

제 8 강 메이븐 프로젝트 : 트랜잭션 심화

1. 트랜잭션이란?

- 트랜잭션은 성공적으로 처리되거나 또는 하나라도 실패하면 완전히 실패를 해야 하는 경우에 사용된다. 예를 들어 도서구매 과정을 보면 ① 결제를 수행한다. ② 결제 내역을 저장한다. ③ 구매 내역을 저장한다. 이렇게 3단계로 이루어질 것이다. 이와 같은 과정은 구매 시 반드시 성공적으로 이루어져야 한다. 한 가지 과정이라도 실패할 경우 반드시 모든 과정이 취소되어야 한다.

2. 트랜잭션의 네 가지 특성 : ACID

① Atomic

: 트랜잭션은 한 개 이상의 동작을 논리적으로 한 개의 작업 단위로 묶는다. 원자성은 트랜잭션 범위에 있는 모든 동작이 모두 실행되거나 또는 모두 실행이 취소됨을 보장한다.

② Consistent

: 트랜잭션이 종료되면 시스템은 비즈니스에서 기대하는 상태가 된다. 예를 들어 서적 구매 트랜잭션이 성공적으로 실행되면 결제 내역, 구매 내역, 잔고 정보가 비즈니스에 맞게 저장되고 변경된다.

③ Isolated

: 트랜잭션은 다른 트랜잭션과 독립적으로 실행되어야 하며, 서로 다른 트랜잭션이 동일한 데이터에 동시에 접근할 경우 알맞게 동시 접근을 제어해야 한다.

④ Durable

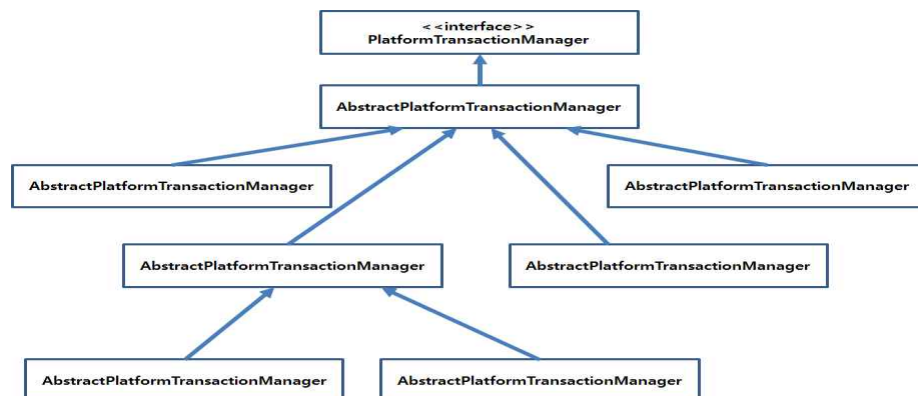
: 트랜잭션이 완료되면 그 결과는 지속적으로 유지되어야 한다.

3. 스프링의 트랜잭션 지원

- 스프링 코드 기반의 트랜잭션 처리뿐만 아니라 선언적 트랜잭션을 지원하고 있다.
- 개발자가 직접적으로 트랜잭션의 범위를 코드 수준에서 정의하고 싶은 경우에는 스프링이 제공하는 트랜잭션 템플릿 클래스를 이용해서 손쉽게 트랜잭션 범위를 지정할 수 있다.
- 스프링은 데이터베이스 연동 기술에 상관없이 동일한 방식으로 트랜잭션을 처리할 수 있도록 하고 있다.

① 스프링은 PlatformTransactionManager 설정

- 스프링은 PlatformTransactionManager 인터페이스를 이용해서 트랜잭션 처리를 추상화하였고 데이터베이스 연동 기술에 따라 알맞은 PlatformTransactionManager 구현 클래스를 제공하고 있다.



☐ 스프링의 TransactionManager는 각 기술에 알맞은 트랜잭션 처리를 수행한다.

② JDBC 기반 트랜잭션 관리자 설정

☐ JDBC나 iBatis와 같이 JDBC를 이용해서 데이터베이스를 연동을 처리하는 경우 다음과 같이 DataSourceTransactionManager를 트랜잭션 관리자로 사용한다.

```
<bean id="dataSource" class="org.apache.commons.dbcp2.BasicDataSource" destroy-method="close"
      p:driverClassName="oracle.jdbc.driver.OracleDriver" p:username="mytest" p:password="mytest" />

<bean id="transactionManager"
      class="org.springframework.jdbc.datasource.DataSourceTransactionManager"
      p:dataSource-ref="dataSource"/>
```

☐ DataSourceTransactionManager는 dataSource 프로퍼티를 통해서 전달받은 DataSource로 부터 Connection을 가져온 뒤 Connection의 commit(), rollback()등의 메서드를 처리한다.

③ 하이버네이트 트랜잭션 관리자 설정

☐ 하이버네이트를 사용하는 경우에는 HibernateTransactionManager를 트랜잭션 관리자로 사용한다.

```
<bean id="transactionManager"
      class="org.springframework.orm.hibernate3.HibernateTransactionManager"
      p:sessionFactory-ref="sessionFactory" />

<bean id="sessionFactory"
      class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
    <property name="dataSource" ref="dataSource" />
    <property name="mappingResource">
        ...
    </property>
    <property name="hibernateProperties">
        ...
    </property>
</bean>
```

☐ HibernateTransactionManager는 sessionFactory 프로퍼티를 통해서 전달받은 하이버네이트 Session으로부터 하이버네이트 Transaction을 생성한 뒤 Transaction을 이용해서 트랜잭션을 관리한다.

④ JPA 트랜잭션 관리자 설정

☐ JPA를 사용할 경우에는 JpaTransactionManager를 트랜잭션 관리자로 사용한다.

```
<bean id="entityManagerFactory"
      class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
    ...
</bean>
```

```
<bean id="transactionManager"
      class="orm.springframework.orm.jpa.JpaTransactionManager"
      p:entityManagerFactory-ref="entityManagerFactory" />
```

- ☐ JpaTransactionManager는 entityManagerFactory 프로퍼티를 통해 전달받은 EntityManagerFactory를 이용해서 트랜잭션을 관리한다.

⑤ JTA 트랜잭션 관리자 설정

- ☐ 다중 자원에 접근하는 경우 JTA(Java Transaction API)를 이용해서 트랜잭션을 처리하는데 이 때 JtaTransactionManager를 사용한다.
- ☐ transactionManagerName 프로퍼티를 이용해서 JTA TransactionManager를 구할 수 있는 JNDI 이름을 설정해야한다.

```
<bean id="transactionManager"
      class="org.springframework.transaction.jta.JtaTransactionManager"
      p:transactionManagerName="java:comp/TransactionManager" />
```

- ☐ 컨테이너가 제공하는 TransactionManager를 사용하지 않고 TransactionEssentials과 같은 오픈소스 라이브러리를 이용해서 로컬 JTA를 사용할 경우에는 userTransaction 프로퍼티를 이용해서 UserTransaction을 직접 설정할 수 있다.

```
<bean id="transactionManager" class="org.springframework.transaction.jta.JtaTransactionManager"
      depends-on="userTransactionService">
    <property name="transactionManager" ref="atomikosTransactionManager" />
    <property name="userTransaction" ref="atomikosUserTransaction"/>
</bean>
<bean id="userTransactionService" .../>
<bean id="atomikosTransactionManager" .../>
<bean id="atomikosUserTransaction" .../>
```

- ☐ JtaTransactionmanager는 javax.transaction.UserTransaction의 commit(), rollback()등을 이용해서 트랜잭션을 처리한다.

4. TransactionTemplate을 이용한 트랜잭션 처리

- ☐ 아래와 같은 상품구매 처리 코드를 생각해 보자

```
1 public class PlaceOrderServiceImpl implements PlaceOrderService{
2     ...
3     @Override
4     public PurchaseOrderResult order(PurchaseOrderRequest orderRequest)
5         throws ItemNotFoundException{
6         Item item = itemDao.findById(orderRequest.getItemId());
7         if(item == null) throw new ItemNotFoundException(orderRequest.getItemId());
8         PaymentInfo paymentInfo = new PaymentInfo(item.getPrice());
9         paymentInfoDao.insert(paymentInfo);
```

8	PurchaseOrder order = new PurchaseOrder(item.getId(), orderRequest.getAddress(), paymentInfo.getId());
9	purchaseOrderDao.insert(order);
10	return new PurchaseOrderResult(item, paymentInfo, order);
11	}
12	}

☐ 위의 코드는 아래와 같은 순서로 상품 구매를 처리할 것이다.

㉠ 첫째. itemDao로부터 구매하려는 상품정보를 구한다. 상품정보가 존재하지 않을 경우 예외를 발생시킨다.

㉡ 둘째. paymentInfoDao를 이용해서 결제정보(PaymentInfo)를 데이터베이스에 삽입한다.

㉢ 셋째. purchaseOrderDao를 이용해서 구매 주문 정보(PurchaseOrder)를 데이터베이스에 삽입한다.

☐ 이와 같은 경우에 트랜잭션에서 처리되어야 할 것이다. 만약 세 번째 과정이 실패하면 이전에 실행된 모든 과정이 롤백이 되어야 할 것이다.

☐ 스프링에서 트랜잭션을 처리하기 위하여 TransactionTemplate를 이용하면 된다.

① TransactionTemplate를 사용하려면 먼저 TransactionTemplate를 빈으로 설정한다.

```
<bean id="transactionTemplate" class="org.springframework.transaction.support.TransactionTemplate"
      p:transactionManager-ref="transactionManager" />
<bean id="placeOrderService" class="tommy.spring.store.PlaceOrderServiceTransactionTemplateImpl"
      p:transactionTemplate-ref="transactionTemplate" .../>
```

② TransactionTemplate 빈을 설정했다면 TransactionTemplate 클래스를 이용해서 트랜잭션을 처리할 수 있다.

1	import org.springframework.transaction.TransactionStatus;
2	import org.springframework.transaction.support.TransactionCallback;
3	import org.springframework.transaction.support.TransactionTemplate;
4	public class PlaceOrderServiceTxTemplateImpl implements PlaceOrderService{
5	private ItemDao itemDao;
6	private PaymentInfoDao paymentInfoDao;
7	private PurchaseOrderDao purchaseOrderDao;
8	private TransactionTemplate transactionTemplate;
	...
9	public void setTransactionTemplate(TransactionTemplate transactionTemplate){
10	this.transactionTemplate = transactionTemplate;
11	}
12	@Override
13	public PurchaseOrderResult order(final PurchaseOrderResult orderRequest)
	throws ItemNotFoundException{
14	return transactionTemplate.execute(new TransactionCallback<PurchaseOrderResult>(){
15	@Override
16	public PurchaseOrderResult doInTransaction(TransactionStatus status){
17	Item item = itemDao.findById(orderRequest.getItemId());

```

18         if(item == null)
19             throw new ItemNotFoundException(orderRequest.getItemId());
20         PaymentInfo paymentInfo = new PaymentInfo(item.getPrice());
21         paymentInfoDao.insert(paymentInfo);
22         PurchaseOrder order = new PurchaseOrder(
23             item.getId(), orderRequest.getAddress(), paymentInfo.getId());
24         purchaseOrderDao.insert(order);
25         return new PurchaseOrderResult(item, paymentInfo, order);
26     }
27 }

```

- ☐ TransactionTemplate.execute() 메서드는 트랜잭션을 시작한 뒤 파라미터로 전달받은 TransactionCallback 구현 객체의 doInTransaction() 메서드를 호출한다.
- ☐ doInTransaction() 메서드의 실행이 종료되면 트랜잭션을 commit 한다. 만약 이때 예외나 에러가 발생한다면 트랜잭션을 rollback 한다.
- ☐ 일반적으로 TransactionTemplate 클래스를 사용할 때는 TransactionCallback 인터페이스의 구현 클래스를 작성하기 보다는 위의 예처럼 익명이너클래스를 통하여 TransactionCallback 객체를 TransactionTemplate.execute() 메서드에 전달한다.

5. 선언적 트랜잭션 처리 : 두 가지 방식으로 정의할 수 있다.

- ① <tx:advice> 태그를 이용한 트랜잭션 처리
- ② @Transactional 어노테이션을 이용한 트랜잭션 설정

6. <tx:advice> 태그를 이용한 선언적 트랜잭션 처리

- ① <tx:advice> 태그를 이용한 트랜잭션 처리
- ☐ <tx:advice> 태그는 스프링2.0부터 지원하는 확장태그이다. tx 네임스페이스를 추가하면 된다.

```

1 <beans xmlns="http://www.springframework.org/schema/beans"
2       xmlns:p="http://www.springframework.org/schema/p"
3       xmlns:aop="http://www.springframework.org/schema/aop"
4       xmlns:tx="http://www.springframework.org/schema/tx"
5       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
6       xsi:schemaLocation="http://www.springframework.org/schema/beans
7       http://www.springframework.org/schema/beans/spring-beans-4.3.xsd
8       http://www.springframework.org/schema/aop
9       http://www.springframework.org/schema/aop/spring-aop-4.3.xsd
10      http://www.springframework.org/schema/tx
11      http://www.springframework.org/schema/tx/spring-tx-4.3.xsd">
12
13     <bean id="transactionManager"
14           class="org.springframework.jdbc.datasource.DataSourceTransactionManager"
15           p:dataSource-ref="dataSource" />
16
17     <tx:advice id="txAdvice" transaction-manager="transactionManager">

```

16	<tx:attributes>
17	<tx:method name="order" />
18	<tx:method name="get*" read-only="true" />
19	</tx:attributes>
20	</tx:advice>
21	...
21	</beans>

□ <tx:advice> 태그의 속성

속성 이름	설명
name	트랜잭션이 적용될 메서드 이름을 명시한다. "*"를 사용한 설정이 가능하다. 예를 들어 "get*"로 설정할 경우 이름이 get으로 시작하는 메서드를 의미한다.
propagation	트랜잭션 전파 규칙을 설정한다.
isolation	트랜잭션 격리레벨을 설정한다.
read-only	읽기 전용 여부를 설정한다.
no-rollback-for	트랜잭션을 롤백하지 않을 예외 타입을 설정한다.
rollback-for	트랜잭션을 롤백할 예외 타입을 설정한다.
timeout	트랜잭션의 타임 아웃 시간을 초 단위로 설정한다.

- <tx:advice> 태그는 Advisor만 생성하는 것이지 실제로 트랜잭션을 적용하는 것은 아니다. 실제 트랜잭션의 적용은 AOP를 통해서 이루어진다.

1	<aop:config>
2	<aop:pointcut expression="execution(public * tommy.spring..*Service.*(..))" id="servicePublicMethod" />
3	<aop:advisor advice-ref="txAdvice" pointcut-ref="servicePublicMethod" />
4	</aop:config>

② <tx:method> 태그의 propagation 속성과 전파규칙

속성 값	설명
REQUIRED(기본값)	메서드를 수행하는데 트랜잭션이 필요하다는 것을 의미한다. 현재 진행 중인 트랜잭션이 존재하면 해당 트랜잭션을 사용한다. 존재하지 않으면 새로운 트랜잭션을 생성한다.
MANDATORY	메서드를 수행하는데 트랜잭션이 필요하다는 것을 의미한다. REQUIRED와는 달리 진행 중인 트랜잭션이 존재하지 않을 경우 예외를 발생시킨다.
REQUIRES_NEW	항상 새로운 트랜잭션을 시작한다. 기존 트랜잭션이 존재하면 기존 트랜잭션을 일시중지하고 새로운 트랜잭션을 시작한다. 새로 시작된 트랜잭션이 종료된 뒤 기존 트랜잭션이 계속된다.
SUPPORTS	메서드가 트랜잭션을 필요로 하지는 않지만 기존 트랜잭션이 존재할 경우 트랜잭션을 사용한다는 것을 의미한다. 진행 중인 트랜잭션이 존재하지 않더라도 메서드는 정상적으로 동작한다.
NOT_SUPPORTED	메서드가 트랜잭션을 필요로 하지 않음을 의미. SUPPORTS와는 달리 진행 중인 트랜잭션이 존재할 경우 메서드가 실행되는 동안 트랜잭션은 일시 중지되면 메서드 종료 후 트랜

	작업을 계속진행.
NEVER	메서드가 트랜잭션을 필요로 하지 않으며 만약 진행 중인 트랜잭션이 존재하면 예외를 발생시킨다.
NESTED	기존 트랜잭션이 존재하면 기존 트랜잭션에서 중첩된 트랜잭션에서 메서드를 실행한다. 기존 트랜잭션이 존재하지 않으면 REQUIRED와 동일하게 동작한다. (JDBC 3.0이상)

③ <tx:method> 태그의 rollback-for 속성과 no-rollback-for 속성을 통한 롤백 처리

- ☐ 스프링 트랜잭션은 기본적으로 RuntimeException 및 Error에 대해서만 롤백 처리를 수행한다. 따라서 Throwable이나 Exception 타입의 예외가 발생하더라도 롤백이 되지 않고 예외가 발생하기 전까지의 작업이 커밋된다.
- ☐ rollback-for 속성은 예외 발생 시 롤백 작업을 수행할 예외타입을 설정한다.
- ☐ no-rollback-for 속성은 예외가 발생하더라도 롤백하지 않을 예외타입을 설정한다.
- ☐ 명시할 예외타입이 한 개 이상인 경우 각각의 예외는 콤마로 구분한다.

```
<tx:method name="regist" rollback-for="Exception" no-rollback-for="MemberNotFoundException" />
```

④ <tx:method> 태그의 isolation 속성과 트랜잭션 격리 레벨

속성 값	설명
DEFAULT	기본 설정을 사용한다.
READ_UNCOMMITTED	다른 트랜잭션에서 커밋하지 않은 데이터를 읽을 수 있다.
READ_COMMITTED	다른 트랜잭션에 의해 커밋된 데이터를 읽을 수 있다.
REPEATABLE_READ	처음에 읽어 온 데이터와 두 번째 읽어 온 데이터가 동일한 값을 갖는다.
SERIALIZABLE	동일한 데이터에 대해서 동시에 두 개 이상의 트랜잭션이 수행 될 수 없다.

7. 어노테이션 기반의 트랜잭션 설정

- ① @Transactional 어노테이션을 이용해서 트랜잭션을 설정할 수 있다.

```

1 import org.springframework.transaction.annotation.Transactional;
2 ...
3 public class PlaceOrderServiceImpl implements PlaceOrderService{
4     ...
5     @Override
6     @Transactional(propagation=Propagation.REQUIRED)
7     public PurchaseOrderResult order(PurchaseOrderRequest orderRequest)
8         throws ItemNotFoundException{
9         ...
10    }
11 }

```

② @Transactional 어노테이션의 주요 속성

속성	설명
propagation	트랜잭션 전파 규칙을 설정한다. Propagation 열거형 타입에 값이 정의되어 있다. 기본 값은 Propagation.REQUIRED 이다.
isolation	트랜잭션 격리 레벨을 설정한다. Isolation 열거형 타입에 값이 정의되어 있다.
readOnly	읽기 전용 여부를 설정한다. boolean 값을 설정하면 기본 값은 false 이다.
rollbackFor	트랜잭션을 롤백할 예외 타입을 설정한다. 예. rollbackFor=(Exception.class)
noRollbackFor	트랜잭션을 롤백하지 않을 예외 타입을 설정한다. 예. noRollbackFor=(ItemNotFoundException.class)
timeout	트랜잭션의 타임아웃 시간을 초단위로 설정한다.

③ @Transactional 어노테이션이 적용된 스프링 빈에 실제로 트랜잭션을 적용하려면 다음과 같이 <tx:annotation-driven> 태그를 설정하면 된다.

```
<tx:annotation-driven transaction-manager="transactionManager" />
<bean id="placeOrderService" class="tommy.spring.store.PlaceOrderServiceImpl" .../>
```

④ <tx:annotation-driven> 태그의 속성

속성	설명	기본 값
transaction-manager	PlatformTransactionManager 빈의 이름	transactionManager
proxy-target-class	클래스에 대해서 프록시를 생성할 지의 여부 true 일 경우 CGLIB를 이용해서 프록시를 생성 false일 경우 자바 다이내믹 프록시를 이용	false
order	Advice 적용순서	int의 최대 값 (가장 낮은 순위)

⑤ <tx:annotation-driven> 태그 대신 PersistenceAnnotationBeanPostProcessor 클래스를 빈으로 등록해도 된다.

```
<bean class="org.springframework.orm.jpa.support.PersistenceAnnotationBeanPostProcessor">
```

8. 실습을 위한 기본 준비

① 아이템 정보를 기억할 테이블 작성 : ITEM

```
CREATE table "ITEM" (
    "ITEM_ID"          NUMBER(5,0) NOT NULL,
    "ITEM_NAME"        VARCHAR2(20),
    "PRICE"            NUMBER(5,0) NOT NULL,
    constraint "ITEM_PK" primary key ("ITEM_ID")
);
```


② ITEM 테이블에서 사용할 시퀀스 생성 : ITEM_SEQ

```
create sequence "ITEM_SEQ" start with 1 increment by 1 maxvalue 99999
nocache nocycle noorder;
```

③ ITEM에 기본적인 상품정보를 상품정보를 삽입하자 (3개정도)

```
insert into item values (item_seq.nextval, '노트북', 50000);
insert into item values (item_seq.nextval, '모니터', 30000);
insert into item values (item_seq.nextval, '외장하드', 10000);
commit;
```

④ 결제정보를 입력할 테이블을 작성하자 : PAYMENT_INFO

```
CREATE table "PAYMENT_INFO" (
    "PAYMENT_INFO_ID"      NUMBER(5,0) NOT NULL,
    "PRICE"                NUMBER(5,0) NOT NULL,
    constraint "PAYMENT_INFO_PK" primary key ("PAYMENT_INFO_ID")
);
```

⑤ PAYMENT_INFO에서 사용할 시퀀스를 생성하자 : PAYMENT_SEQ

```
create sequence "PAYMENT_SEQ" start with 1 increment by 1 maxvalue 99999
nocache nocycle noorder;
```

⑥ 주문정보를 기억할 테이블을 작성하자 : PURCHASE_ORDER

```
CREATE table "PURCHASE_ORDER" (
    "PURCHASE_ORDER_ID"    NUMBER(5,0) NOT NULL,
    "ITEM_ID"              NUMBER(5,0) NOT NULL,
    "PAYMENT_INFO_ID"      NUMBER(5,0) NOT NULL,
    "ADDRESS"              VARCHAR2(20) NOT NULL,
    constraint "PURCHASE_ORDER_PK" primary key ("PURCHASE_ORDER_ID")
);
```

⑦ PURCHASE_ORDER 테이블에서 사용할 시퀀스를 만들자 : PURCHASE_SEQ

```
create sequence "PURCHASE_SEQ" start with 1 increment by 1
maxvalue 99999 nocache nocycle noorder;
```

⑧ SpringTX라는 메이븐 프로젝트를 생성한다.

■ src/main/java 폴더안의 모든 패키지 및 파일을 삭제하고, JDK 버전을 1.8로 변경한다.

- pom.xml 파일에 메이븐 설정정보를 등록한다.
- 스프링 의존관계를 추가한다.
 - 기본 스프링 관련 : spring-context, common-logging
 - 데이터베이스 관련 : spring-jdbc, common-dbcp2, ojdbc8
 - AOP 관련 : aspectjrt, aspectjweaver
- 프로젝트 우 클릭 후 [Maven]-[Update Project...]를 수행한다.

⑨ 기본적인 모델 클래스를 작성한다.

- ☐ 아이템 정보를 기억할 Item 클래스

```

1 package tommy.spring.store.vo;
2 public class Item {
3     private Integer id;
4     private String name;
5     private int price;
6     // getter, setter 추가
7 }
```

- ☐ 결제 정보를 기억할 PaymentInfo 클래스

```

1 package tommy.spring.store.vo;
2 public class PaymentInfo {
3     private Integer id;
4     private int price;
5     public PaymentInfo() {
6     }
7     public PaymentInfo(int price) {
8         this.price = price;
9     }
10    // getter, setter 추가
11 }
```

- ☐ 구매 주문 정보를 기억할 PurchaseOrder 클래스

```

1 package tommy.spring.store.vo;
2 public class PurchaseOrder {
3     private Integer id;
4     private Integer itemId;
5     private String address;
6     private Integer paymentInfoId;
7     public PurchaseOrder() {
8     }
9     public PurchaseOrder(Integer itemId, String address, Integer paymentInfoId) {
10         this.itemId = itemId;
11         this.address = address;
12         this.paymentInfoId = paymentInfoId;
13     }
```

14	// getter, setter 추가
15	}

☐ 아이템을 찾지 못할 경우의 사용자 정의 예외 : `ItemNotFoundException` 클래스

```
1 package tommy.spring.store.vo;
2 public class ItemNotFoundException extends RuntimeException {
3     private static final long serialVersionUID = 1L;
4     private Integer itemId;
5     public ItemNotFoundException(Integer itemId) {
6         super("not found item : id = " + itemId);
7         this.itemId = itemId;
8     }
9     public Integer getItemId() {
10         return itemId;
11     }
12 }
```

☐ 구매 주문 요구사항을 나타내는 PurchaseOrderRequest 클래스

```
1 package tommy.spring.store.vo;
2 public class PurchaseOrderRequest {
3     private Integer itemId;
4     private String address;
5     // getter, setter 추가
6 }
```

□ 구매 주문 결과를 저장할 객체 PurchaseOrderResult 클래스

```

1 package tommy.spring.store.vo;
2 public class PurchaseOrderResult {
3     private Item item;
4     private PaymentInfo paymentInfo;
5     private PurchaseOrder order;
6     public PurchaseOrderResult(Item item, PaymentInfo paymentInfo,
                                PurchaseOrder order) {
7         this.item = item;
8         this.paymentInfo = paymentInfo;
9         this.order = order;
10    }
11    // getter, setter 추가
12 }

```

⑩ 각각의 모델에 대한 DAO 인터페이스 작성

□ 아이템을 검색해 줄 ItemDao 인터페이스

1	package tommy.spring.store.dao;
2	import tommy.spring.store.vo.Item;
3	public interface ItemDao {
4	Item findById(Integer itemId);
5	}

□ 결제정보를 저장해줄 PaymentInfoDao 인터페이스

1	package tommy.spring.store.dao;
2	import tommy.spring.store.vo.PaymentInfo;
3	public interface PaymentInfoDao {
4	void insert(PaymentInfo paymentInfo);
5	}

□ 구매 주문 결과를 저장해 줄 PurchaseOrderDao 인터페이스

1	package tommy.spring.store.dao;
2	import tommy.spring.store.vo.PurchaseOrder;
3	public interface PurchaseOrderDao {
4	void insert(PurchaseOrder order);
5	}

⑪ DAO 인터페이스를 상속받아 실제 데이터베이스 작업을 처리해 줄 클래스를 작성하자.

□ ItemDao 인터페이스를 구현한 JdbcItemDao 클래스

1	package tommy.spring.store.dao.jdbc;
2	import java.sql.ResultSet;
3	import java.sql.SQLException;
4	import org.springframework.jdbc.core.JdbcTemplate;
5	import org.springframework.jdbc.core.RowMapper;
6	import tommy.spring.store.dao.ItemDao;
7	import tommy.spring.store.vo.Item;
8	public class JdbcItemDao implements ItemDao {
9	private JdbcTemplate jdbcTemplate;
10	public JdbcItemDao(JdbcTemplate jdbcTemplate) {
11	this.jdbcTemplate = jdbcTemplate;
12	}
13	@Override
14	public Item findById(Integer itemId) {
15	return jdbcTemplate.queryForObject("select * from ITEM where ITEM_ID = ?",
	new Object[] { itemId }, new RowMapper<Item>() {
16	@Override
17	public Item mapRow(ResultSet rs, int row) throws SQLException {
18	Item item = new Item();
19	item.setId(rs.getInt("ITEM_ID"));
20	item.setPrice(rs.getInt("PRICE"));
21	return item;

22	}
23	});
24	}
25	}

□ PaymentInfoDao 인터페이스를 구현한 JdbcPaymentInfoDao 클래스

1	package tommy.spring.store.dao.jdbc;
2	import java.util.Collections;
3	import java.util.HashMap;
4	import java.util.Map;
5	import org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate;
6	import org.springframework.jdbc.core.simple.SimpleJdbcInsert;
7	import tommy.spring.store.dao.PaymentInfoDao;
8	import tommy.spring.store.vo.PaymentInfo;
9	public class JdbcPaymentInfoDao implements PaymentInfoDao {
10	private SimpleJdbcInsert insert;
11	private NamedParameterJdbcTemplate namedJdbcTemplate;
12	public JdbcPaymentInfoDao(SimpleJdbcInsert insert) {
13	this.insert = insert;
14	insert.withTableName("PAYMENT_INFO").usingColumns(
	"PAYMENT_INFO_ID", "PRICE");
15	}
16	public void setNamedJdbcTemplate(
	NamedParameterJdbcTemplate namedJdbcTemplate) {
17	this.namedJdbcTemplate = namedJdbcTemplate;
18	}
19	public int nextVal() {
20	return namedJdbcTemplate.queryForObject(
	"select payment_seq.nextval from dual",
	Collections.emptyMap(), Integer.class);
21	}
22	@Override
23	public void insert(PaymentInfo paymentInfo) {
24	Map<String, Object> paramValueMap = new HashMap<String, Object>();
25	paymentInfo.setId(nextVal());
26	paramValueMap.put("PAYMENT_INFO_ID", paymentInfo.getId());
27	paramValueMap.put("PRICE", paymentInfo.getPrice());
28	insert.execute(paramValueMap);
29	}
30	}

□ PurchaseOrderDao 인터페이스를 구현한 JdbcPurchaseOrderDao 클래스

1	package tommy.spring.store.dao.jdbc;
2	import java.util.HashMap;
3	import java.util.Map;
4	import org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate;

```

5 import org.springframework.jdbc.core.simple.SimpleJdbcInsert;
6 import tommy.spring.store.dao.PurchaseOrderDao;
7 import tommy.spring.store.vo.PurchaseOrder;
8 public class JdbcPurchaseOrderDao implements PurchaseOrderDao {
9     private SimpleJdbcInsert insert;
10    private NamedParameterJdbcTemplate namedJdbcTemplate;
11    public JdbcPurchaseOrderDao(SimpleJdbcInsert insert) {
12        this.insert = insert;
13        insert.withTableName("PURCHASE_ORDER").usingColumns(
14            "PURCHASE_ORDER_ID", "ITEM_ID", "PAYMENT_INFO_ID", "ADDRESS");
15    }
16    public void setNamedJdbcTemplate(
17        NamedParameterJdbcTemplate namedJdbcTemplate) {
18        this.namedJdbcTemplate = namedJdbcTemplate;
19    }
20    public int nextVal() {
21        Map<String, Object> emptyMap = new HashMap<String, Object>();
22        return namedJdbcTemplate.queryForObject(
23            "select purchase_seq.nextval from dual", emptyMap, Integer.class);
24    }
25    @Override
26    public void insert(PurchaseOrder order) {
27        Map<String, Object> args = new HashMap<String, Object>();
28        order.setId(nextVal());
29        args.put("PURCHASE_ORDER_ID", order.getId());
30        args.put("ITEM_ID", order.getItemId());
31        args.put("PAYMENT_INFO_ID", order.getPaymentInfoId());
32        args.put("ADDRESS", order.getAddress());
33        insert.execute(args);
34    }
35 }

```

□ 이제 모든 준비가 끝났다. 상황별로 트랜잭션 처리를 구현해 보자.

9. 실습예제 1 :

① 구매서비스를 추상화한 PlaceOrderService 인터페이스를 작성하자.

```

1 package tommy.spring.store.service;
2 import tommy.spring.store.vo.ItemNotFoundException;
3 import tommy.spring.store.vo.PurchaseOrderRequest;
4 import tommy.spring.store.vo.PurchaseOrderResult;
5 public interface PlaceOrderService {
6     public PurchaseOrderResult order(PurchaseOrderRequest orderRequest)
7                                     throws ItemNotFoundException;
8 }

```

② 구매서비스를 실제로 구현한 PlaceOrderServiceImplOne 클래스를 구현하자.

1	package tommy.spring.store.service;
2	import tommy.spring.store.dao.ItemDao;
3	import tommy.spring.store.dao.PaymentInfoDao;
4	import tommy.spring.store.dao.PurchaseOrderDao;
5	import tommy.spring.store.vo.Item;
6	import tommy.spring.store.vo.ItemNotFoundException;
7	import tommy.spring.store.vo.PaymentInfo;
8	import tommy.spring.store.vo.PurchaseOrder;
9	import tommy.spring.store.vo.PurchaseOrderRequest;
10	import tommy.spring.store.vo.PurchaseOrderResult;
11	public class PlaceOrderServiceImplOne implements PlaceOrderService {
12	private ItemDao itemDao;
13	private PaymentInfoDao paymentInfoDao;
14	private PurchaseOrderDao purchaseOrderDao;
15	public void setItemDao(ItemDao itemDao) {
16	this.itemDao = itemDao;
17	}
18	public void setPaymentInfoDao(PaymentInfoDao paymentInformationDao) {
19	this.paymentInfoDao = paymentInformationDao;
20	}
21	public void setPurchaseOrderDao(PurchaseOrderDao purchaseOrderDao) {
22	this.purchaseOrderDao = purchaseOrderDao;
23	}
24	@Override
25	public PurchaseOrderResult order(PurchaseOrderRequest orderRequest)
	throws ItemNotFoundException {
26	Item item = itemDao.findById(orderRequest.getItemId());
27	if (item == null)
28	throw new ItemNotFoundException(orderRequest.getItemId());
29	PaymentInfo paymentInfo = new PaymentInfo(item.getPrice());
30	paymentInfoDao.insert(paymentInfo);
31	PurchaseOrder order = new PurchaseOrder(
	item.getId(), orderRequest.getAddress(), paymentInfo.getId());
32	purchaseOrderDao.insert(order);
33	return new PurchaseOrderResult(item, paymentInfo, order);
34	}
35	}

③ 스프링 설정 파일을 작성하자.

- ☐ src/main/resources 폴더를 생성하고 applicationContext.xml 파일을 작성하자
- : bean, p, jdbc 체크

	<!-- 상단 부분 생략 -->
1	<bean id="dataSource" class="org.apache.commons.dbcp2.BasicDataSource"
	destroy-method="close" p:driverClassName="oracle.jdbc.driver.OracleDriver"
	p:url="jdbc:oracle:thin:@localhost:1521/XEPDB1" p:username="mytest"
2	p:password="mytest" />
	<bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate"

	p:dataSource-ref="dataSource" />
3	<bean id="namedParameterJdbcTemplate"
	class="org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate">
4	<constructor-arg><ref bean="dataSource" /></constructor-arg>
5	</bean>
6	<bean id="simpleJdbcInsert" class="org.springframework.jdbc.core.simple.SimpleJdbcInsert"
	scope="prototype">
7	<constructor-arg><ref bean="dataSource" /></constructor-arg>
8	</bean>
	<!-- 하단 부분 생략 -->

□ transactionOne.xml 작성 : bean, p, aop, jdbc, tx 체크

	<!-- 상단 부분 생략 -->
1	<import resource="applicationContext.xml"/>
2	<bean id="transactionManager"
	class="org.springframework.jdbc.datasource.DataSourceTransactionManager"
	p:dataSource-ref="dataSource" />
3	<bean id="itemDao" class="tommy.spring.store.dao.jdbc.JdbcItemDao">
4	<constructor-arg><ref bean="jdbcTemplate" /></constructor-arg>
5	</bean>
6	<bean id="paymentInfoDao" class="tommy.spring.store.dao.jdbc.JdbcPaymentInfoDao">
7	<constructor-arg><ref bean="simpleJdbcInsert" /></constructor-arg>
8	<property name="namedJdbcTemplate" ref="namedParameterJdbcTemplate" />
9	</bean>
10	<bean id="purchaseOrderDao" class="tommy.spring.store.dao.jdbc.JdbcPurchaseOrderDao">
11	<constructor-arg><ref bean="simpleJdbcInsert" /></constructor-arg>
12	<property name="namedJdbcTemplate" ref="namedParameterJdbcTemplate" />
13	</bean>
14	<tx:advice id="txAdvice" transaction-manager="transactionManager">
15	<tx:attributes>
16	<tx:method name="order" propagation="REQUIRED" />
17	<tx:method name="get*" read-only="true" />
18	</tx:attributes>
19	</tx:advice>
20	<aop:config>
21	<aop:pointcut expression="execution(public * tommy.spring.*Service.*(..))"
	id="servicePublicMethod" />
22	<aop:advisor advice-ref="txAdvice" pointcut-ref="servicePublicMethod" />
23	</aop:config>
24	<bean id="placeOrderService" class="tommy.spring.store.service.PlaceOrderServiceImplOne"
	p:itemDao-ref="itemDao" p:paymentInfoDao-ref="paymentInfoDao"
	p:purchaseOrderDao-ref="purchaseOrderDao" />
	<!-- 하단 부분 생략 -->

④ 실습용 테스트 클래스를 작성하자 : OrderServiceTestOne 클래스


```

1 package tommy.spring.store.controller;
2 import org.springframework.context.support.AbstractApplicationContext;
3 import org.springframework.context.support.ClassPathXmlApplicationContext;
4 import tommy.spring.store.service.PlaceOrderService;
5 import tommy.spring.store.vo.PurchaseOrderRequest;
6 import tommy.spring.store.vo.PurchaseOrderResult;
7 public class OrderServiceTestOne {
8     private PlaceOrderService placeOrderService;
9     private AbstractApplicationContext context;
10    public OrderServiceTestOne() {
11        String[] configLocations = new String[] { "transactionOne.xml" };
12        context = new ClassPathXmlApplicationContext(configLocations);
13        placeOrderService = (PlaceOrderService)
14                                context.getBean("placeOrderService");
15    }
16    public void order() {
17        PurchaseOrderRequest orderRequest = new PurchaseOrderRequest();
18        orderRequest.setItemId(1);
19        orderRequest.setAddress("서울 종로구");
20        PurchaseOrderResult orderResult = placeOrderService.order(orderRequest);
21        System.out.println("주문상태 정보");
22        System.out.println("아이템 : " + orderResult.getItem().getId());
23        System.out.println("가격 : " + orderResult.getPaymentInfo().getPrice());
24    }
25    public void close() {
26        context.close();
27    }
28    public static void main(String[] args) {
29        OrderServiceTestOne test = new OrderServiceTestOne();
30        test.order();
31        test.close();
32    }

```

- ⑤ OrderServiceTestOne 클래스를 실행하여 구매서비스가 정상적으로 이루어지는지 결과를 확인하자.

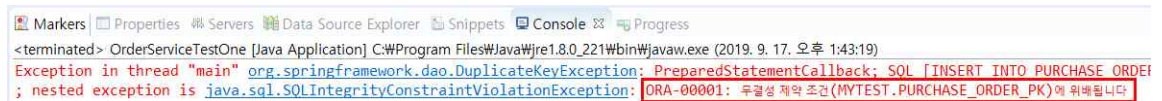
The first screenshot shows the 'Markers' panel with a 'terminated' event for 'OrderServiceTestOne' and a red box highlighting the output: '주문상태 정보', '아이템 : 1', '가격 : 50000'.

The second screenshot shows the 'SQL' panel with a red box highlighting the output: '1', '1', '50000'.

The third screenshot shows the 'SQL' panel with a red box highlighting the output: '1', '1', '1', '서울 종로구'.

- ⑥ 이제 purchase_order 테이블에 아래와 같이 데이터를 하나 임의로 넣어 넣고 롤백이 되는지 테스트 해 보자. 데이터가 삽입 되었다면 OrderServiceTestOne 클래스를 실행하고 payment_info 테이블을 확인해 보자.

```
insert into purchase_order values(2, 1, 100, '서울 중구');
commit;
```



```
<terminated> OrderServiceTestOne [Java Application] C:\Program Files\Java\jre1.8.0_221\bin\javaw.exe (2019. 9. 17. 오후 1:43:19)
Exception in thread "main" org.springframework.dao.DuplicateKeyException: PreparedStatementCallback; SQL [INSERT INTO PURCHASE_ORDER]; nested exception is java.sql.SQLIntegrityConstraintViolationException: ORA-00001: unique constraint (MYTEST.PURCHASE_ORDER_PK) violated
```

- payment_info 테이블에 들어갔던 결제정보가 취소되는 것을 확인할 수 있다.

10. 실습예제 2 :

- ① 구매서비스를 실제로 구현한 PlaceOrderServiceImplTwo 클래스를 구현하자.

```
1 package tommy.spring.store.service;
2 import org.springframework.transaction.TransactionStatus;
3 import org.springframework.transaction.support.TransactionCallback;
4 import org.springframework.transaction.support.TransactionTemplate;
5 import tommy.spring.store.dao.ItemDao;
6 import tommy.spring.store.dao.PaymentInfoDao;
7 import tommy.spring.store.dao.PurchaseOrderDao;
8 import tommy.spring.store.vo.Item;
9 import tommy.spring.store.vo.ItemNotFoundException;
10 import tommy.spring.store.vo.PaymentInfo;
11 import tommy.spring.store.vo.PurchaseOrder;
12 import tommy.spring.store.vo.PurchaseOrderRequest;
13 import tommy.spring.store.vo.PurchaseOrderResult;
14 public class PlaceOrderServiceImplTwo implements PlaceOrderService {
15     private ItemDao itemDao;
16     private PaymentInfoDao paymentInfoDao;
17     private PurchaseOrderDao purchaseOrderDao;
18     private TransactionTemplate transactionTemplate;
19     public void setItemDao(ItemDao itemDao) {
20         this.itemDao = itemDao;
21     }
22     public void setPaymentInfoDao(PaymentInfoDao paymentInformationDao) {
23         this.paymentInfoDao = paymentInformationDao;
24     }
25     public void setPurchaseOrderDao(PurchaseOrderDao purchaseOrderDao) {
26         this.purchaseOrderDao = purchaseOrderDao;
27     }
28     public void setTransactionTemplate(TransactionTemplate transactionTemplate) {
29         this.transactionTemplate = transactionTemplate;
30     }
```

```

31      @Override
32      public PurchaseOrderResult order(final PurchaseOrderRequest orderRequest)
                                   throws ItemNotFoundException {
33          return transactionTemplate.execute(
                                   new TransactionCallback<PurchaseOrderResult>() {
34              @Override
35              public PurchaseOrderResult doInTransaction(TransactionStatus status) {
36                  Item item = itemDao.findById(orderRequest.getItemId());
37                  if (item == null)
38                      throw new ItemNotFoundException(orderRequest.getItemId());
39                  PaymentInfo paymentInfo = new PaymentInfo(item.getPrice());
40                  paymentInfoDao.insert(paymentInfo);
41                  PurchaseOrder order = new PurchaseOrder(item.getId(),
                                   orderRequest.getAddress(), paymentInfo.getId());
42                  purchaseOrderDao.insert(order);
43                  return new PurchaseOrderResult(item, paymentInfo, order);
44              }
45          });
46      }
47  }

```

② 스프링 설정파일을 작성하자. : transactionTwo.xml

```

1  <bean id="transactionManager"
        class="org.springframework.jdbc.datasource.DataSourceTransactionManager"
        p:dataSource-ref="dataSource" />
2  <bean id="itemDao" class="tommy.spring.store.dao.jdbc.JdbcItemDao">
3      <constructor-arg><ref bean="jdbcTemplate" /></constructor-arg>
4  </bean>
5  <bean id="paymentInfoDao" class="tommy.spring.store.dao.jdbc.JdbcPaymentInfoDao">
6      <constructor-arg><ref bean="simpleJdbcInsert" /></constructor-arg>
7      <property name="namedJdbcTemplate" ref="namedParameterJdbcTemplate" />
8  </bean>
9  <bean id="purchaseOrderDao" class="tommy.spring.store.dao.jdbc.JdbcPurchaseOrderDao">
10     <constructor-arg><ref bean="simpleJdbcInsert" /></constructor-arg>
11     <property name="namedJdbcTemplate" ref="namedParameterJdbcTemplate" />
12 </bean>
13 <!-- TransactionTemplate을 이용한 트랜잭션 구현 -->
14 <bean id="transactionTemplate"
        class="org.springframework.transaction.support.TransactionTemplate"
        p:transactionManager-ref="transactionManager" />
15 <bean id="placeOrderService" class="tommy.spring.store.service.PlaceOrderServiceImplTwo"
        p:itemDao-ref="itemDao" p:paymentInfoDao-ref="paymentInfoDao"
        p:purchaseOrderDao-ref="purchaseOrderDao"
        p:transactionTemplate-ref="transactionTemplate" />

```

③ 테스트용 클래스 작성 : OrderServiceTestTwo 클래스

```

1 package tommy.spring.store.controller;
2 import org.springframework.context.ApplicationContext;
3 import org.springframework.context.support.ClassPathXmlApplicationContext;
4 import tommy.spring.store.service.PlaceOrderService;
5 import tommy.spring.store.vo.PurchaseOrderRequest;
6 import tommy.spring.store.vo.PurchaseOrderResult;
7 public class OrderServiceTestTwo {
8     private PlaceOrderService placeOrderService;
9     private AbstractApplicationContext context;
10    public OrderServiceTestTwo() {
11        String[] configLocations = new String[] {
12            "applicationcontext.xml","transactionTwo.xml" };
13        context = new ClassPathXmlApplicationContext(configLocations);
14        placeOrderService = (PlaceOrderService)
15            context.getBean("placeOrderService");
16    }
17    public void order() {
18        PurchaseOrderRequest orderRequest = new PurchaseOrderRequest();
19        orderRequest.setItemId(2);
20        orderRequest.setAddress("서울 강남구");
21        PurchaseOrderResult orderResult = placeOrderService.order(orderRequest);
22        System.out.println("주문상태 정보");
23        System.out.println("아이템 : " + orderResult.getItem().getId());
24        System.out.println("가격 : " + orderResult.getPaymentInfo().getPrice());
25    }
26    public void close() {
27        context.close();
28    }
29    public static void main(String[] args) {
30        OrderServiceTestTwo test = new OrderServiceTestTwo();
31        test.order();
32        test.close();
33    }
34 }

```

④ OrderServiceTestTwo 클래스를 실행하여 결과를 확인해 보자.

The first screenshot shows the 'Markers' window with a message '주문상태 정보' and '아이템 : 2' and '가격 : 30000'.

The second screenshot shows the 'SQL' window with a table of payment information:

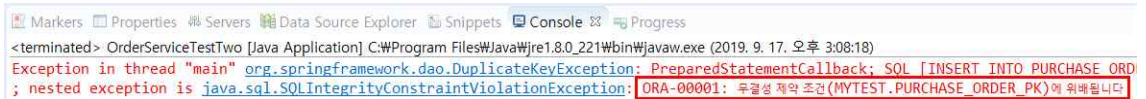
	PAYMENT_INFO_ID	PRICE
1	3	30000
2	1	150000

The third screenshot shows the 'SQL' window with a table of purchase order information:

	PURCHASE_ORDER_ID	ITEM_ID	PAYMENT_INFO_ID	ADDRESS
1	3	2	3	서울 강남구
2	1	1	1	서울 종로구
3	2	1	100	서울 송구

- ⑤ 앞의 예제처럼 이제 purchase_order 테이블에 아래와 같이 데이터를 하나 임의로 넣어 넣고 롤백이 되는지 테스트 해 보자. 데이터가 삽입 되었다면 OrderServiceTestTwo 클래스를 실행하고 payment_info 테이블을 확인해 보자.

```
insert into purchase_order values(4, 1, 100, '서울 중구');
commit;
```



```
<terminated> OrderServiceTestTwo [Java Application] C:\Program Files\Java\jre1.8.0_221\bin\javaw.exe (2019. 9. 17. 오후 3:08:18)
Exception in thread "main" org.springframework.dao.DuplicateKeyException: PreparedStatementCallback; SQL [INSERT INTO PURCHASE_ORDER]; nested exception is java.sql.SQLIntegrityConstraintViolationException: ORA-00001: 두결성 제약 조건 (MYTEST.PURCHASE_ORDER_PK)에 위배됩니다
```

- payment_info 테이블에 들어갔던 결제정보가 취소되는 것을 확인할 수 있다.

11. 실습예제 3 :

- ① 구매서비스를 구현한 클래스를 작성하자 : PlaceOrderServiceImplThree

```
1 package tommy.spring.store.service;
2 import org.springframework.transaction.annotation.Transactional;
3 import tommy.spring.store.dao.ItemDao;
4 import tommy.spring.store.dao.PaymentInfoDao;
5 import tommy.spring.store.dao.PurchaseOrderDao;
6 import tommy.spring.store.vo.Item;
7 import tommy.spring.store.vo.ItemNotFoundException;
8 import tommy.spring.store.vo.PaymentInfo;
9 import tommy.spring.store.vo.PurchaseOrder;
10 import tommy.spring.store.vo.PurchaseOrderRequest;
11 import tommy.spring.store.vo.PurchaseOrderResult;
12 public class PlaceOrderServiceImplThree implements PlaceOrderService {
13     private ItemDao itemDao;
14     private PaymentInfoDao paymentInfoDao;
15     private PurchaseOrderDao purchaseOrderDao;
16     public void setItemDao(ItemDao itemDao) {
17         this.itemDao = itemDao;
18     }
19     public void setPaymentInfoDao(PaymentInfoDao paymentInformationDao) {
20         this.paymentInfoDao = paymentInformationDao;
21     }
22     public void setPurchaseOrderDao(PurchaseOrderDao purchaseOrderDao) {
23         this.purchaseOrderDao = purchaseOrderDao;
24     }
25     @Override
26     @Transactional
27     public PurchaseOrderResult order(PurchaseOrderRequest orderRequest)
28                                     throws ItemNotFoundException {
29         Item item = itemDao.findById(orderRequest.getItemId());
30         if (item == null)
```

30	throw new ItemNotFoundException(orderRequest.getItemId());
31	PaymentInfo paymentInfo = new PaymentInfo(item.getPrice());
32	paymentInfoDao.insert(paymentInfo);
33	PurchaseOrder order = new PurchaseOrder(item.getId(),
	orderRequest.getAddress(), paymentInfo.getId());
34	purchaseOrderDao.insert(order);
35	return new PurchaseOrderResult(item, paymentInfo, order);
36	}
37	}

② 스프링 설정파일을 작성하자 : transactionThree.xml

1	<bean id="transactionManager"
	class="org.springframework.jdbc.datasource.DataSourceTransactionManager"
	p:dataSource-ref="dataSource" />
2	<bean id="itemDao" class="tommy.spring.store.dao.jdbc.JdbcItemDao">
3	<constructor-arg><ref bean="jdbcTemplate" /></constructor-arg>
4	</bean>
5	<bean id="paymentInfoDao" class="tommy.spring.store.dao.jdbc.JdbcPaymentInfoDao">
6	<constructor-arg><ref bean="simpleJdbcInsert" /></constructor-arg>
7	<property name="namedJdbcTemplate" ref="namedParameterJdbcTemplate" />
8	</bean>
9	<bean id="purchaseOrderDao" class="tommy.spring.store.dao.jdbc.JdbcPurchaseOrderDao">
10	<constructor-arg><ref bean="simpleJdbcInsert" /></constructor-arg>
11	<property name="namedJdbcTemplate" ref="namedParameterJdbcTemplate" />
12	</bean>
13	<tx:annotation-driven transaction-manager="transactionManager" />
14	<bean id="placeOrderService" class="tommy.spring.store.service.PlaceOrderServiceImplThree"
	p:itemDao-ref="itemDao" p:paymentInfoDao-ref="paymentInfoDao"
	p:purchaseOrderDao-ref="purchaseOrderDao" />

③ 테스트용 메인클래스를 작성하자 : OrderServiceTestThree

1	package tommy.spring.store.controller;
2	import org.springframework.context.ApplicationContext;
3	import org.springframework.context.support.ClassPathXmlApplicationContext;
4	import tommy.spring.store.service.PlaceOrderService;
5	import tommy.spring.store.vo.PurchaseOrderRequest;
6	import tommy.spring.store.vo.PurchaseOrderResult;
7	public class OrderServiceTestThree {
8	private PlaceOrderService placeOrderService;
9	private AbstractApplicationContext context;
10	public OrderServiceTestThree() {
	String[] configLocations = new String[] {
11	"applicationContext.xml", "transactionThree.xml" };
12	context = new ClassPathXmlApplicationContext(configLocations);
	placeOrderService =

```

13         (PlaceOrderService) context.getBean("placeOrderService");
14     }
15     public void order() {
16         PurchaseOrderRequest orderRequest = new PurchaseOrderRequest();
17         orderRequest.setItemId(3);
18         orderRequest.setAddress("서울 강남구");
19         PurchaseOrderResult orderResult = placeOrderService.order(orderRequest);
20         System.out.println("주문상태 정보");
21         System.out.println("아이템 : " + orderResult.getItem().getId());
22         System.out.println("가격 : " + orderResult.getPaymentInfo().getPrice());
23     }
24     public void close() {
25         context.close();
26     }
27     public static void main(String[] args) {
28         OrderServiceTestThree test = new OrderServiceTestThree();
29         test.order();
30         test.close();
31     }
32 }

```

④ OrderServiceTestThree 클래스를 실행하여 결과를 확인해 보자.

The screenshot shows the IDE's 'Run' console and 'SQL' window. The console output shows the results of the 'order()' method call, and the SQL window shows the data inserted into the 'purchase_order' table.

Console Output:

```

<terminated> C
주문상태 정보
아이템 : 3
가격 : 10000

```

SQL Query Results:

PAYMENT_INFO_ID	PRICE
1	3 30000
2	1 50000
3	5 10000

PURCHASE_ORDER_ID	ITEM_ID	PAYMENT_INFO_ID	ADDRESS
1	3	2	3 서울 강남구
2	4	1	100 서울 송구
3	1	1	1 서울 송로구
4	2	1	100 서울 송구
5	5	3	5 서울 강남구

⑤ 앞의 예제처럼 이제 purchase_order 테이블에 아래와 같이 데이터를 하나 임의로 넣어 넣고 롤백이 되는지 테스트 해 보자. 데이터가 삽입 되었다면 OrderServiceTestThree 클래스를 실행하고 payment_info 테이블을 확인해 보자.

```

insert into purchase_order values(6, 1, 100, '서울 동작구');
commit;

```

The screenshot shows the IDE's console output after running the test. It displays a 'DuplicateKeyException' and a 'SQLIntegrityConstraintViolationException' with the message 'ORA-00001: unique constraint (MYTEST.PURCHASE_ORDER_PK) already exists'.

```

Exception in thread "main" org.springframework.dao.DuplicateKeyException: PreparedStatementCallback; SQL [INSERT INTO PURCHASE_ORDER VALUES (6, 1, 100, '서울 동작구');]; nested exception is java.sql.SQLIntegrityConstraintViolationException: ORA-00001: unique constraint (MYTEST.PURCHASE_ORDER_PK) already exists

```

■ payment_info 테이블에 들어갔던 결제정보가 취소되는 것을 확인할 수 있다.