CS544
Enterprise Architecture
Midterm May 2017

Name____

Student ID ____

NOTE: This material is private and confidential. It is the property of MUM and is not to be disseminated.

1. [15 points] **Circle w**hich of the following is TRUE/FALSE concerning Spring Transaction Management:

T **F** Every interaction with an RDBMS requires a transaction whether a READ or a WRITE. Without a Transaction Management capability like Spring's, DB operations would fail.

EXPLAIN**:__ RDBMS have a built-in transaction capability. The advantage Spring TM provides is capability to manage across multiple DB interaction**

T F Spring Transaction Management is based on a logical unit of work.

EXPLAIN:__ **Spans one or MORE DB requests AND "Atomic" across those requests…Either ALL or None**

T F Spring Transaction Management with JPA requires a Persistence Context.

EXPLAIN:____ **Persistence Context is needed. It establishes the DB Connection & maintains a cache for "DB-aware" objects.**

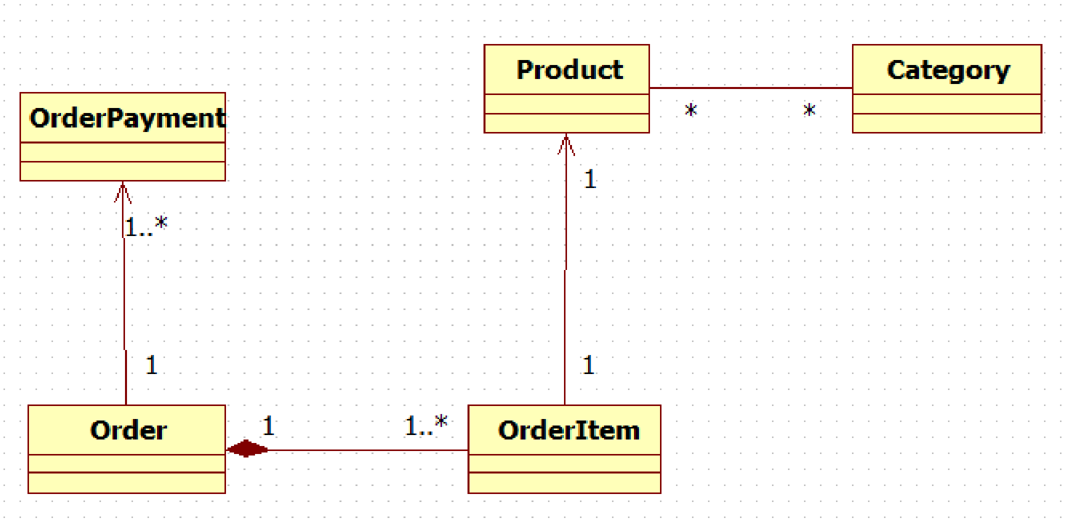
T **F** Spring @Transaction has no built-in metadata for managing any of the DB ACID properties.

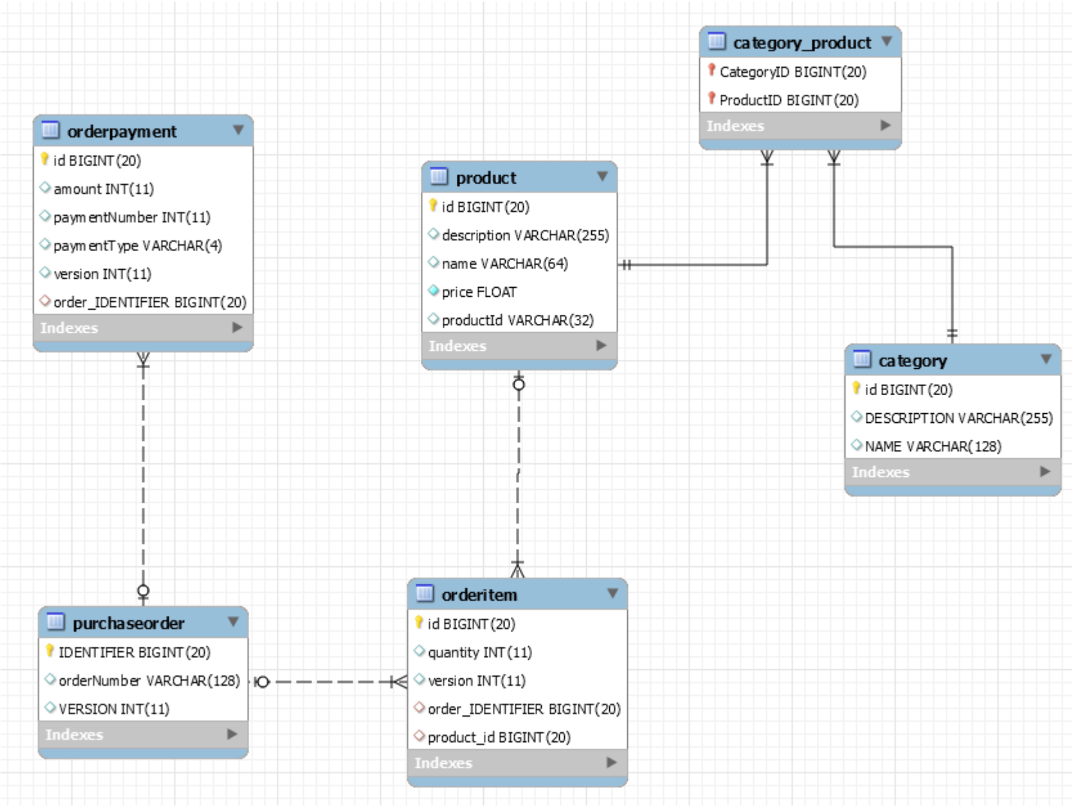
EXPLAIN:____ **@Transactional has an optional parameter for indicating isolation levels. For example - @Transactional (*isolation=Isolation.READ_COMMITTED)***

T F Spring Declarative Transaction Management requires little or no application code related to transaction management.

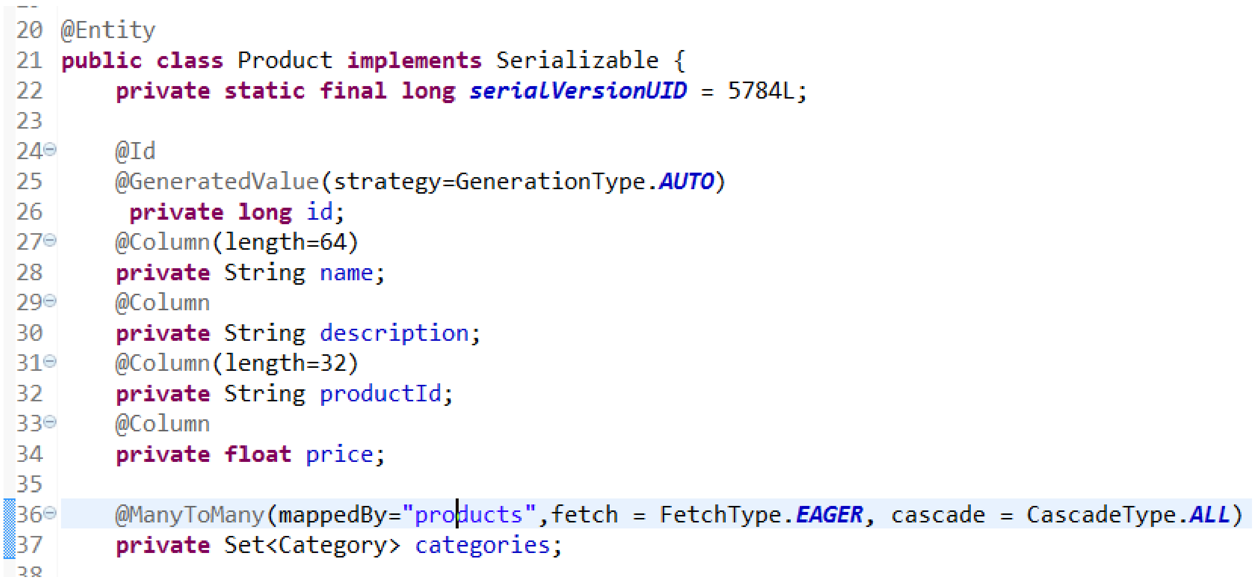
EXPLAIN:____ **Spring Declarative TM has little impact on application code…Can simply annotate a class with @Transactional & the Spring framework takes care of the details.**

2. [20 points] Annotate the Domain Objects based on the Domain Model and Entity Relationship Diagram provided. NOTE: All the fields are not listed. Only annotate the fields that are listed.



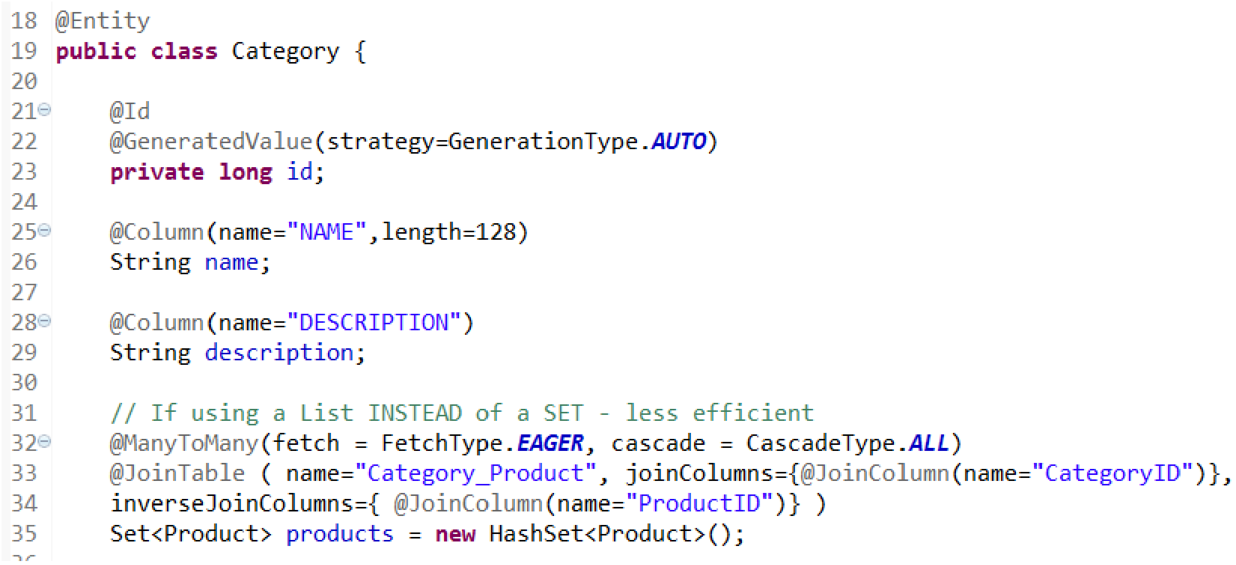


Product.java



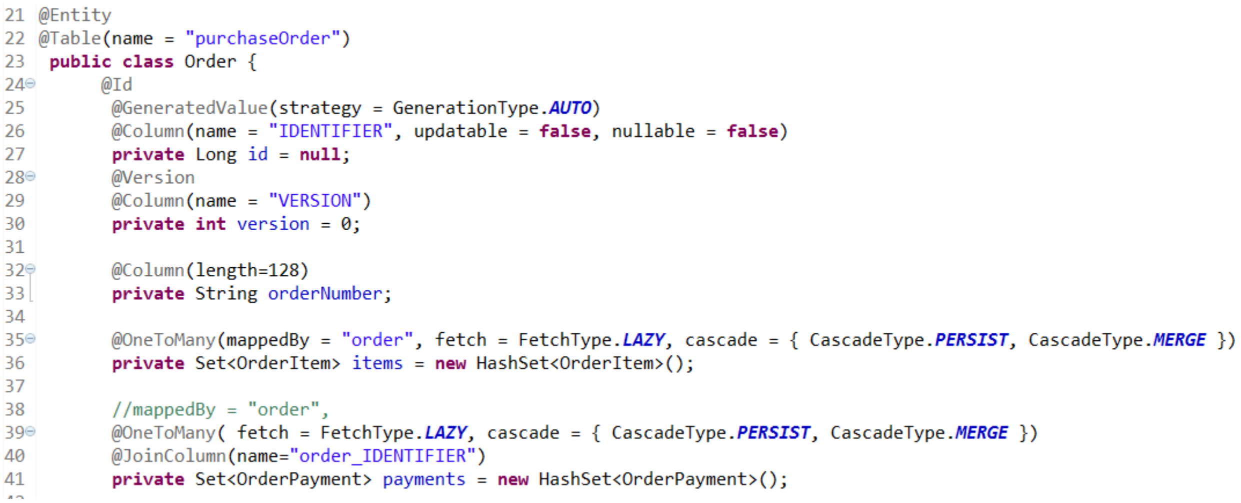
```java
@Entity
public class Product implements Serializable {
    private static final long serialVersionUID = 5784L;

    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
     private long id;
    @Column(length=64)
    private String name;
    @Column
    private String description;
    @Column(length=32)
    private String productId;
    @Column
    private float price;

    @ManyToMany(mappedBy="products",fetch = FetchType.EAGER, cascade = CascadeType.ALL)
    private Set<Category> categories;
```

Category.java



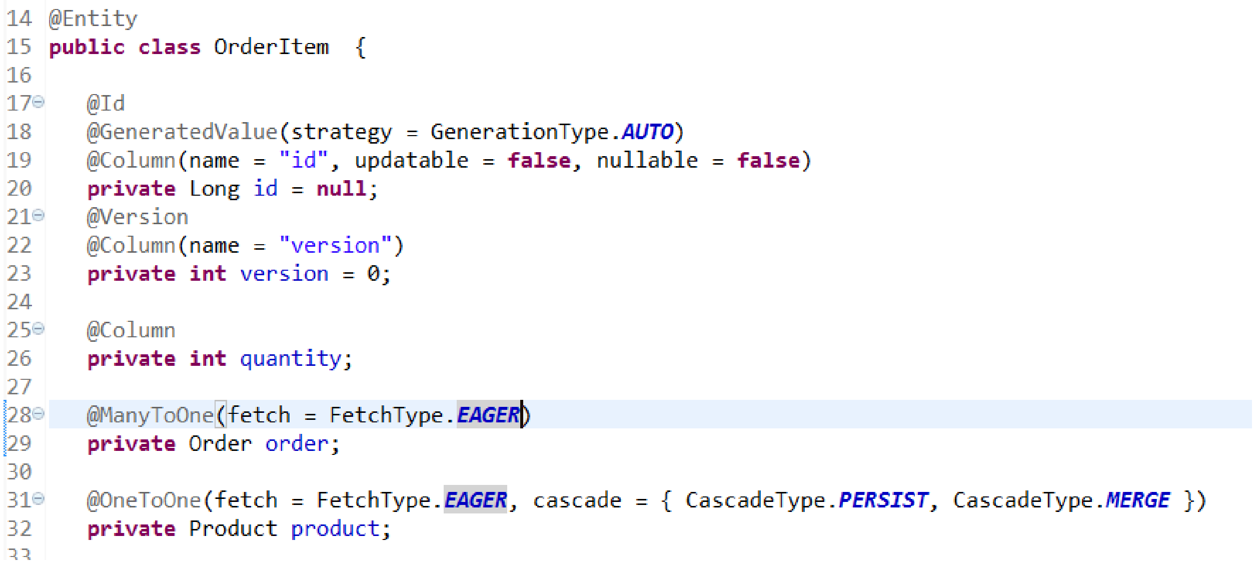
```java
@Entity
public class Category {

    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private long id;

    @Column(name="NAME",length=128)
    String name;

    @Column(name="DESCRIPTION")
    String description;

    // If using a List INSTEAD of a SET - less efficient
    @ManyToMany(fetch = FetchType.EAGER, cascade = CascadeType.ALL)
    @JoinTable ( name="Category_Product", joinColumns={@JoinColumn(name="CategoryID")},
    inverseJoinColumns={ @JoinColumn(name="ProductID")} )
    Set<Product> products = new HashSet<Product>();
```

Order.java



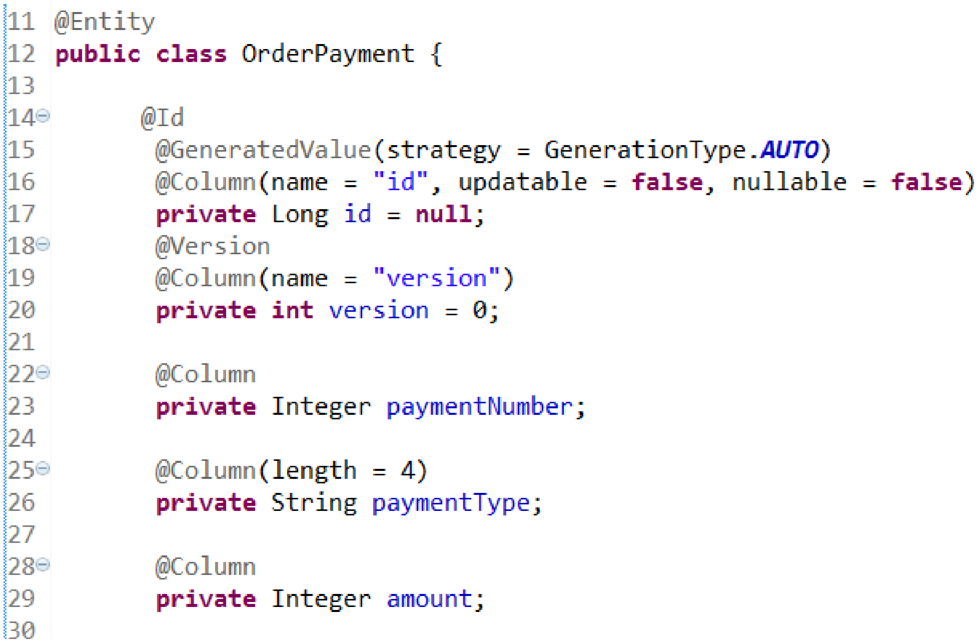
```java
@Entity
@Table(name = "purchaseOrder")
 public class Order {
       @Id
        @GeneratedValue(strategy = GenerationType.AUTO)
        @Column(name = "IDENTIFIER", updatable = false, nullable = false)
        private Long id = null;
        @Version
        @Column(name = "VERSION")
        private int version = 0;

        @Column(length=128)
        private String orderNumber;

        @OneToMany(mappedBy = "order", fetch = FetchType.LAZY, cascade = { CascadeType.PERSIST, CascadeType.MERGE })
        private Set<OrderItem> items = new HashSet<OrderItem>();

        //mappedBy = "order",
        @OneToMany( fetch = FetchType.LAZY, cascade = { CascadeType.PERSIST, CascadeType.MERGE })
        @JoinColumn(name="order_IDENTIFIER")
        private Set<OrderPayment> payments = new HashSet<OrderPayment>();
```

OrderItem.java



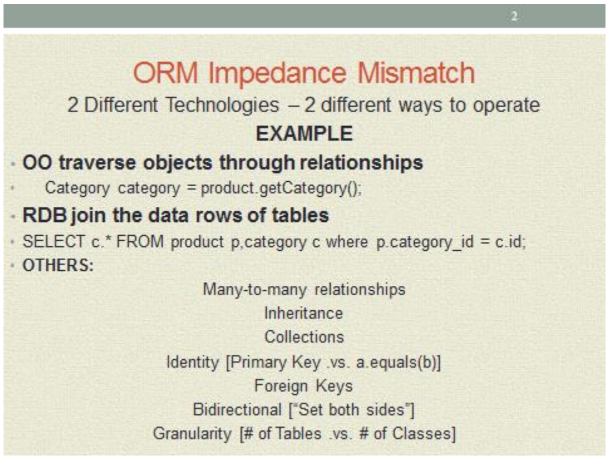
```java
@Entity
public class OrderItem  {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name = "id", updatable = false, nullable = false)
    private Long id = null;
    @Version
    @Column(name = "version")
    private int version = 0;

    @Column
    private int quantity;

    @ManyToOne(fetch = FetchType.EAGER)
    private Order order;

    @OneToOne(fetch = FetchType.EAGER, cascade = { CascadeType.PERSIST, CascadeType.MERGE })
    private Product product;
```

OrderPayment.java



```java
@Entity
public class OrderPayment {

    @Id
     @GeneratedValue(strategy = GenerationType.AUTO)
     @Column(name = "id", updatable = false, nullable = false)
     private Long id = null;
     @Version
     @Column(name = "version")
     private int version = 0;

     @Column
     private Integer paymentNumber;

     @Column(length = 4)
     private String paymentType;

     @Column
     private Integer amount;

```

3. [15 points] The reason for an ORM is because object models and relational models do not work very well together. Describe what is known as the Object-Relational Impedance mismatch. Give specific examples of the problems that arise from the mismatch.



2

ORM Impedance Mismatch

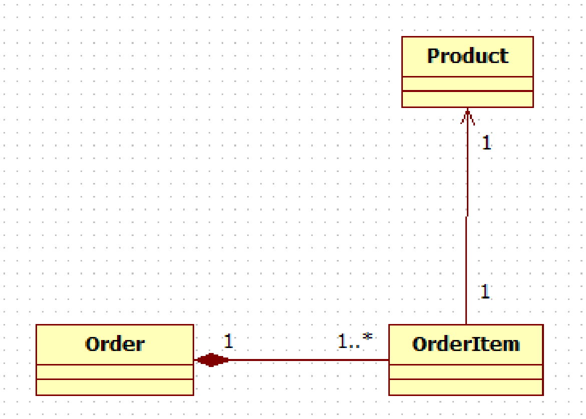
2 Different Technologies – 2 different ways to operate

EXAMPLE

- **OO traverse objects through relationships**
- Category category = product.getCategory();
- **RDB join the data rows of tables**
- SELECT c.* FROM product p,category c where p.category_id = c.id;
- **OTHERS:**

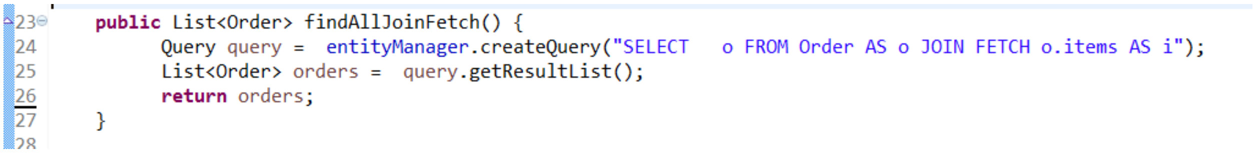
Many-to-many relationships
Inheritance
Collections
Identity [Primary Key .vs. a.equals(b)]
Foreign Keys
Bidirectional ["Set both sides"]
Granularity [# of Tables .vs. # of Classes]

4. [15 points] For the following relationships implement a Join fetch of all Orders with their Order Item collection.



What performance problem does the Join fetch address? Give details.
What performance problem does it cause? Give details.
What can be done to “clean up” the data returned by the fetch?

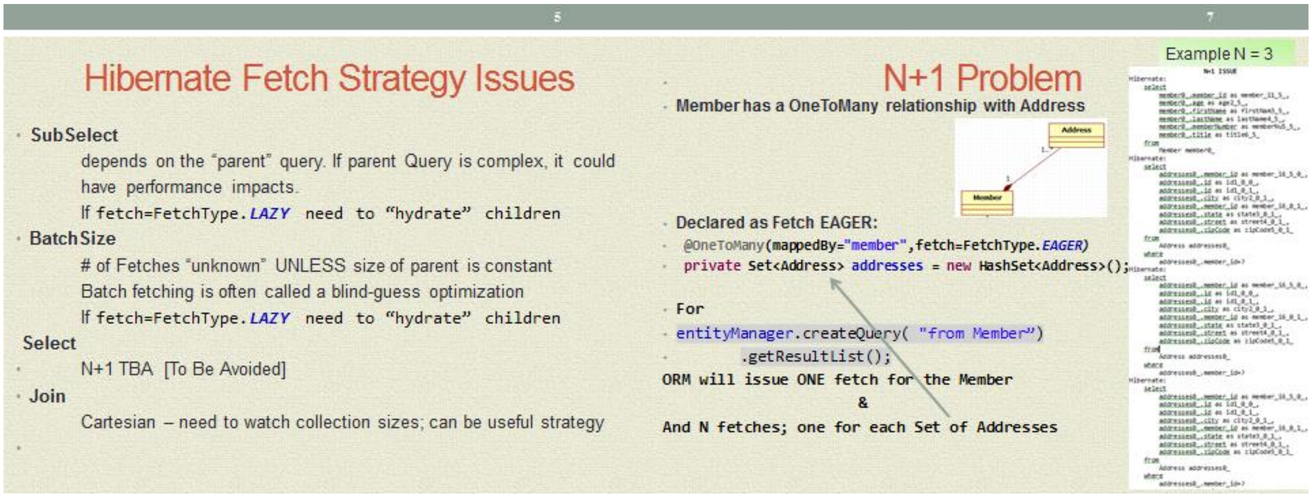
In OrderDaoImpl.Java



```
public List<Order> findAllJoinFetch() {
      Query query =  entityManager.createQuery("SELECT   o FROM Order AS o JOIN FETCH o.items AS i");
      List<Order> orders =  query.getResultList();
      return orders;
}
```

Join Fetch does ONE fetch for ALL collections.
The Join Fetch will get ALL the Orders AND OrderItems in ONE Select/fetch.

It solves the N+1 issue.
However it suffers from the Cartesian product issue.



Hibernate Fetch Strategy Issues

- SubSelect
  depends on the “parent” query. If parent Query is complex, it could have performance impacts.
  If `fetch=FetchType.LAZY` need to “hydrate” children
- Batch Size
  # of Fetches “unknown” UNLESS size of parent is constant
  Batch fetching is often called a blind-guess optimization
  If `fetch=FetchType.LAZY` need to “hydrate” children

Select
- N+1 TBA [To Be Avoided]
- Join
  Cartesian – need to watch collection sizes; can be useful strategy

N+1 Problem

- Member has a OneToMany relationship with Address

- Declared as Fetch EAGER:

```
@OneToMany(mappedBy="member",fetch=FetchType.EAGER)
private Set<Address> addresses = new HashSet<Address>();
```

- For

```
entityManager.createQuery( "from Member”)
        .getResultList();
ORM will issue ONE fetch for the Member
                    &
And N fetches; one for each Set of Addresses
```

Example N = 3

```
                    N+1 ISSUE
Hibernate:
    select
        member0_.member_id as member_i1_5_,
        member0_.age as age2_5_,
        member0_.firstName as firstNam3_5_,
        member0_.lastName as lastName4_5_,
        member0_.memberNumber as memberNu5_5_,
        member0_.title as title6_5_
    from
        Member member0_
Hibernate:
    select
        addresses0_.member_id as member_i6_5_0_,
        addresses0_.id as id1_0_0_,
        addresses0_.id as id1_0_1_,
        addresses0_.city as city2_0_1_,
        addresses0_.member_id as member_i6_0_1_,
        addresses0_.state as state3_0_1_,
        addresses0_.street as street4_0_1_,
        addresses0_.zipCode as zipCode5_0_1_
    from
        Address addresses0_
    where
        addresses0_.member_id=?
Hibernate:
    select
        addresses0_.member_id as member_i6_5_0_,
        addresses0_.id as id1_0_0_,
        addresses0_.id as id1_0_1_,
        addresses0_.city as city2_0_1_,
        addresses0_.member_id as member_i6_0_1_,
        addresses0_.state as state3_0_1_,
        addresses0_.street as street4_0_1_,
        addresses0_.zipCode as zipCode5_0_1_
    from
        Address addresses0_
    where
        addresses0_.member_id=?
Hibernate:
    select
        addresses0_.member_id as member_i6_5_0_,
        addresses0_.id as id1_0_0_,
        addresses0_.id as id1_0_1_,
        addresses0_.city as city2_0_1_,
        addresses0_.member_id as member_i6_0_1_,
        addresses0_.state as state3_0_1_,
        addresses0_.street as street4_0_1_,
        addresses0_.zipCode as zipCode5_0_1_
    from
        Address addresses0_
    where
        addresses0_.member_id=?
```

Cartesian Product Problem

- For sets A and B, the Cartesian product is A × B
- For sets A,B and C, the Cartesian product is A × B × C - etc.

2 Members
X
Address/Member

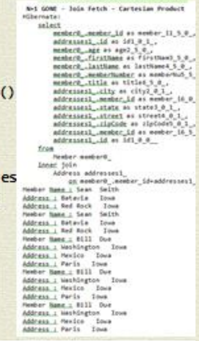
- Member [STILL] has a OneToMany relationship with Address
- NOW - Declared as Fetch LAZY:
- `@OneToMany(mappedBy="member",fetch=FetchType.LAZY)`
- `private Set<Address> addresses = new HashSet<Address>()`
- For:
- `Query query=entityManager.createQuery("SELECT m`
  `FROM Member AS m JOIN FETCH m.addresses AS a");`
- **`ORM will do ONE Fetch BUT will generate duplicates`**
- # Members x # Addresses [per Member]
- ORM NOTE: Product =

of "root" table results

X

# of results in individual "root" table child table.

Sean has 2 Addresses so 2 copies 2; Bill has 3 so 3 copies



```
N+1 GONE - Join Fetch - Cartesian Product
Hibernate:
    select
        member0_.member_id as member_11_5_0_,
        addresses1_.id as id1_0_1_,
        member0_.age as age2_5_0_,
        member0_.firstName as firstNam3_5_0_,
        member0_.lastName as lastName4_5_0_,
        member0_.memberNumber as memberNu5_5_
        member0_.title as title6_5_0_,
        addresses1_.city as city2_0_1_,
        addresses1_.member_id as member_16_0_
        addresses1_.state as state3_0_1_,
        addresses1_.street as street4_0_1_,
        addresses1_.zipCode as zipCode5_0_1_,
        addresses1_.member_id as member_16_5_
        addresses1_.id as id1_0_0__
    from
        Member member0_
    inner join
        Address addresses1_
            on member0_.member_id=addresses1_
Member Name : Sean  Smith
Address : Batavia   Iowa
Address : Red Rock   Iowa
Member Name : Sean  Smith
Address : Batavia   Iowa
Address : Red Rock   Iowa
Member Name : Bill  Due
Address : Washington   Iowa
Address : Mexico   Iowa
Address : Paris   Iowa
Member Name : Bill  Due
Address : Washington   Iowa
Address : Mexico   Iowa
Address : Paris   Iowa
Member Name : Bill  Due
Address : Washington   Iowa
Address : Mexico   Iowa
Address : Paris   Iowa
```

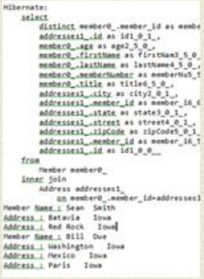
"Reduce" the Cartesian Product

- `Query query=entityManager.createQuery("SELECT DISTINCT m`
  `FROM Member AS m JOIN FETCH m.addresses AS a");`

- DISTINCT keyword removes duplicates

However

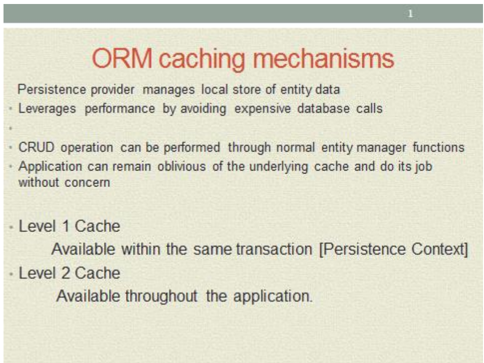
It accomplishes it in Memory [After DB fetch]



```
Hibernate:
    select
        distinct member0_.member_id as membe
        addresses1_.id as id1_0_1_,
        member0_.age as age2_5_0_,
        member0_.firstName as firstNam3_5_0_
        member0_.lastName as lastName4_5_0_,
        member0_.memberNumber as memberNu5_5
        member0_.title as title6_5_0_,
        addresses1_.city as city2_0_1_,
        addresses1_.member_id as member_16_0
        addresses1_.state as state3_0_1_,
        addresses1_.street as street4_0_1_,
        addresses1_.zipCode as zipCode5_0_1_
        addresses1_.member_id as member_16_5
        addresses1_.id as id1_0_0__
    from
        Member member0_
    inner join
        Address addresses1_
            on member0_.member_id=addresses1
Member Name : Sean  Smith
Address : Batavia   Iowa
Address : Red Rock   Iowa
Member Name : Bill  Due
Address : Washington   Iowa
Address : Mexico   Iowa
Address : Paris   Iowa
```

5. [15 points] Explain the concept of ORM caching. Include a discussion of :
   - First level relate to Persistence Context; Fetch Strategy
   - Second level
     - Read-only - read-write
     - Second-level .vs. query
     - When do you decide to use a second level cache?

Be specific. Give examples. Diagrams are good.

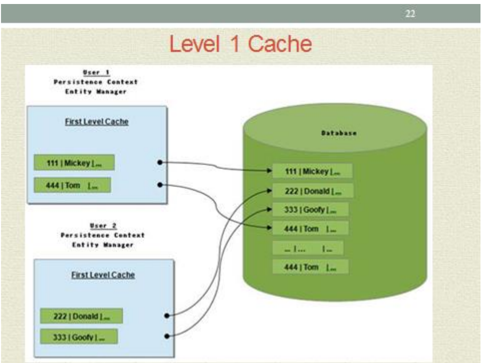


1

ORM caching mechanisms

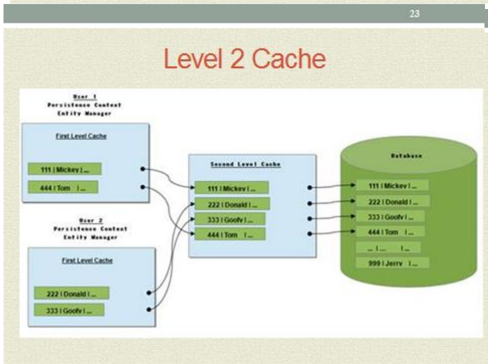
Persistence provider manages local store of entity data

- Leverages performance by avoiding expensive database calls
- CRUD operation can be performed through normal entity manager functions
- Application can remain oblivious of the underlying cache and do its job without concern
- Level 1 Cache
  Available within the same transaction [Persistence Context]
- Level 2 Cache
  Available throughout the application.



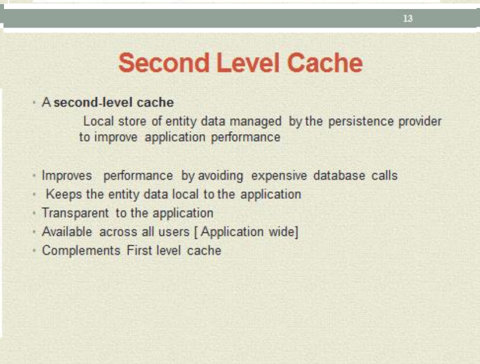
22

Level 1 Cache



23

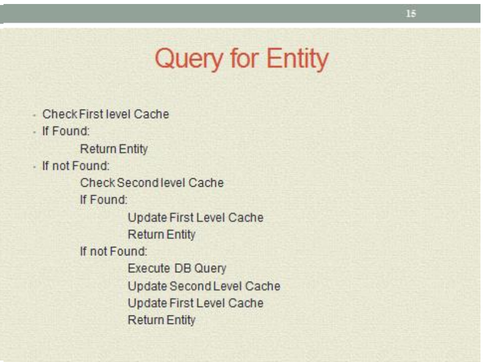
Level 2 Cache



13

Second Level Cache

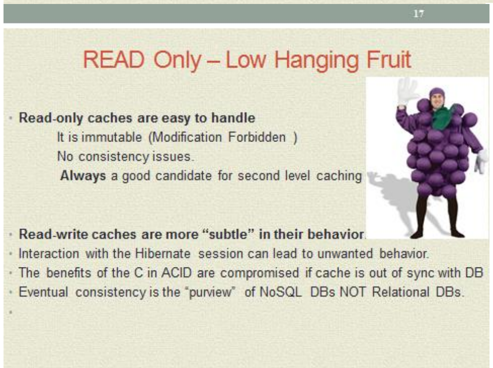
- A **second-level cache**
  Local store of entity data managed by the persistence provider to improve application performance
- Improves performance by avoiding expensive database calls
- Keeps the entity data local to the application
- Transparent to the application
- Available across all users [Application wide]
- Complements First level cache



16

Query for Entity

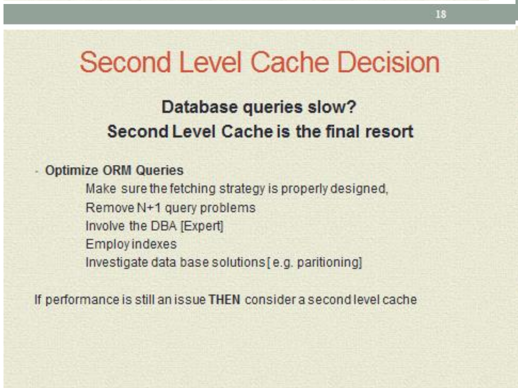
- Check First level Cache
- If Found:
  Return Entity
- If not Found:
  Check Second level Cache
  If Found:
    Update First Level Cache
    Return Entity
  If not Found:
    Execute DB Query
    Update Second Level Cache
    Update First Level Cache
    Return Entity



17

READ Only – Low Hanging Fruit

- **Read-only caches are easy to handle**
  It is immutable (Modification Forbidden)
  No consistency issues.
  Always a good candidate for second level caching
- **Read-write caches are more “subtle” in their behavior**
- Interaction with the Hibernate session can lead to unwanted behavior.
- The benefits of the C in ACID are compromised if cache is out of sync with DB
- Eventual consistency is the “purview” of NoSQL DBs NOT Relational DBs.



18

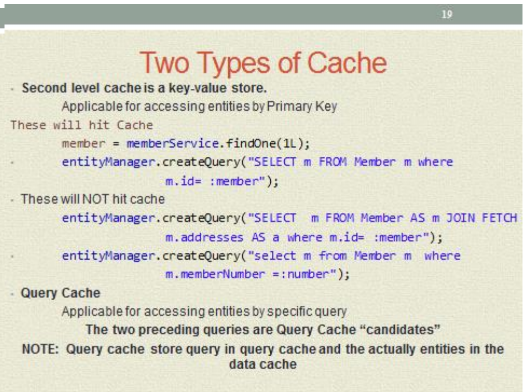
Second Level Cache Decision

Database queries slow?

Second Level Cache is the final resort

- **Optimize ORM Queries**
  Make sure the fetching strategy is properly designed,
  Remove N+1 query problems
  Involve the DBA [Expert]
  Employ indexes
  Investigate data base solutions [e.g. paritioning]

If performance is still an issue **THEN** consider a second level cache



19

Two Types of Cache

- **Second level cache is a key-value store.**
  Applicable for accessing entities by Primary Key

```
These will hit Cache
        member = memberService.findOne(1L);
        entityManager.createQuery("SELECT m FROM Member m where
                            m.id= :member");
```

- These will NOT hit cache

```
        entityManager.createQuery("SELECT  m FROM Member AS m JOIN FETCH
                            m.addresses AS a where m.id= :member");
        entityManager.createQuery("select m from Member m  where
                            m.memberNumber =:number");
```

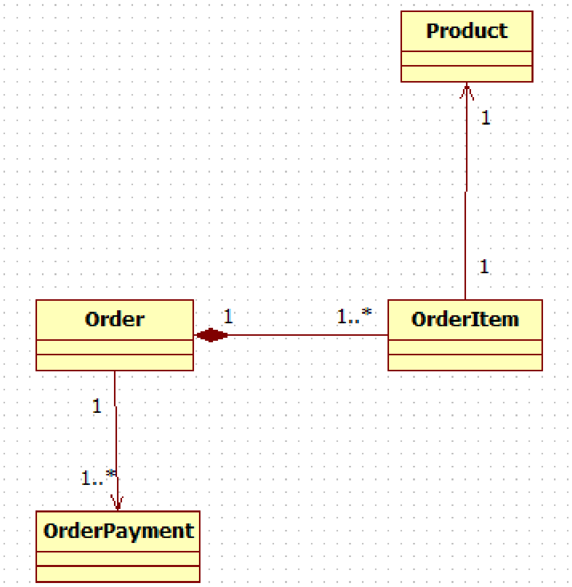
- **Query Cache**
  Applicable for accessing entities by specific query
  The two preceding queries are Query Cache “candidates”

NOTE: Query cache store query in query cache and the actually entities in the data cache

6. [15 points] Implement a parameterized JQPL query with this signature:

```
public List<Product> findByAmountRangeAndQuantity(Integer minPayment,
                                                   Integer maxPayment,
                                                   Integer quantity)
```

The query looks up all Product[s] where the Order Item quantity is greater than the supplied quantity and the Order Payment Amount is within the supplied parameters.

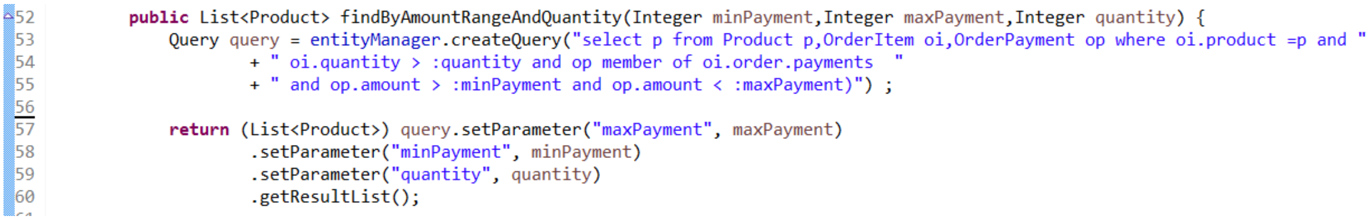


The Query should be a parameterized query. Also show the modifications to all classes in order to adhere to the N-Tier architecture convention. Identify the specific packages that each modified class is in.

edu.mum.dao. ProductDao

```
public List<Product> findByAmountRangeAndQuantity(Integer minPayment, Integer maxPayment,Integer quantity)
```

edu.mum.dao.impl. ProductDaoImpl



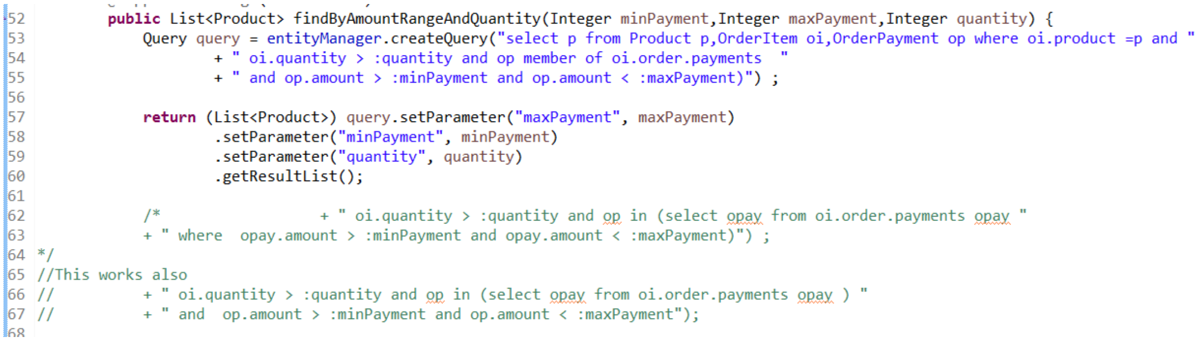
```
    public List<Product> findByAmountRangeAndQuantity(Integer minPayment,Integer maxPayment,Integer quantity) {
        Query query = entityManager.createQuery("select p from Product p,OrderItem oi,OrderPayment op where oi.product =p and "
                + " oi.quantity > :quantity and op member of oi.order.payments  "
                + " and op.amount > :minPayment and op.amount < :maxPayment)") ;

        return (List<Product>) query.setParameter("maxPayment", maxPayment)
                .setParameter("minPayment", minPayment)
                .setParameter("quantity", quantity)
                .getResultList();
```

edu.mum.service. ProductService

```
public List<Product> findByAmountRangeAndQuantity(Integer minPayment, Integer maxPayment,Integer quantity)
```

edu.mum.service.impl. ProductService Impl

```
public List<Product> findByAmountRangeAndQuantity(Integer minPayment, Integer maxPayment,Integer quantity) {
        return productDao.findByAmountRangeAndQuantity(minPayment, maxPayment,quantity);
    }
```



```java
        public List<Product> findByAmountRangeAndQuantity(Integer minPayment,Integer maxPayment,Integer quantity) {
            Query query = entityManager.createQuery("select p from Product p,OrderItem oi,OrderPayment op where oi.product =p and "
                    + " oi.quantity > :quantity and op member of oi.order.payments   "
                    + " and op.amount > :minPayment and op.amount < :maxPayment)") ;

            return (List<Product>) query.setParameter("maxPayment", maxPayment)
                    .setParameter("minPayment", minPayment)
                    .setParameter("quantity", quantity)
                    .getResultList();

            /*                  + " oi.quantity > :quantity and op in (select opay from oi.order.payments opay "
            + " where  opay.amount > :minPayment and opay.amount < :maxPayment)") ;
*/
//This works also
//          + " oi.quantity > :quantity and op in (select opay from oi.order.payments opay ) "
//          + " and  op.amount > :minPayment and op.amount < :maxPayment");
```