJUnitConfig(RewardsConfig.class)
@Transactional
public class RewardNetworkTest {

    @Test
    @Commit
    public void testRewardAccountFor() {
      ...    // Whatever happens here will be committed
    }
}
```

Make *all* tests transactional

Commit transaction at end of test