Transactions

- **A series of actions that must be performed as a group**
- **If any part of the transaction fails, the entire transaction fails**

- Example: Performing a money transfer between two bank accounts

id	acctnum	balance
1	101	000000000000100.2500
2	102	000000000000300.5000

Move $50.00 from one account to another

```
start transaction;
update account set balance='50.25' where id=1;
update account set balance='350.50' where id=2;
commit;
```

id	acctnum	balance
1	101	000000000000050.2500
2	102	000000000000350.5000

Transactions

- **Once the transaction is started, the changes do not appear in the database until a commit operation**
- **Other ongoing database operations will not see the changes being made until a commit is made**
- **Prior to a commit, any changes made in the transaction can be rolled back**

Rollbacks

- **A Rollback means the entire operation is undone**

id	acctnum	balance
1	101	000000000000100.2500
2	102	000000000000300.5000

Move $50.00 from one account to another

```
start transaction;
update account set balance='50.25' where id=1;
rollback;
```

id	acctnum	balance
1	101	000000000000100.2500
2	102	000000000000300.5000

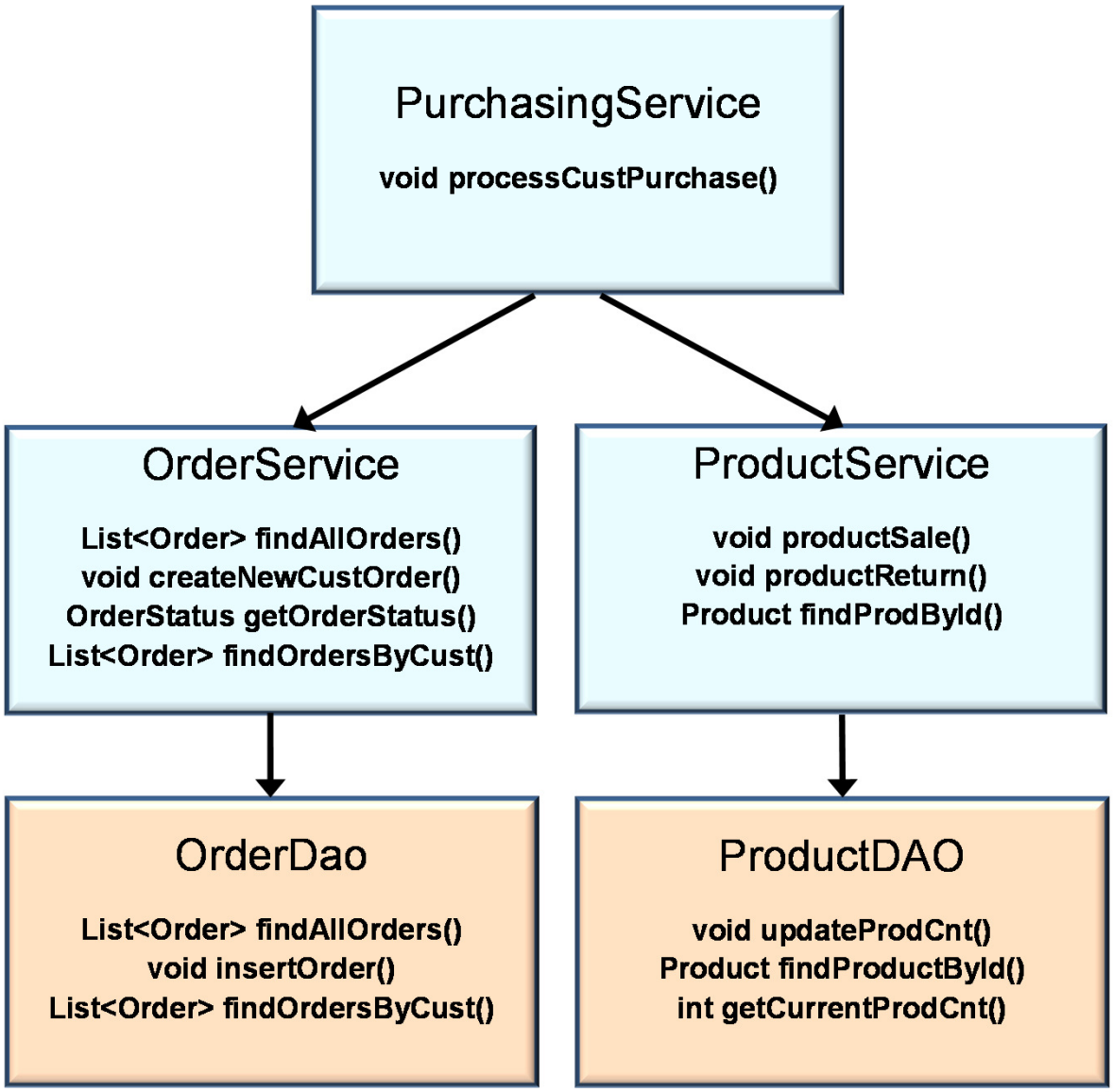
Database not changed

ACID

- **Requirements for Database reliability**
  - **Atomic**
    - **All or nothing (can't split an atom)**
  - **Consistent**
    - **The database goes from one consistent state to another**
    - **Never left in an intermediate state**
    - **Data always meets all validation rules**
  - **Isolated**
    - **When are intermediate results visible?**
    - **In progress changes are hidden from other on-going operations**
    - **Other transactions only see data changes after commited**
  - **Durable**
    - **Once a transaction completes, its changes will not be lost (even if database crashes)**

Transaction Placement

- **A customer makes a purchase. Where should the transaction start and end.**
- **If an error occurs anywhere in the purchase update, we should rollback: order table update, product table update, customer account table update**



Transaction Placement

- **Transactions are best placed on the Service Layer**
- **If any of the multiple DAO operations fail, the entire service operation should be rolled back**

PurchasingService

```
void processCustPurchase(Purchase purch) {
    /*   Transaction Start  */
    … Business logic to perform purchase …
    OrderService.createNewCustOrder(purch);
    ProductService.productSale(prod);
    AccountService.updateAcct(cust, newBal);
    /*   Transaction End  */
}
```

A Cross-Cutting Concern

- **It's a lot of work to add all the necessary transaction management code**
  - **Starting transactions**
  - **Rolling back transactions at various failure points**
  - **Ending transactions**
- **Transaction handling must be done in many Service classes and perhaps some DAO classes**
- **A perfect use case for AOP!**

Transactions in Spring

- **Spring will automatically add transaction management code where you request it**
  - **Use @Transactional**
  - **No transaction management in your Java code (thanks to AOP!)**
  - **Non-intrusive transaction management**
- **Any RuntimeException results in a rollback**
  - **It's assumed less serious Checked Exceptions can be recovered from and no rollback is necessary**
  - **This behavior can be changed**
- **Spring allows you to control**
  - **Isolation Levels**
  - **Transaction Styles (e.g. flat, nested)**

Transactions in Spring

- **Easy to Use**
- **Transaction code is independent of the transaction technology**
- **No knowledge required of the underlying transaction API**
- **XML-based configuration and Annotation-based approaches**
- **No need for application server**

Types of Transactions

- **Local**
  - **Single transaction resource**
  - **E.g., a JDBC connection with a single database**

- **Global**
  - **Multiple transactional resources**
  - **Multiple databases, perhaps on different servers**
  - **Must ensure correctness across multiple resources**
    - **a transaction may span more than one database on more than one server!**
  - **Requires an application server**
    - **Application server will coordinate multiple resource managers**

Transaction Manager

- **Start, maintain, commit or rollback transactions as required**
- **Even on transactions that span multiple functions in different classes**

- **Different Transaction Managers**
  - **JDBC**
    - **DataSourceTransactionManager**
  - **JTA (Java Transaction API)**
    - **Required for global transactions**
    - **JtaTransactionManager**
  - **Hibernate**
    - **HibernateTransactionManager**
  - **JPA**
    - **JpaTransactionManager**

Creating a Transaction Manager

1) Instantiate a transaction manager bean
2) Assign the transaction manager a datasource
3) Turn on transactional annotations

```
<!-- Instantiate TransactionManager and connect to a dataSource -->
<bean id="txManager"
    class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource" />
</bean>

<!-- Turn on Transactional annotations -->
<tx:annotation-driven transaction-manager="txManager" />
```

Using @Transactional

- Use on a class
  - All methods will have transaction advice applied

```
@Service("orderService")
@Transactional
public class OrderServiceImpl implements OrderService {
  @Autowired
  private OrderDAO orderDao;
```

- Use on a method
  - Advice only applied to the specific method

```
@Service("orderService")
public class OrderServiceImpl implements OrderService {
  @Autowired
  private OrderDAO orderDao;

  @Transactional
  public Order createNewCustOrder(String cusNum,
                  ArrayList<Product> prodsInOrder)
  {
     ...
  }
```

Rollback on Failure

- What should happen if while processing a Purchase:
  - OrderService inserts the new Order
  - ProductService fails (bad product name?)

```
@Transactional
@Service("purchaseService")
public class PurchaseServiceImpl implements PurchaseService {
  @Autowired
  OrderService orderService;
  @Autowired
  ProductService productService;

  public Order processCustomerPurchase(String cusNum, Product prod) {
    Order newOrder;

    newOrder = orderService.createNewCustOrder(cusNum, prod);
    productService.productSale(prod);

    return newOrder;
  }
}
```

Rollback on Failure

- Only RuntimeException results in a rollback of the transaction that was initiated on processCustomerPurchase()

```
try {
  order = purchaseService.processCustomerPurchase("6C779",
                    prodToOrder);
} catch (Exception ex) {
      System.out.println("Purchase Failed");
}
```

- Warning: Checked exceptions will not result in rollback by default

Attributes on @Transactional

Attribute	Value type	Description
timeout	numeric (sec.)	connection timeout
readOnly	`true|false`	Read/Write transaction
isolation	`Isolation` *(enum)*	isolation level
propagation	`Propagation` *(enum)*	propagation type
rollbackFor	`Throwable`	request rollback
noRollbackFor	`Throwable`	cancel rollback
rollbackForClassname	`Throwable` class name	request rollback
noRollbackForClassname	`Throwable` class name	cancel rollback

Changing Rollback Defaults

- Use rollbackFor to add exceptions that are not RuntimeExceptions
- Use noRollbackFor to prevent rollback for various RuntimeExceptions
- Can provide a single class or a list of classes

```
@Transactional(
    rollbackFor={java.io.IOException.class,
            com.npu.orderApp.exceptions.ProductDaoException.class},
    noRollbackFor = ArithmeticException.class)
@Service("purchaseService")
public class PurchaseServiceImpl implements PurchaseService {
```

```
 @Transactional(
            rollbackForClassName={"java.io.IOException"})
 @Service("purchaseService")
public class PurchaseServiceImpl implements PurchaseService {
```

Changing Rollback Defaults

- Do not convert your business logic exceptions to RuntimeException simply to provide for transaction rollbacks
- In many cases, you still want the benefits of Checked Exceptions
  - Reducing the likelihood of bugs by forcing callers to catch and handle bugs that should have recovery code
- For those cases add the rollbackFor or rollbackForClassName attribute

Rollback for Checked Exceptions

- If a bad Product name is used, throw a Checked Exception: ProductNotFoundException

ProductDAO class

```
public int updateTotalOrders(String prodName,
             int change) throws ProductNotFoundException
{
    String sql = "update product set totalOrders=:newTotalOrders where id=:id";
    int curTotalOrders, newTotalOrders;
    Product prod;
    MapSqlParameterSource params;
    int prodId, rowsAffected;

    prod = findProdByName(prodName);
    if (prod == null) {
        throw new ProductNotFoundException(prodName);
    }

    curTotalOrders = prod.getTotalOrders();
    newTotalOrders = curTotalOrders + change;
    prodId = prod.getId();

    params = new MapSqlParameterSource("id", prodId);
    params.addValue("newTotalOrders", newTotalOrders);
    rowsAffected = namedTemplate.update(sql, params);
    return rowsAffected;
  }
```

Rollback for Checked Exceptions

- Checked Exception ProductNotFoundException will result in a rollback
  - Possibly thrown by productService
  - New Customer Order entry should be rolled back

PurchaseService class

```
@Transactional(
    rollbackForClassName="ProductNotFoundException")
public Order processCustomerPurchase(
    String cusNum, String prodName,  float prodPrice)
    throws ProductNotFoundException
{
   Order newOrder;
   Product prod;

   newOrder =
     orderService.createNewCustOrder(cusNum, prodPrice);
   productService.productSale(prodName, 1);

   return newOrder;
}
```

Optimizing for Read Only

- readOnly property
  - Allows optimization of transactions when writes will not be occurring as part of the transaction
- Improving efficiency
  - Place a read-only property at class level
  - Override only those methods that will involve writes

```
@Transactional(readOnly=true)
public class OrderServiceImpl implements OrderService {
  public int getTotalOrders() {
    return orderDao.getOrderCount();
  }

  public Order findOrderById(int id) {
    Order order = orderDao.findOrderById(id);
    return order;
  }
  @Transactional(readOnly=false)
  public Order createNewCustOrder(String cusNum, Product prod)
    ...
    return newOrder;
  }
}
```

Transaction Propagation

- What happens when one Transactional method calls another Transactional method?

PurchaseService

```
@Transactional
public Order processCustomerPurchase(String cusNum, Product prod) {
    Order newOrder;

    newOrder = orderService.createNewCustOrder(cusNum, prod);
    productService.productSale(prod);

    return newOrder;
}
```

OrderService

```
@Transactional
public Order createNewCustOrder(String cusNum, Product prod) {
  ...

  return newOrder;
}
```

Transaction Propagation

- 3 Possibilities
  1. Continue the transaction (roll into the existing transaction)
  2. Start a new transaction for the new method
  3. Stop transactions for the new method

Propagation Rules

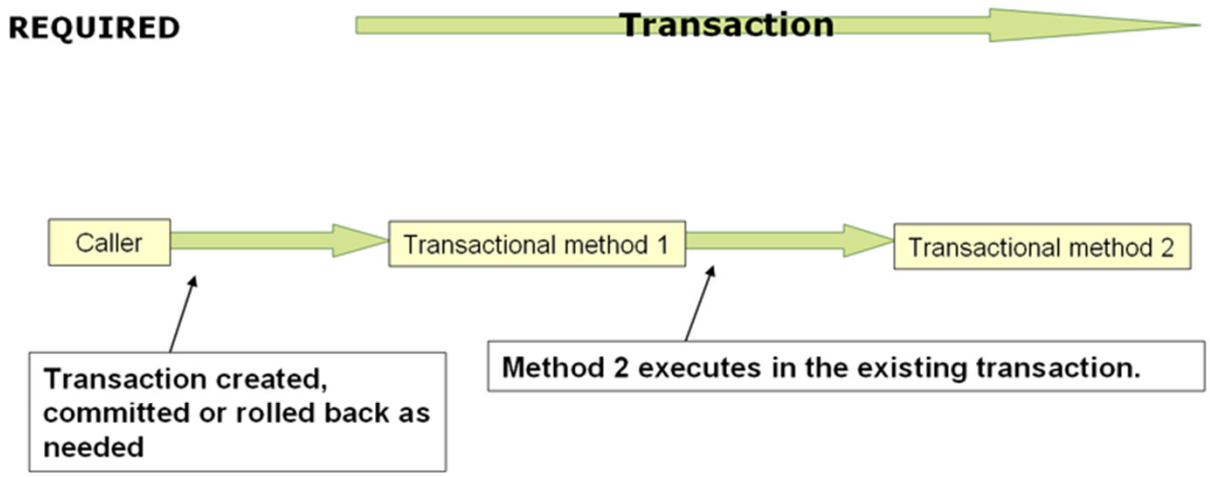
- REQUIRED (Default)
  - Execute method as part of current transaction or create a new transaction if one doesn't already exist
- REQUIRES_NEW
  - Create a new transaction for the method; suspend current transaction, if present
- NESTED
  - Continue the transaction (roll into the existing transaction)
- MANDATORY
  - Require transaction to already exist; throw exception if transaction not existing
- NEVER
  - Throw exception if transaction exists
- SUPPORTS
  - Use existing transaction, if exists; otherwise run without a transaction
- NOT_SUPPORTED
  - Run outside a transaction context

Specifying Propagation Rule

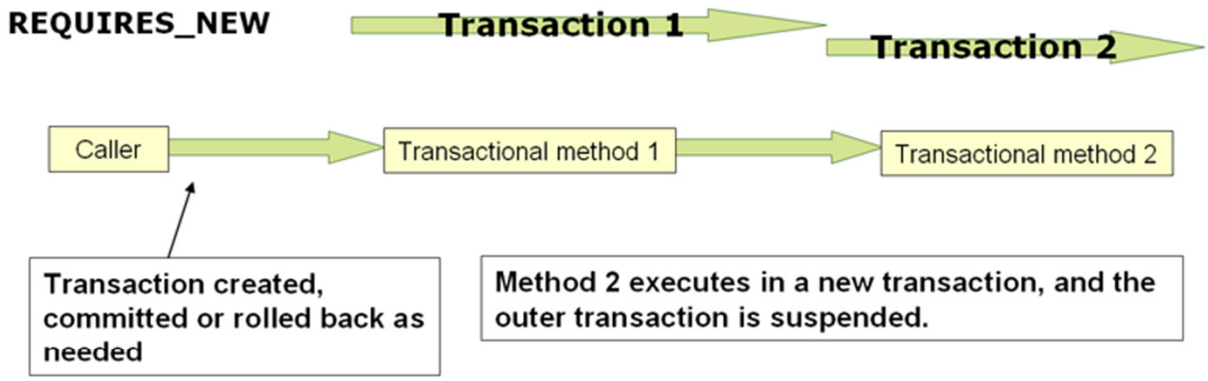
- Use propagation attribute

```
@Transactional(propagation=Propagation.REQUIRES_NEW)
public void productSale(String prodName, int numSold)
                    throws  ProductNotFoundException
{
    prodDao.updateTotalOrders(prodName, numSold);
}
```

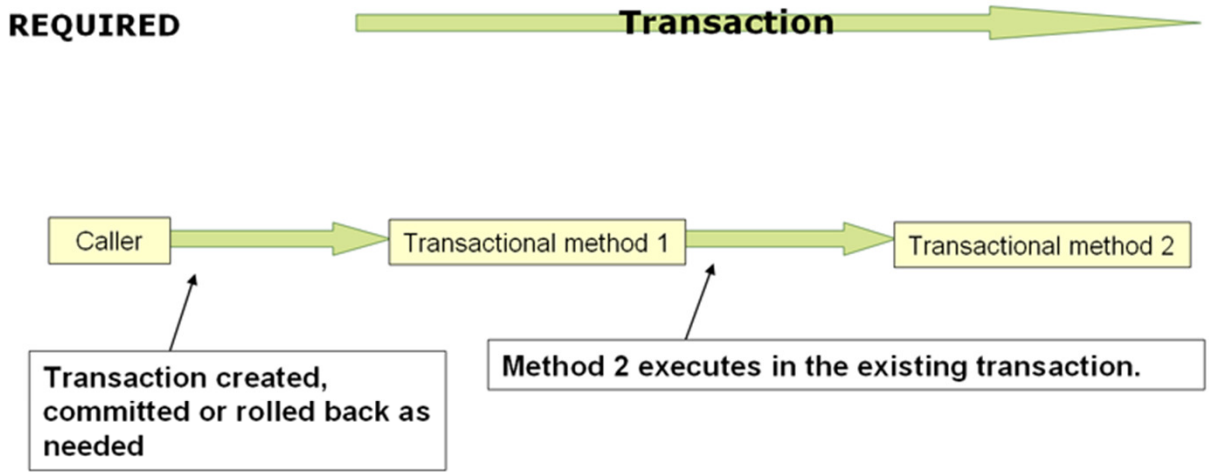
REQUIRED


REQUIRED
Transaction
Caller
Transactional method 1
Transactional method 2
Transaction created, committed or rolled back as needed
Method 2 executes in the existing transaction.

REQUIRES_NEW



NESTED



- Second method executes as part of the first transaction
- However, second method may be rolled back independently of the outer transaction
- If Outer method rolls back, inner method is rolled back, as well (but not vice versa)
- This is the idea of savepoints
  - One transaction
  - Rollback to last savepoint

Isolation Levels

- Defines how a transaction is affected by other on-going transactions

id	acctnum	balance
1	101	00000000000100.2500
2	102	00000000000300.5000

Transaction 1

```
start transaction;

select * from account where id=1;

update account set
        balance='50.25' where id=1;









select * from account where id=2;

rollback;
```

Transaction 2

```
start transaction;

select * from account
                    where id=1;

commit;
```

Two on-going transactions. What row should Transaction 2 return?

id	acctnum	balance
1	101	00000000000100.2500

id	acctnum	balance
1	101	00000000000050.2500

Transaction Isolation

- If the transactions are "isolated", Transaction 1 will not have any effect on Transaction 2

id	acctnum	balance
1	101	000000000000100.2500
2	102	000000000000300.5000

Transaction 1

```
start transaction;

select * from account where id=1;

update account set
        balance='50.25' where id=1;









select * from account where id=2;

rollback;
```

Transaction 2

```
start transaction;

select * from account
                    where id=1;

commit;
```

Isolated result of Transaction 2

id	acctnum	balance
1	101	000000000000100.2500

Transaction Isolation

- Transaction 1 makes the same query twice. Will the same data be returned?

id	acctnum	balance
1	101	00000000000100.2500
2	102	00000000000300.5000

Transaction 1

```
start transaction;

select * from account where
        balance>100.0;









select * from account where
     balance>100.0;

commit;
```

Transaction 2

```
start transaction;

update account set
      balance='50.25' where id=1;

commit;
```

What will the second query of Transaction 1 return?

id	acctnum	balance
1	101	00000000000100.2500
2	102	00000000000300.5000

2	102	00000000000300.5000

Transaction Isolation

- Complete isolation would give the same result for both queries of Transaction 1

id	acctnum	balance
1	101	00000000000100.2500
2	102	00000000000300.5000

Transaction 1

```
start transaction;

select * from account where
        balance>100.0;









select * from account where
     balance>100.0;

commit;
```

Transaction 2

```
start transaction;

update account set
      balance='50.25' where id=1;

commit;
```

What will the second query of Transaction 1 return?

id	acctnum	balance
1	101	00000000000100.2500
2	102	00000000000300.5000

Complete Isolation

- Complete isolation allows a transaction to complete its work without interference or changing data
- Expensive to implement
  - Locking Required
  - Performance Hit

Transaction Isolation

- Complete isolation would require making Transaction 2 wait until Transaction 1 completes

id	acctnum	balance
1	101	00000000000100.2500
2	102	00000000000300.5000

Transaction 1

```
start transaction;

select * from account where
            balance>100.0;









select * from account where
      balance>100.0;

commit;
```

Transaction 2

Waiting …

```
start transaction;

update account set
        balance='50.25' where id=1;

commit;
```

Isolation Levels

- Complete isolation is usually too costly
- By changing what isolation means, we can improve efficiency while still meeting our needs
- The ANSI/ISO SQL standard defines 4 different isolation levels

- What are the problems if isolation is compromised?

Dirty Reads

- Occur when one transaction reads data that has not been committed
- If the data change is later rolled back, the transaction will have used “bad data”

Non Repeatable Read

- A transaction performs the same query multiple times but instead of the same result, the data read is different

Transaction 1

```
start transaction;

select * from account where
        balance>100.0;




select * from account where
    balance>100.0;rollback;

commit;
```

id	acctnum	balance
1	101	00000000000100.2500
2	102	00000000000300.5000

Transaction 2

```
start transaction;

update account set
      balance='50.25' where id=1;

commit;
```

id	acctnum	balance
1	101	00000000000100.2500
2	102	00000000000300.5000

Output of Transaction 1

id	acctnum	balance
2	102	00000000000300.5000

Phantom Reads

- A transaction performs a query that returns multiple rows
- A second transaction inserts one or more rows
- The first transaction makes the same query and finds rows that weren't there before

Phantom Reads

- A transaction performs a query that returns multiple rows

Transaction 1

```
select * from account where
        balance>100.0;

select * from account where
     balance>100.0;

commit;
```

id	acctnum	balance
1	101	000000000000100.2500
2	102	000000000000300.5000

Transaction 2

```
start transaction;

insert into account
     (id,acctnum,balance) values
                    (3,75, 250.00);

commit;
```

id	acctnum	balance
1	101	000000000000100.2500
2	102	000000000000300.5000

Output of Transaction 1

id	acctnum	balance
1	101	000000000000050.2500
2	102	000000000000300.5000
3	75	000000000000250.0000

ANSI/ISO SQL Isolation Levels

- **Serializable**
  - Expensive to implement
- **Repeatable reads**
  - Phantom reads may occur
- **Read committed**
  - Non-repeatable reads may occur
- **Read uncommitted**
  - Dirty reads may occur

Tradeoff: Efficiency vs. Protection

Isolation level	Dirty reads	Non-repeatable reads	Phantoms
Read Uncommitted	may occur	may occur	may occur
Read Committed	-	may occur	may occur
Repeatable Read	-	-	may occur
Anomaly Serializable	-	-	-

MySQL default: Repeatable read

Specifying Isolation

- Use isolation attribute
  - ISOLATION_DEFAULT
  - ISOLATION_READ_UNCOMMITTED
  - ISOLATION_READ_COMMITTED
  - ISOLATION_REPEATABLE_READ
  - ISOLATION_SERIALIZABLE

```
@Transactional(isolation=Isolation.READ_COMMITTED)
public void productSale(String prodName, int numSold)
                throws ProductNotFoundException
{
    prodDao.updateTotalOrders(prodName, numSold);
}
```

- Default Spring behavior
  - ISOLATION_DEFAULT

Specifying Isolation

- The transaction isolation level is determined by the transaction start method
  - Isolation level will not change if other methods in the same transaction define different isolations

```
@Transactional(isolation=Isolation.READ_COMMITTED)
public Order processCustomerPurchase(String cusNum,
        String prodName, float prodPrice)  throws ProductNotFoundException
{
    Order newOrder;
    Product prod;

    newOrder = orderService.createNewCustOrder(cusNum, prodPrice);
    productService.productSale(prodName, 1);

    return newOrder;
  }
```

```
@Transactional(isolation=Isolation.REPEATABLE_READ)
public void productSale(String prodName, int numSold)
                throws  ProductNotFoundException
{
    prodDao.updateTotalOrders(prodName, numSold);
}
```

- Since there is only one transaction which starts as READ_COMMITTED, the entire transaction will be READ_COMMITTED.

Transactions and Testing

- Spring provides @TransactionConfiguration for use with testing
  - Default is to automatically rollback transactions at the end of each test
  - Used at the class level
  - Make sure you also make the class @Transactional

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = { "/test.xml" })
@TransactionConfiguration
@Transactional
public class OrderServiceTest {
  ...
}
```

- Can turn off default rollback

```
@TransactionConfiguration(defaultRollback=false)
```