# Application Delivery Fundamentals 2.0

## Spring /JPA Transaction Management

# Course Goals / Objectives

- At the end of this module, participants will be able to:
  - Demonstrate an understanding Transaction Management Basics
  - Identify the different Transaction Isolation Levels
  - Explain the need for JDBC Transaction Management
  - Understand the Spring Transaction Management
  - Understand the Programmatic approach of Spring JPA Transaction Management

# Agenda



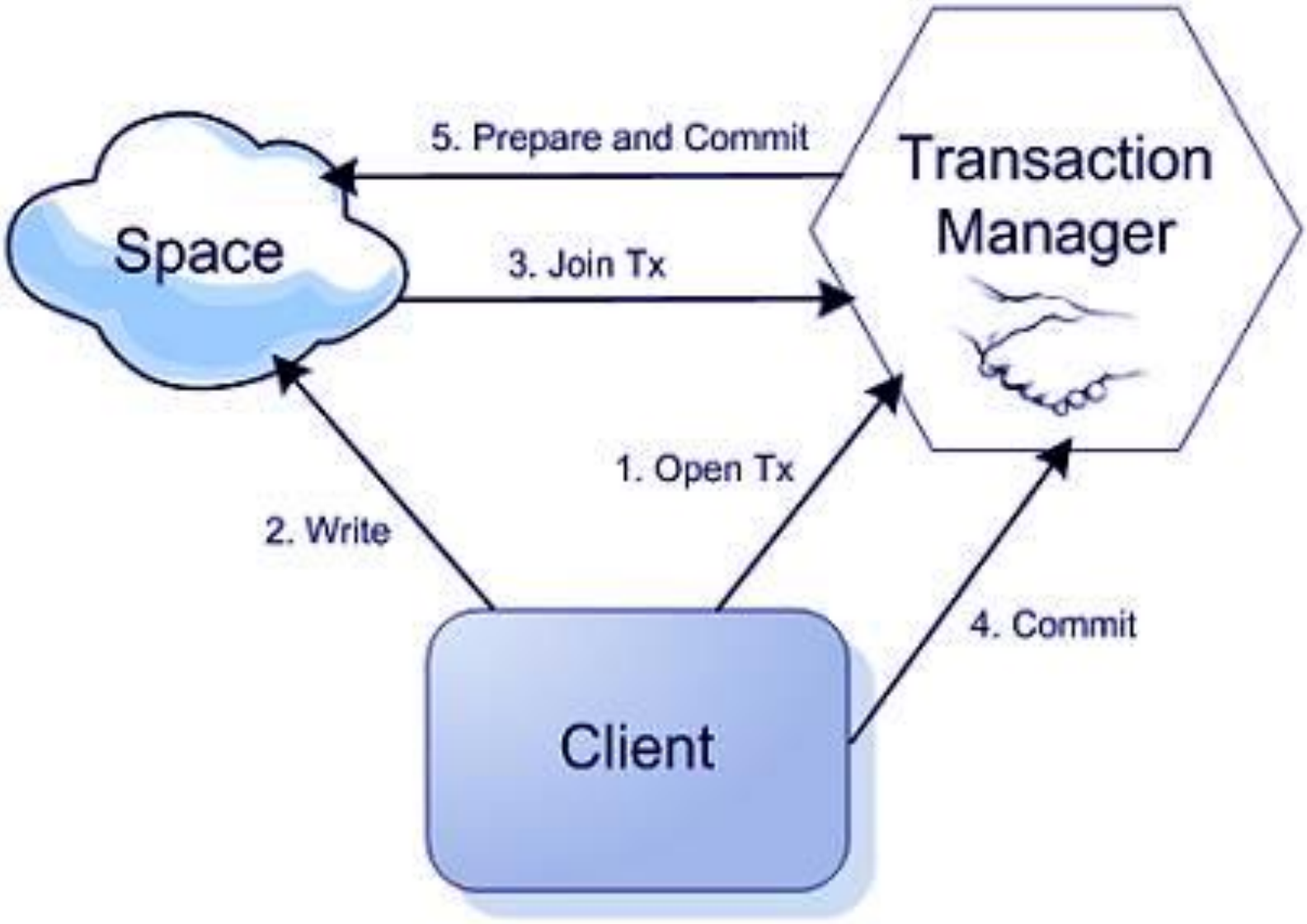
- This module will cover the following topics:
  - Transaction Management Basics
  - JDBC Transaction Management
  - Introduction to Spring Transaction
  - Spring Transaction Properties
  - Spring JPA Transaction Management
  - Spring Transactional Attribute

# Transaction Management Basics

- What is Transaction ?
  - A transaction can be defined as an indivisible unit of work comprised of several operations, all or none of which must be performed in order to preserve data integrity.
  - In computer programming, a transaction usually means a sequence of information exchange and related work (such as updating multiple task at the same time)
  - It is treated as a unit for the purposes of satisfying a request and for ensuring database integrity.
  - For a transaction to be completed and database changes to made permanent, a transaction has to be completed entirely
- Transaction Example Contd…

# Transaction Management Basics (Continued…)

- For Example :
  - An online purchase where the customer use his Credit/Debit card and purchase things where the amount has to be debited from one account and the same should be credited in another account so this is treated as single transaction. The transaction is treated successful when the database is actually changed to reflect the process. If something happens before the transaction is successfully completed, any changes to the database must be kept track of so that they can be undone.



# JDBC Transaction Management

- JDBC transactions are controlled through the Connection object. There are two modes for managing transactions within JDBC:
  - auto-commit
  - manual-commit.
- The setAutoCommit method is used to switch between the two modes.

# JDBC Transaction Management

- **Auto-commit Mode**
  - Auto-commit mode is the default transaction mode for JDBC. When a connection is made, it is in auto-commit mode until setAutoCommit is used to disable auto-commit.
  - In auto-commit mode each individual statement is automatically committed when it completes successfully, no explicit transaction management is necessary. However, the return code must still be checked, as it is possible for the implicit transaction to fail.

(Contd…)

# JDBC Transaction Management (Continued…)

- **Manual-commit Mode**
  - When auto-commit is disabled, i.e. manual-commit is set, all executed statements are included in the same transaction until it is explicitly completed.
  - When an application turns auto-commit off, the next statement against the database starts a transaction. The transaction continues either the commit or the rollback method is called. The next command sent to the database after that starts a new transaction.

# Introduction to Spring Transaction Management

- Provides flexible and powerful layer for transaction management
- Provides a consistent programming model across different transaction APIs.
- Masks/hides differences between local and distributed/global (JTA) transactions for the client code
- Choice of transaction demarcation strategy and choice of transaction manager are independent of each other.

(Contd…)

# Introduction to Spring Transaction Management

- For any transaction manager chosen, both programmatic and declarative transaction demarcation work equally well.
- No application code needs to be changed when the transaction manager is changed – only configuration files are updated.
- Spring provides implementations of Transaction managers for both Hibernate and JPA:
  - HibernateTransactionManager
  - JpaTransactionManager

(Contd…)

# Introduction to Spring Transaction Management

- DataSource Transaction manager -
- We can use DataSourceTransactionManager for simple JDBC persistence mechanism. Sample configuration of DataSourceTransactionManager looks like below
- <bean id=”transactionManager”
- class=”org.springframework.jdbc.datasource.DataSourceTransactionManager>
- <property name=”dataSource” ref= “datasource” />
- </bean>

# Introduction to Spring Transaction Management

- Hibernate Transaction manager – Hibernate transaction manager should be used when our application is using Hibernate. Sample configuration of HibernateTransactionManager looks like below
- <bean id=”transactionManager”
- class=”org.springframework.orm.hibernate3.HibernateTransactionManager>
- <property name=”sessionFactory” ref= “sessionFactory” />
- </bean>

# Introduction to Spring Transaction Management

- Jdo Transaction manager –Use below configuration to use Java data object transaction manager .
- <bean id=”transactionManager”
- class=”org.springframework.orm.jdo.JdoTransactionManager>
- <property name=”persistanceManagerFactory” ref= “persistanceManagerFactory” />
- </bean>

# Introduction to Spring Transaction Management

- Jta Transaction manager – If our transaction is across multiple data sources than we need to use Java Transactions API transactions . Internally JTA implementation handles transaction responsibility.
- Use below configuration to configure JTA transaction manager.

- <bean id=”transactionManager”
- class=”org.springframework.transaction.jta.JtaTransactionManager>
- <property name=”transactonManagerName” ref=“java:/TransactionManager” />
- </bean>

# Introduction to Spring Transaction Management

- These can be used to back Spring annotation-based transaction management
  - Will automatically commit or rollback appropriately on annotated methods

# Spring :Transaction Properties

## Transaction Properties

- All transactions share these properties: atomicity, consistency, isolation, and durability (represented by the acronym ACID).
  - *Atomicity*: This implies indivisibility; any indivisible operation (one which will either complete fully or not at all) is said to be atomic.
  - *Consistency*: A transaction must transition persistent data from one consistent state to another. If a failure occurs during processing, the data must be restored to the state it was in prior to the transaction.

(Contd..)

# Spring :Transaction Properties

- *Isolation*: Transactions should not affect each other. A transaction in progress, not yet *committed* or *rolled back* (these terms are explained at the end of this section), must be isolated from other transactions. Although several transactions may run concurrently, it should appear to each that all the others completed before or after it; all such concurrent transactions must effectively end in sequential order.
- *Durability*: Once a transaction has successfully committed, state changes committed by that transaction must be durable and persistent, despite any failures that occur afterwards.

# Spring : Transaction Choice

- Traditionally developers have two choices for transaction management.
  - Global transactions.
  - Local transactions.

    - Global Transactions
      - Global transactions are managed by the application server, using the Java Transaction API (JTA).
      - global transactions provide the ability to work with multiple transactional resources (typically relational databases and message queues).

# Spring : Transaction Choice

- Local transactions
  - Local transactions are resource-specific, the most common example would be a transaction associated with a JDBC connection.
  - With local transactions, the application server is not involved in transaction management and cannot help ensure correctness across multiple resources.

# Spring : JPA Transactions

- Uses JPATransactionManager – daoContext.xml

```
<!-- Setup Transactions -->
        <bean id="transactionManager"
        class="org.springframework.orm.jpa.JpaTransactionManager"
                        p:entityManagerFactory-ref="entityManagerFactory" />
        <!-- activates @Transactional -->
        <tx:annotation-driven transaction-manager="transactionManager" />
```

- JpaTransactionManager manages JTA transactions using the configured JTA DataSource from entityManagerFactory.
- @Transactional annotation abstracts the transactional semantics – begin, commit and rollback from the developer.

(Contd…)

# Spring : JPA Transactions

- JPaTransactionManager begins a JTA transaction, invokes the method and either commits or rollback the global transaction.

```
@Transactional
public Order modifyOrder( Order entity ) {
        setTotalOrderAmount( entity );
        this.orderDao.persist( entity );
        return entity;
        }
```

# Spring : @Transactional

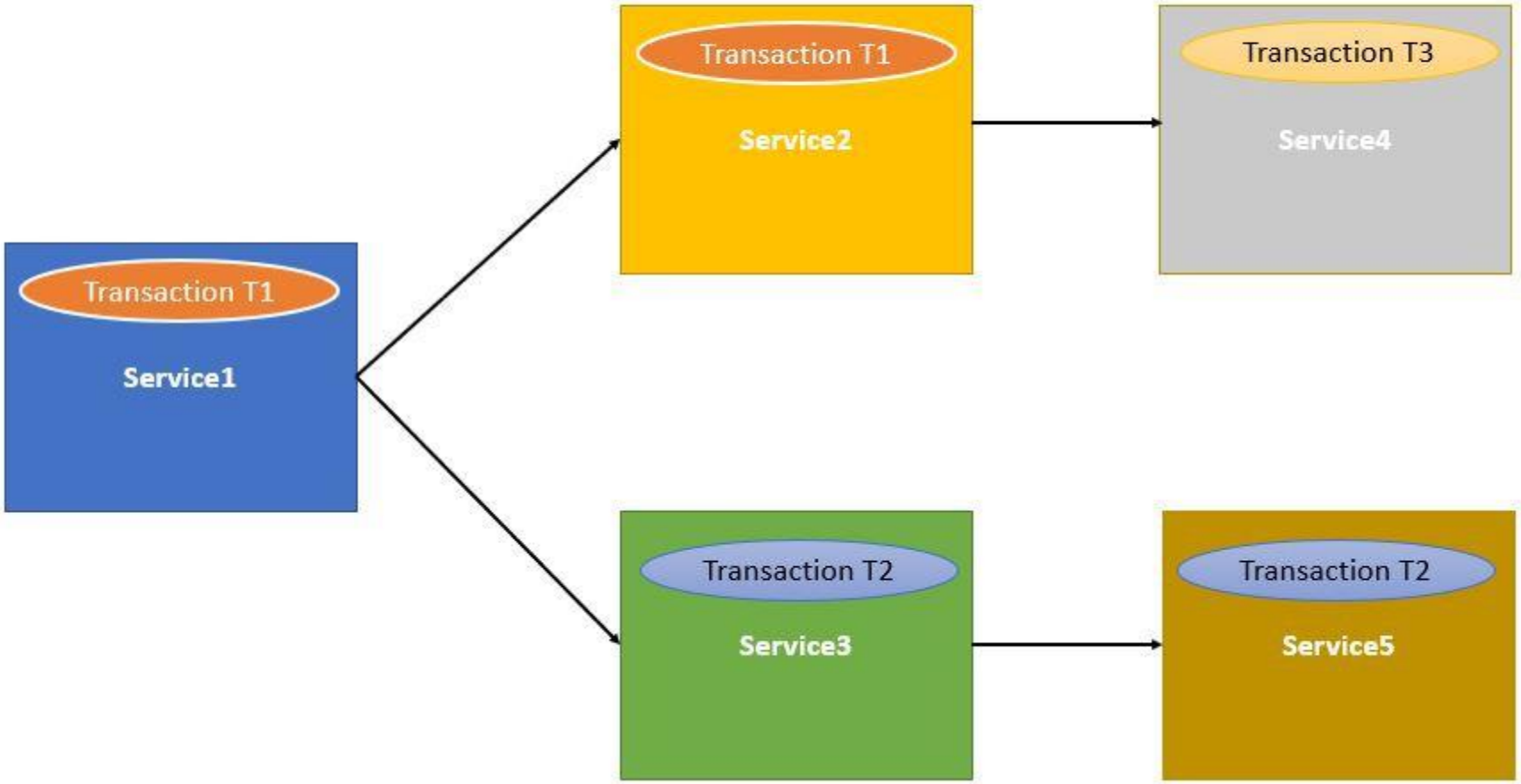
- Declarative transaction management was only available in EJB containers. But now Spring offers support for declarative transactions to POJOs.
- This is done through spring AOP frame work using auto proxies.
- Spring enables declarative transaction management to be applied to any class or method.
- If applied at class level will apply to all methods including getters and setters
- It offers configurable propagation and *rollback rules.*

# Spring : Transaction Attributes

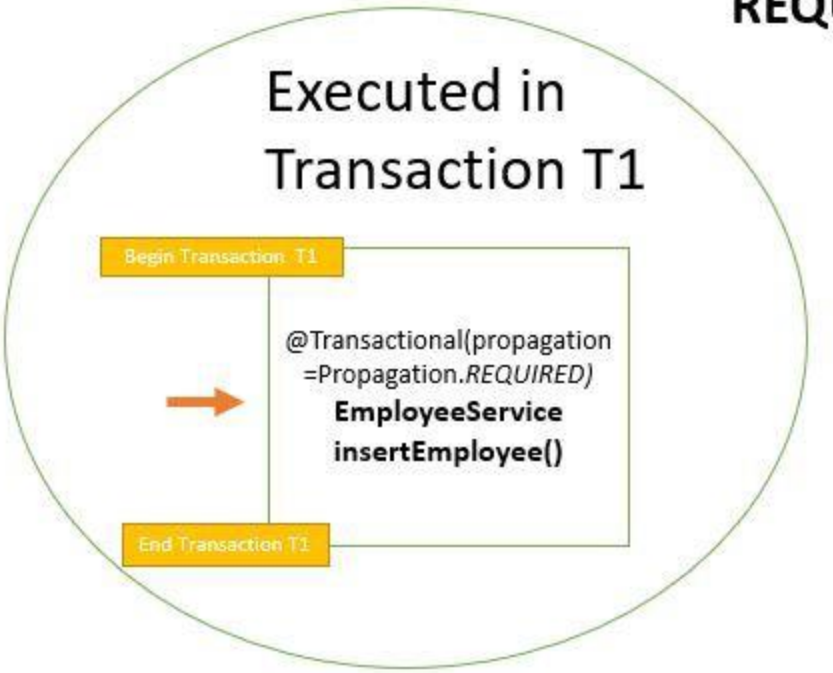
- All the transaction manager have the following transaction attributes that can be configured.
  - Propagation behavior
  - Isolation level
  - Read-only hints
  - The transaction timeout period

(Contd…)

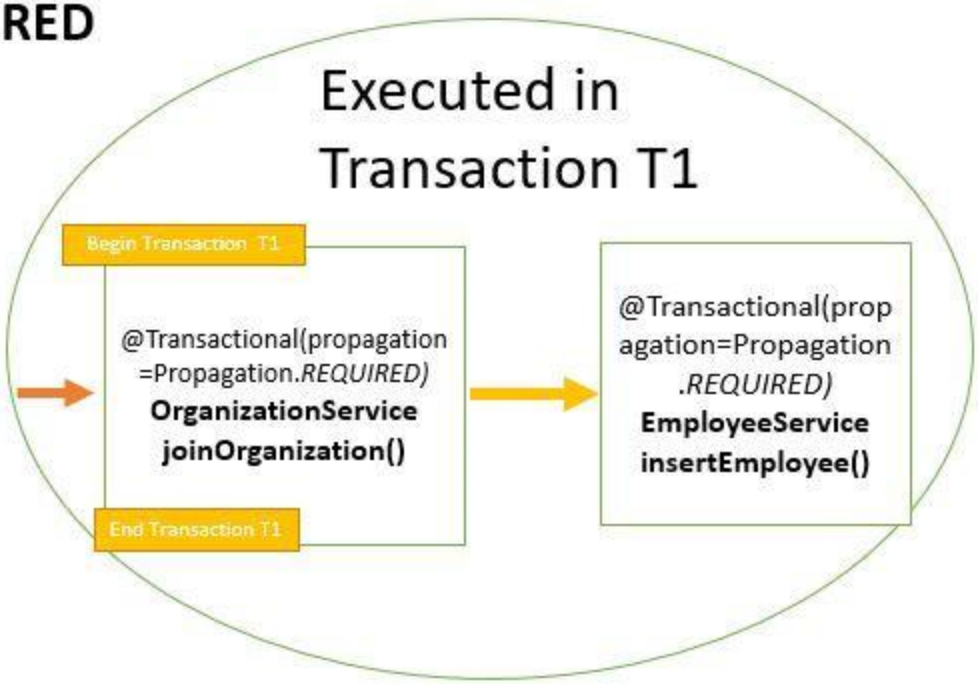
# TRANSACTION PROPAGATION



# TRANSACTION PROPAGATION-REQUIRED




If the insertEmployee() method **is called directly** it creates its **own new transaction**.

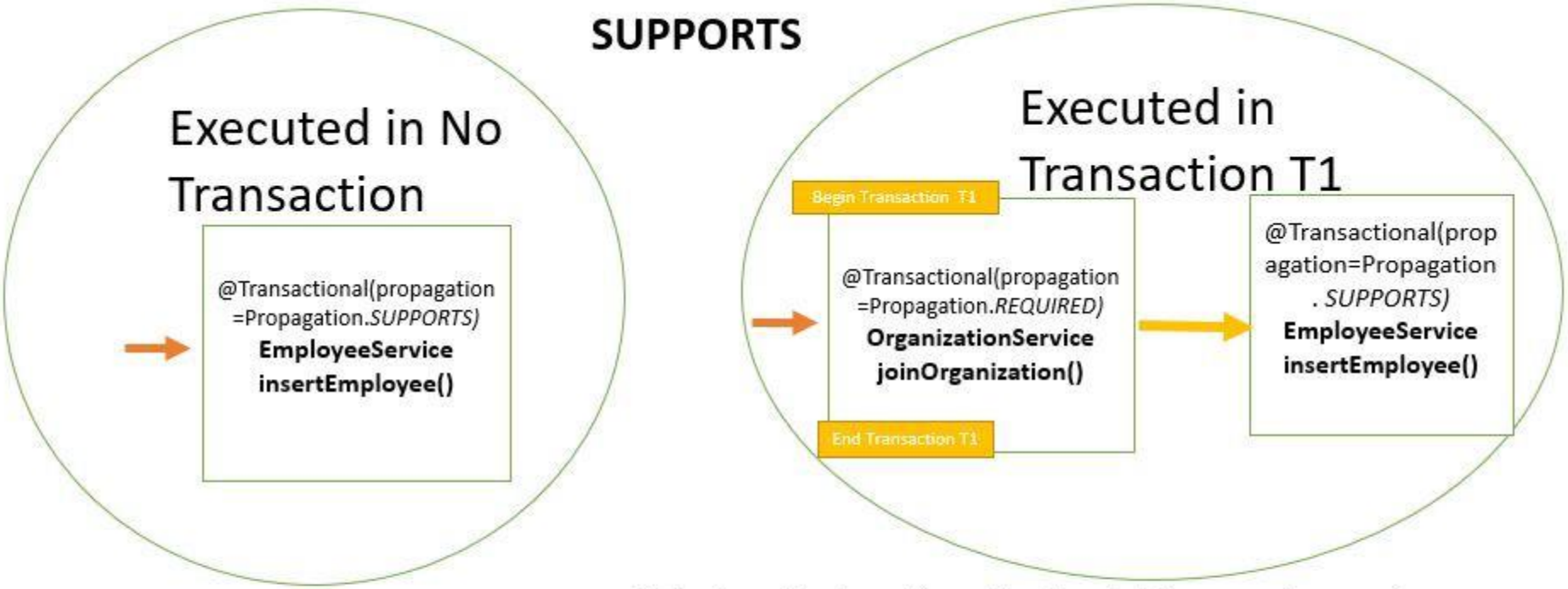



If the insertEmployee() **method is called from another service –**

1. If the calling service has a transaction then method **makes use of the existing transaction**.
2. If the calling service does not have a transaction then the method **will create new transaction**.

So in the case of **REQUIRED** the insertEmployee() method makes use of the calling service transaction if it exists else creates its own.

# TRANSACTION PROPAGATION-SUPPORTS



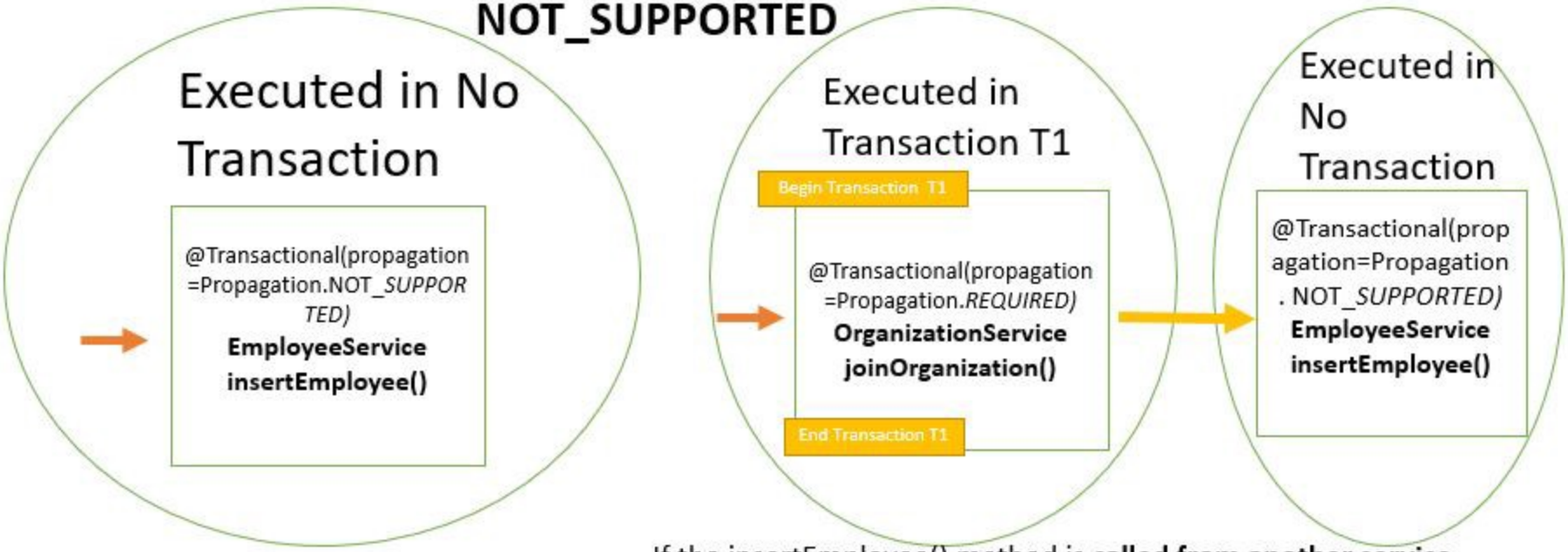

If the insertEmployee() method **is called directly** it does **not create own new transaction**

If the insertEmployee() method is **called from another service**

1. If the calling service method has a transaction then method **makes use of the existing transaction**.
2. If the calling service method does not have a transaction then the method **will not create a new transaction**.

So in the case of **SUPPORTS** the insertEmployee() method will create make use of calling service transaction if it exists. Else it will not create a new transaction but run without transaction.

# TRANSACTION PROPAGATION-NOT_SUPPORTED



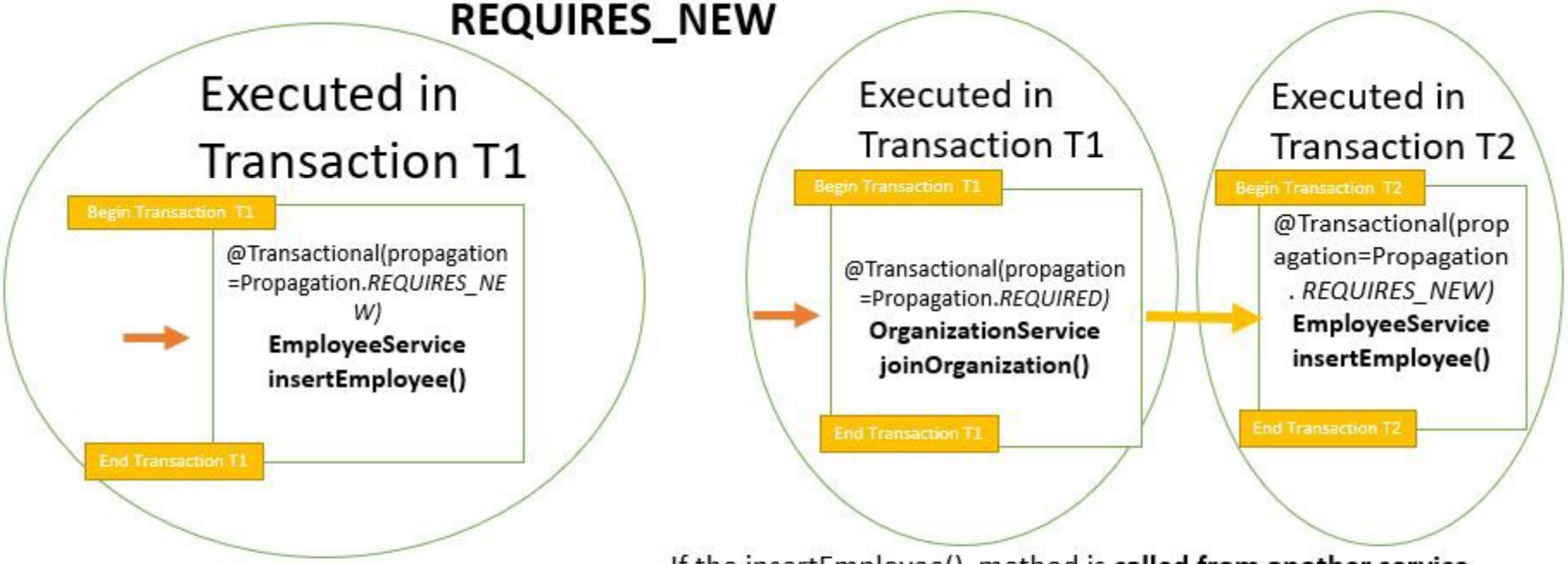

If the insertEmployee() method is **called directly** it does **not create own new transaction**

If the insertEmployee() method is **called from another service**

1. If the calling service method has a transaction then method **does not make use of the existing transaction neither does it create its own transaction. It run without transaction**.
2. If the calling service method does not have a transaction then the method **will not create a new transaction and run without transaction**.

So in the case of **NOT_SUPPORTED** the insertEmployee() method never run in transaction.

# TRANSACTION PROPAGATION-REQUIRES_NEW




If the insertEmployee() method **is called directly** it creates its **own new transaction**

If the insertEmployee() method is **called from another service**

1. If the calling service method has a transaction then method **does not make use of the existing transaction but creates its own transaction**.
2. If the calling service method does not have a transaction the method **will create a new transaction**

So in the case of **REQUIRES_NEW** the insertEmployee() method always creates a new Transaction

# TRANSACTION PROPAGATION-NEVER

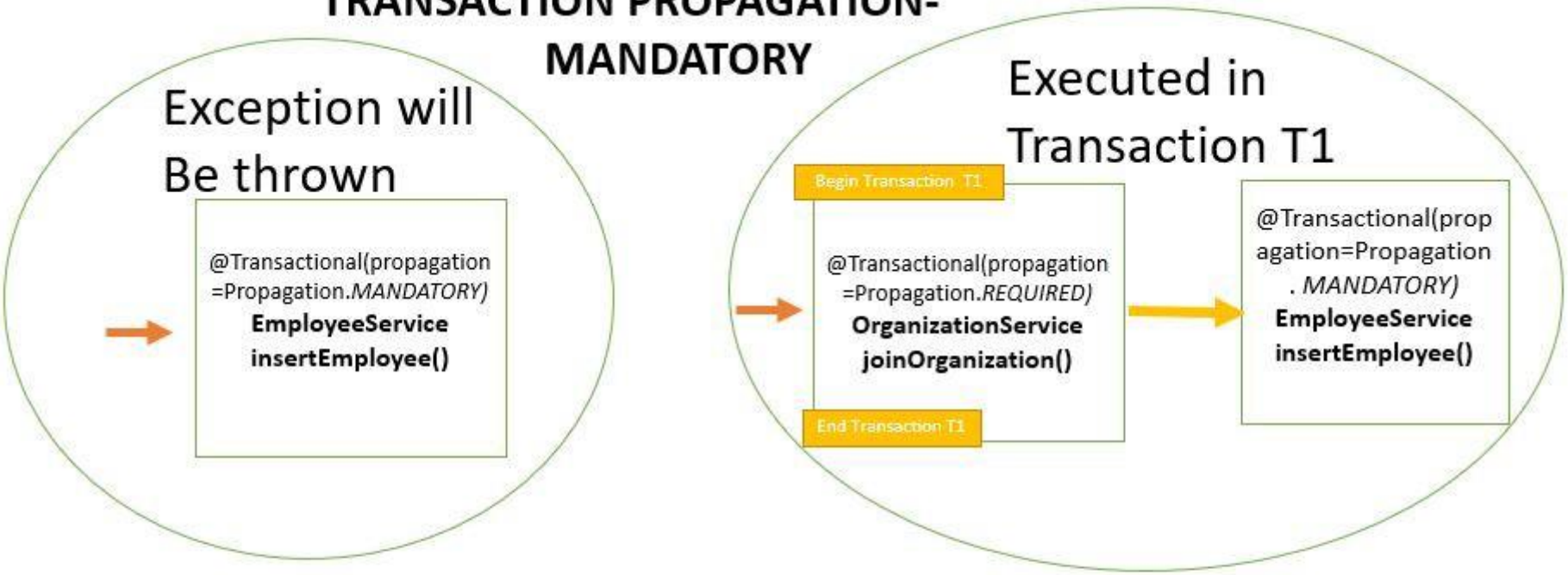

If insertEmployee() method **is called directly** it creates it does not create a new transaction

If the insertEmployee() method is **called from another service**

1. If the calling service method has a transaction then method **throws an exception**.
2. If the calling service method does not have a transaction the method **will not create a new one and run without transaction.**

So in the case of **NEVER** the insertEmployee() method never uses Transaction

# TRANSACTION PROPAGATION-MANDATORY




If insertEmployee() method is **called directly** it will **throw an exception**.

If the insertEmployee() method is **called from another service**

1. If the calling service method has a transaction then method **makes use of the existing transaction**.
2. If the calling service method does not have a transaction the method **will throw an exception**

So in the case of **MANDATORY** the insertEmployee() method always needs the calling service to have a transaction else exception is thrown

# Spring : Transaction Attributes

- **Propagation behavior** defines the boundaries of the transaction with respect to the client and to the method being called. Spring defines seven distinct propagation behaviors, as cataloged.
  - *PROPAGATION_MANDATORY*: Indicates that the method must run within a transaction. If existing transaction is in progress, an exception will be thrown

(Contd…)

# Spring : Transaction Attributes

- *PROPAGATION_NESTED*: Indicates that the method should be run within a nested transaction if an existing transaction is in progress. The nested transaction can be committed and rolled back individually from the enclosing transaction.
- *PROPAGATION_REQUIRED*: Beware that vendor support for this propagation behavior is spotty at best. Consult the documentation for your resource manager to determine if nested transactions are supported.

(Contd…)

# Spring : Transaction Attributes

- *PROPAGATION_NEVER*: Indicates that the current method should not run within a transactional context. If there is an existing transaction in progress, an exception will be thrown.
- *PROPAGATION_NOT_SUPPORTED*:Indicates that the method should not run within a transaction. If an existing transaction is in progress, it will be suspended for the duration of the method. If using JTATransactionManager,access to TransactionManager is required.

# Spring : @Transactional attributes

```
@Transactional(
propagation=Propagation.REQUIRED,
isolation=Isolation.READ_COMMITTED,
rollbackFor = {RuntimeException.class},
readOnly = true )
public Order findEntry( Object pk ) {
        return (Order)this.orderDao.findEntry( pk );
        }
```

# Spring JPA Transaction Management (1 of 2)

- Transaction semantics:

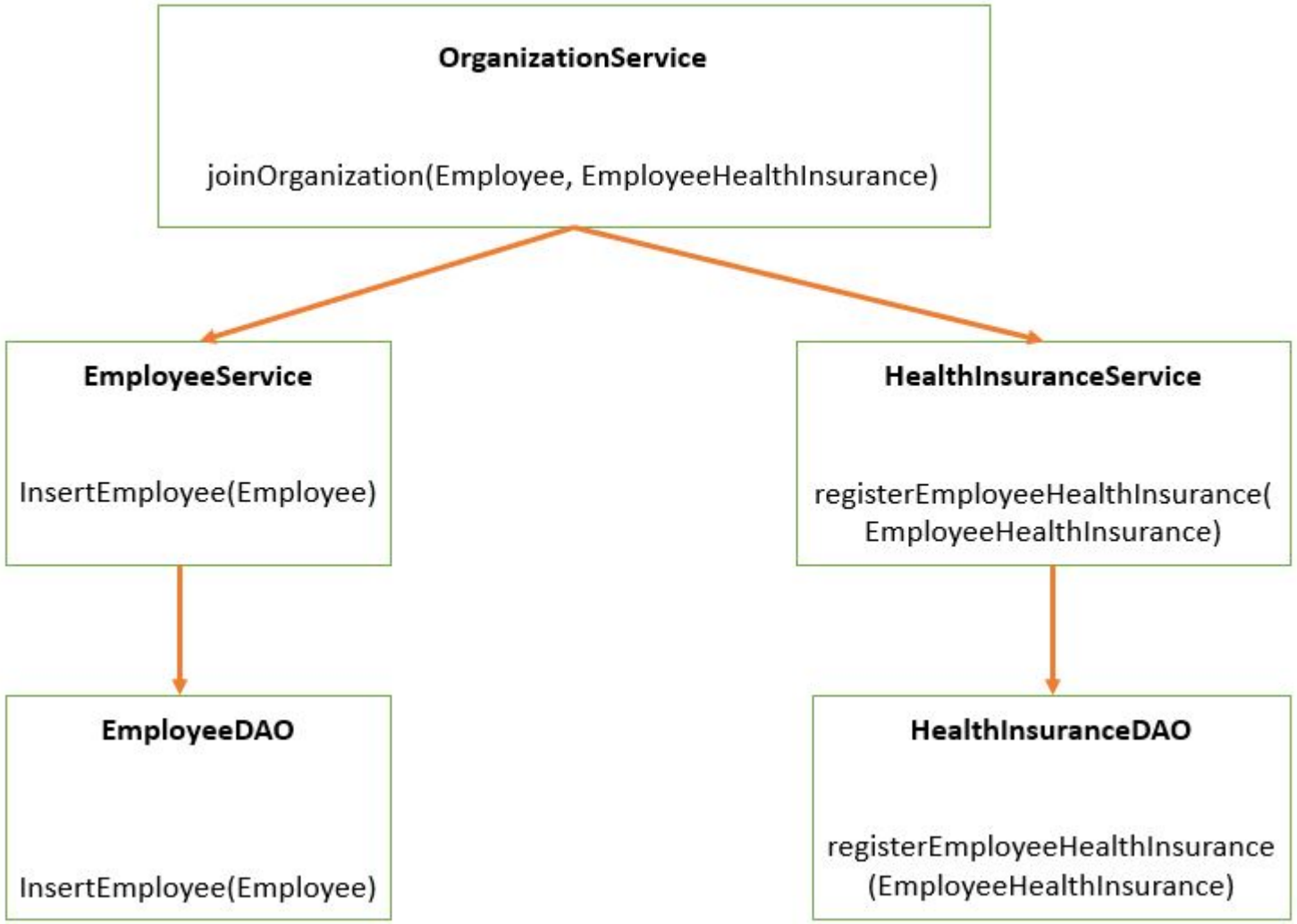
| Semantic | Description | Default Value |
|---|---|---|
| Propagation | Defines scope and interaction of transactions | PROPAGATION_REQUIRED |
| Isolation | Defines concurrent transactions' capacity to view data during another transaction | ISOLATION_DEFAULT (uses the default value of the underlying data source) |
| Timeout | Defines the number of seconds a transaction can run | TIMEOUT_DEFAULT (can be changed to any positive integer/number) |
| Read-only | (Boolean) Indicates whether a transaction can modify any persistent state | FALSE (transactions are read/write) |
| ExceptionHandling | Defines how the checked and runtime exceptions should be handled | RuntimeExceptions: Rollback; Normal/Checked Exceptions: Commit |

# Spring JPA Transaction Management (2 of 2)

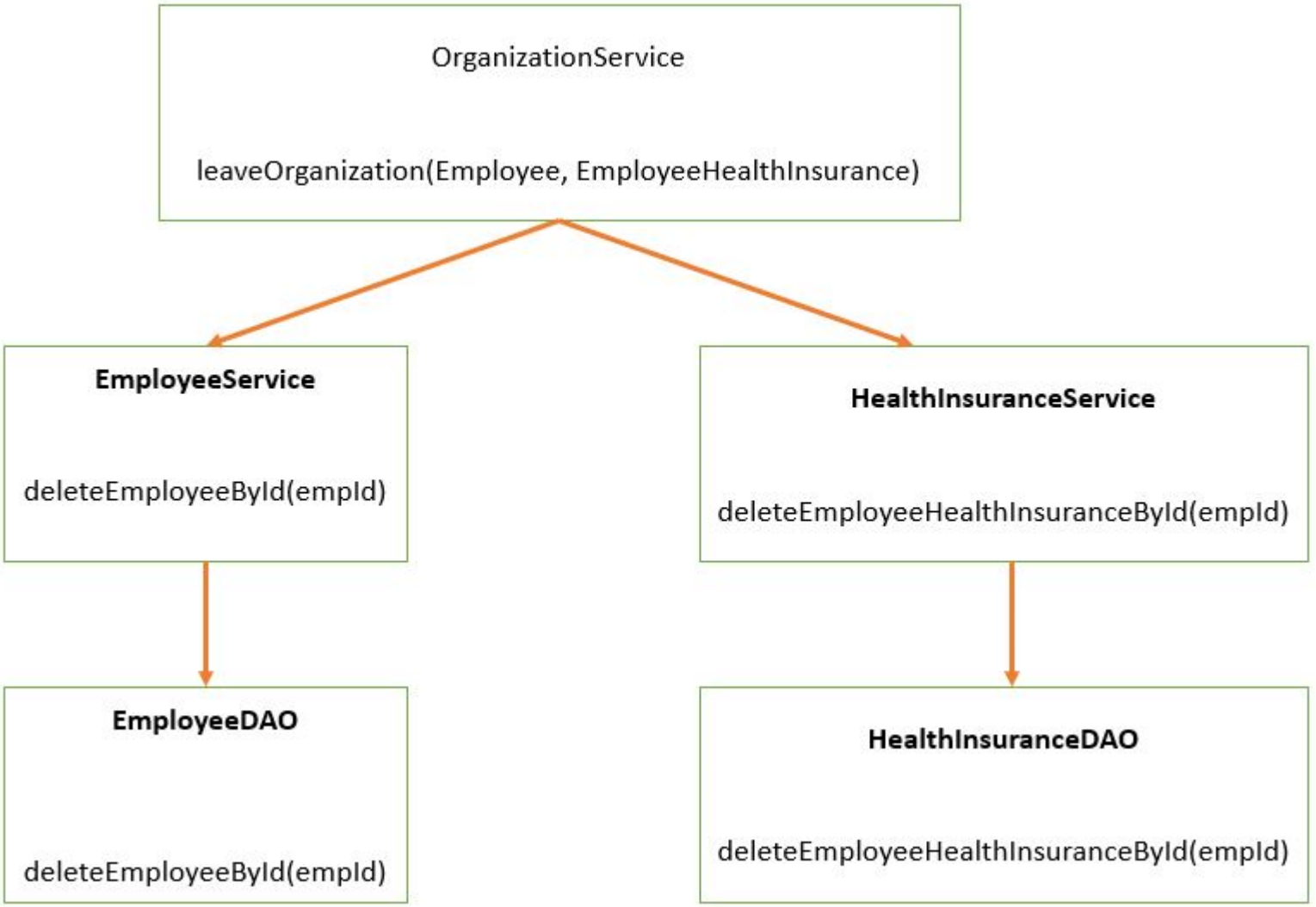
- Transaction semantics can be defined either programmatically or declaratively:

| Programmatic | Declarative |
|---|---|
| Instantiate an instance of ***DefaultTransactionDefinition*** class that implements the TransactionDefinition and sets the transaction properties | Declare the transaction semantics in a configuration file in this element <tx:attributes> |

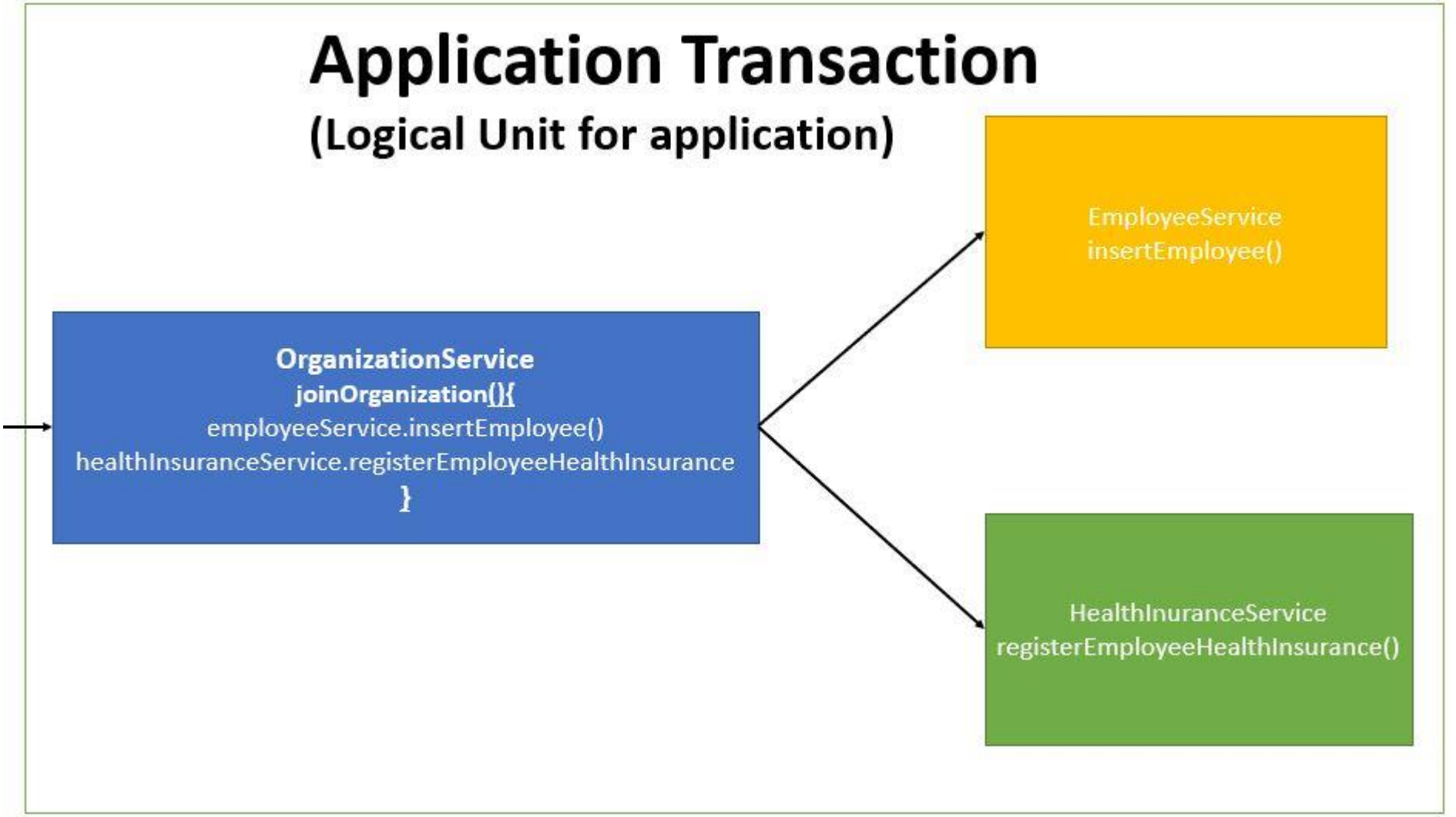
# Join Work flow



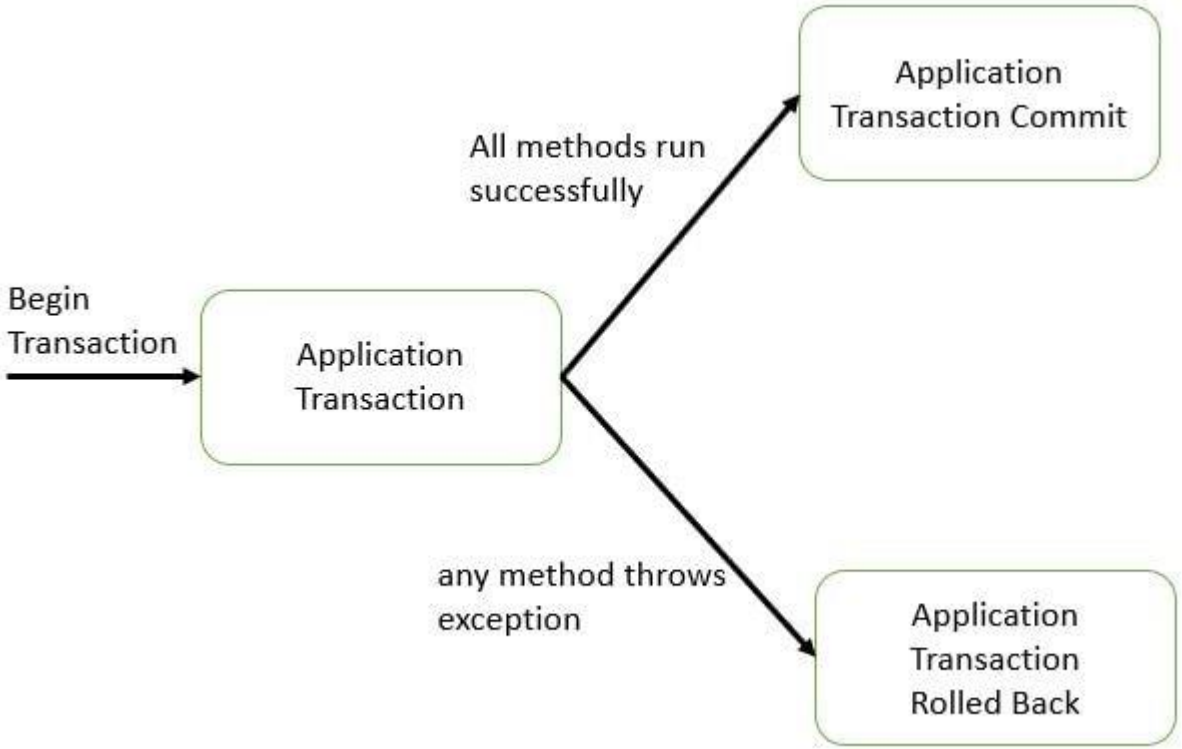
# Exit Work flow



# Join Organization

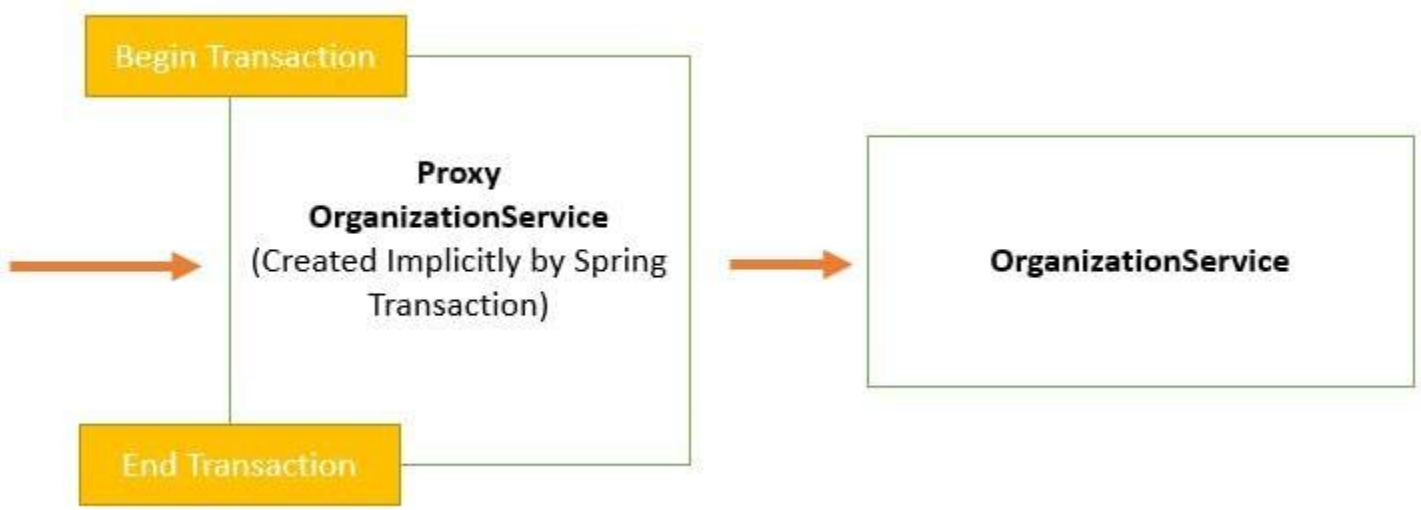


# Join Organization



# Join Organization

# Join Organization



# See It

- **Demonstration**:
  - Faculty will demonstrate how to implement JPA  Transaction in Spring application.
- **Environment** – Eclipse
- **Time** – 30 Minutes

Instructions Contd…

# See It- Instructions

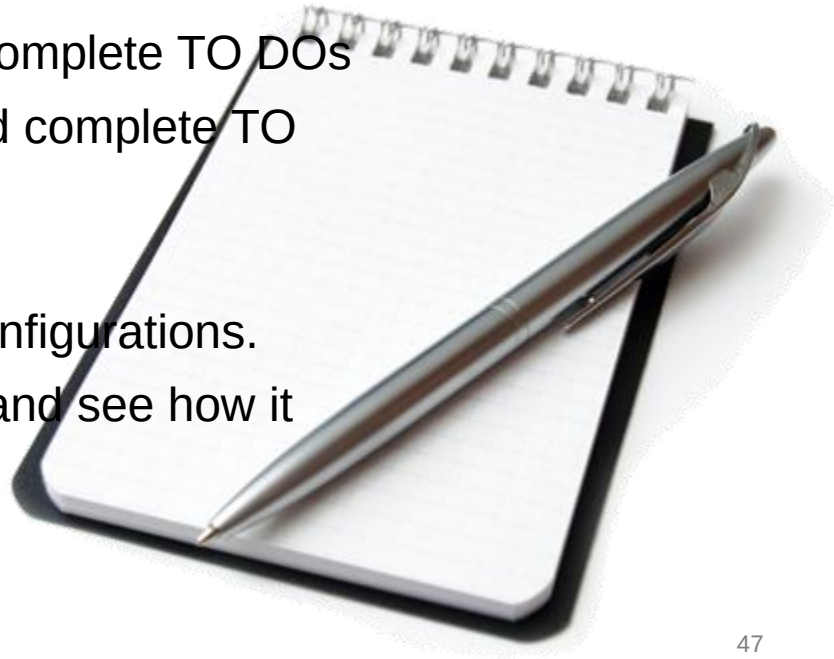
- **File**
  - MuseumDaoImpl.java / JPAMuseumSample.java / applicationContext.xml
- **Steps**
  - Open the project ADFExtensionCodebaseM10JPATransaction_participant in Eclipse
  - Open JPAMuseumSample.java and run as java application.
  - Refer the log for queries and result.
  - Refer applicationContect.xml for JPA configurations.
  - Try various combinations, refer the log and see how it behaves.

# Try It

- **Activity**
  - Insert the Customer details in to the Table using Transaction Management
- **Environment** – Eclipse
- **Time** – 90 Minutes

# Try It- Instructions

- **File**
  - CustomerDaoImpl.java / JPACustomerActivity.java / applicationContext.xml
- **Steps**
  - Open the project ADFExtensionCodebaseM10JPATransaction_participant in Eclipse
  - Complete CustomerDaoImpl.java and complete TO DOs
  - Complete JPACustomerActivity.java and complete TO DOs and run as java application.
  - Refer the log for queries and result.
  - Refer applicationContect.xml for JPA configurations.
  - Try various combinations, refer the log and see how it behaves.



# Course / Module Summary

- Spring transaction management is a set of computations producing changes to recoverable data which demonstrate atomicity, consistency, isolation, and durability (ACID) properties.
- Spring transaction management support can be availed either programmatically or declaratively.

# Questions