CS544
Enterprise Architecture
Exam 1 March 2018

Name_____

Student ID _____

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

1. [10 points] **Circle w**hich of the following is TRUE/FALSE concerning Spring Inversion of Control/Dependency Injection:

   T **F** Only Managed Beans can be injected in Spring, a POJO or JavaBean cannot.

   EXPLAIN:____If the POJO or JavaBean is a Spring Managed bean, they can be injected.

   T **F** @Autowired works only on interfaces. It cannot work directly on classes.

   EXPLAIN:__ It can work on classes. However you lose some of the value, testing; changing implementations

   T F In practice, the IoC container is not exactly the same as Dependency Injection as it involves a discovery step concerning the dependency.

   EXPLAIN:__ IoC involves a "look up the dependency"[Pull] step before injecting it. Spring has declarative configuration that identifies "where to find" the resource to inject [Push].

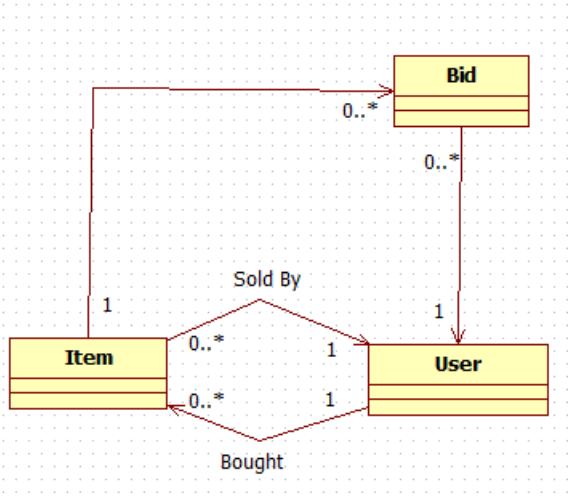
   T **F** A domain object needed in a @Service class is usually a good candidate for Dependency Injection

   EXPLAIN:____ DI is mainly used for "cross" layer transitions, access plumbing resources, etc. NOT for passing business data around…

   T **F** In Spring, DI can be done through either XML or through Annotations. They are mutually exclusive. That means, if you use XML for DI for one bean, they you should use it, exclusively for all beans.

   EXPLAIN:____ You could inject a DAO[memberDAO in memberService] through XML & inject another DAO[productDAO in productService] through @Autowired. Remember, XML configuration is the final word – if you configure the same bean twice, the XML configuration will take precedence.

2. [15 points] For the following relationships implement a SubSelect that fetches all items with their corresponding collection of bids.



What performance problem[s] does the SubSelect fetch address? What are its issues?
How does it work? – Explain the "algorithm" based on a universe of 10 Items each with a collection of 5-10 Bids.
Compare it to Join Fetch.

In Item.Java

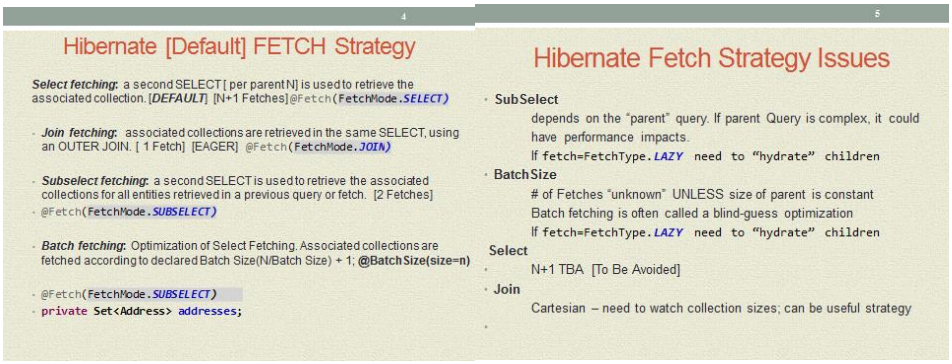
```
public class Item {

     @OneToMany(cascade = CascadeType.ALL, fetch = FetchType.LAZY)
     @Fetch(FetchMode.SUBSELECT)
      private List<Bid> bids = new ArrayList<Bid>();
```

In ItemServiceImpl.Java

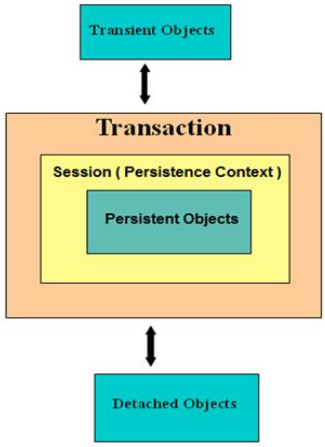
```
      public List<Item> findbySubSelect() {

            List<Item> items = (List<Item>)this.findAll();
            // hydrate since LAZY load
            items.get(0).getBids().get(0);

            return items;
      }
```

Subselect does ONE fetch for the Items and ONE fetch for ALL collections. Therefore the number of Items & Bids in the collections does not matter,
It solves the Cartesian issue && the N+1 issue.
If the collection is LAZY loaded, then doing a get all for the "parent" entities [Item as in above] will get the "parent"/Item list. Within the same transaction, a reference to ONE of the Bid collections will fetch ALL the collections.

The Join Fetch, on the other hand will get ALL the Bids AND Items in ONE Select/fetch.
The Join Fetch, however suffers from the Cartesian product issue [ItemsXBids] which means that more than one copy of each Item/Bid pair will be included in the fetch. The number of copies depends on the number of Bids in the collection. So if Item A has 3 bids, 3 copies of Item A will be present.



3. [15 points] In JPA, the Persistence Context plays an important role in the implementation of an ORM. Explain, by example the Entity lifecycle as it pertains to the following drawing.



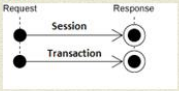
ORM Session

Spring "manages" through @Transactional

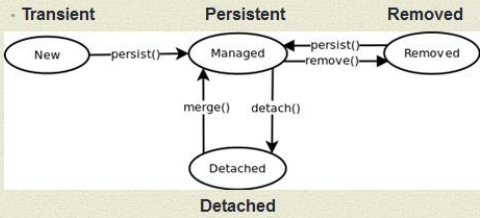
- Transaction Unit of work
  - **Common Pattern: *session-per-request***
  - Session == Database Transaction
- START –
  - Open a Session
  - Open a single database connection
  - Start a Transaction
  - Associate & Manage entities W/R the session
  - Exercise DB CRUD operations

Session == Persistence Context

- END –
  - End Transaction
  - Close a Session.



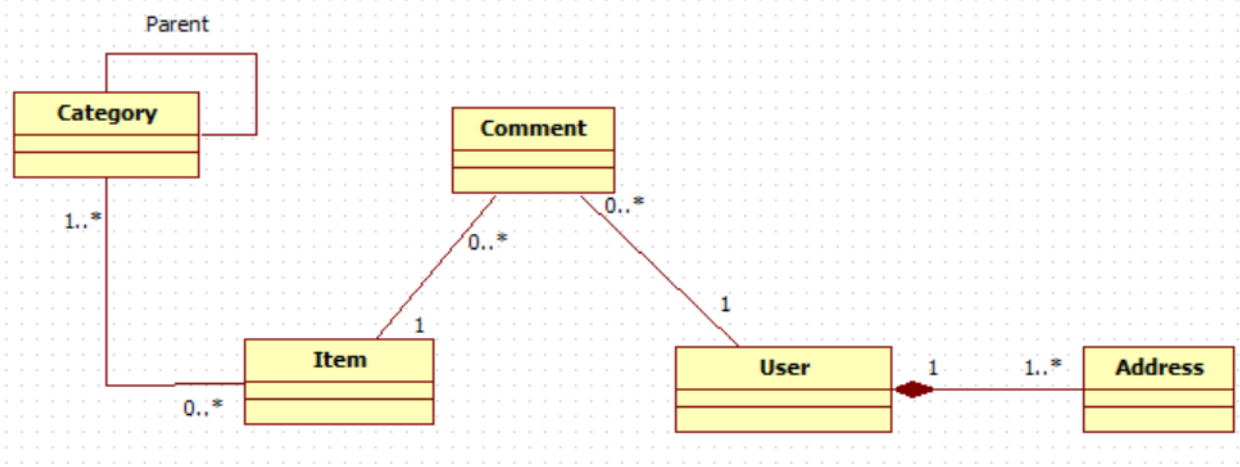
ORM Entity Lifecycle

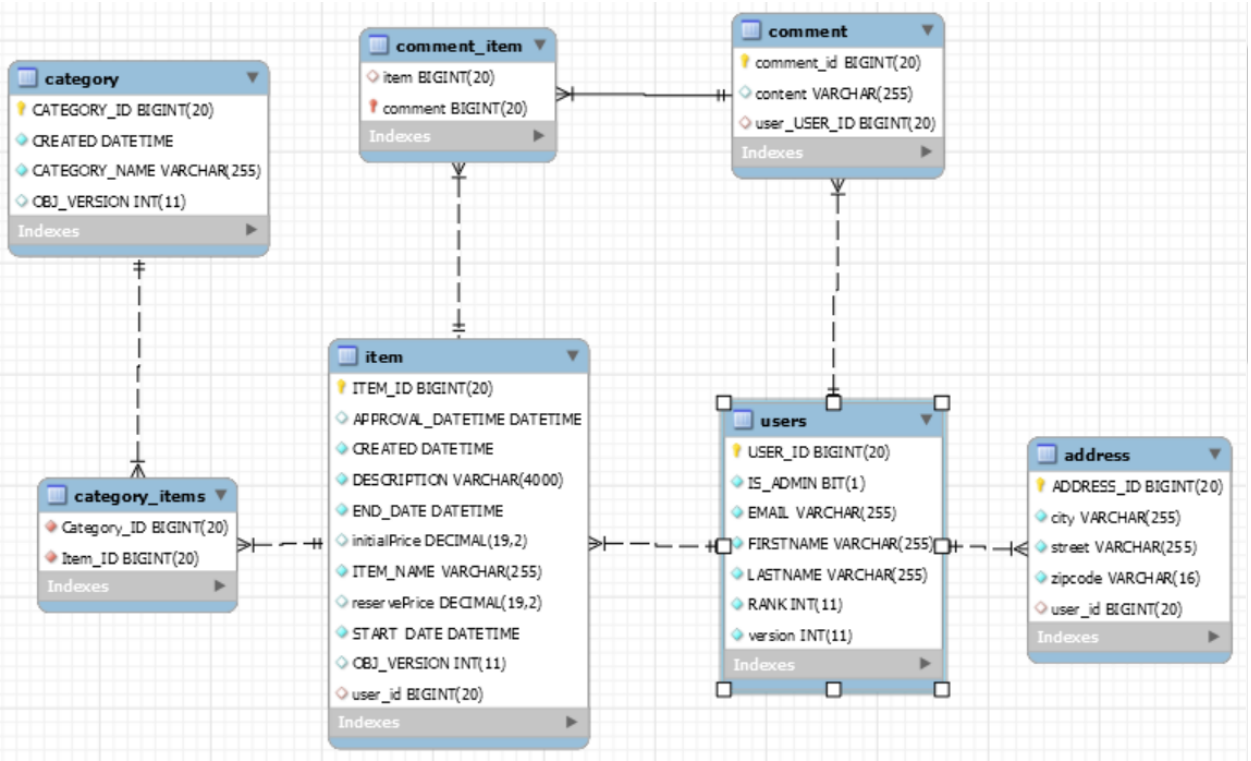


ORM-related Entity States

- ***Transient –***
- it has just been instantiated using the new operator
- not associated with a Session
- no persistent representation in the database
- ***Persistent –***
- representation in the database
- Has been saved or loaded in Session
- Changes made to an object are synchronized with the database when the unit of work completes..
- ***Detached –***
- An object that has been persistent, but Session has been closed
- ***Removed –***
- An object is deleted from the database when the unit of work completes

4. [20 points] Annotate the Domain Objects based on the Domain Model and Entity Relationship Diagram provided. NOTE: All the Domain Objects are not listed. All the fields are not listed. Only annotate the objects and fields that are listed.





```java
@Entity
@Table(name = "USERS")
public class User implements Serializable  {

    @Id @GeneratedValue(strategy=GenerationType.AUTO)
    @Column(name = "USER_ID")
    private Long id = null;

    @Version
    private int version = 0;


     @Column(name = "FIRSTNAME", nullable = false)
    private String firstName;

    @Column(name = "LASTNAME", nullable = false)
    private String lastName;

    @Column(name = "EMAIL", nullable = false)
    private String email;

    @Column(name = "RANK", nullable = false)
    private int ranking = 0;

    @Column(name = "IS_ADMIN", nullable = false)
    private boolean admin = false;


    @OneToMany(fetch=FetchType.LAZY, cascade = CascadeType.PERSIST, mappedBy="user")
     List<Comment> comments;

    @OneToMany(mappedBy="user", fetch=FetchType.LAZY, cascade = CascadeType.PERSIST)
    List<Address> addresses;

```

```java
@Entity
public class Comment {
    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    @Column(name="comment_id")
    private long id;
     @JoinColumn
    @ManyToOne(fetch=FetchType.EAGER)
    private User user;


    @ManyToOne(fetch=FetchType.EAGER, cascade = {CascadeType.PERSIST, CascadeType.MERGE})
    @JoinTable ( name="comment_item", joinColumns={@JoinColumn(name="comment")},
    inverseJoinColumns={ @JoinColumn(name="item")} )
    private Item item;

    private String content;

```

```java
@Entity
@Table(name = "ITEM")
 public class Item implements Serializable {

     @Id @GeneratedValue
    @Column(name = "ITEM_ID")
    private Long id = null;

    @Version
    @Column(name = "OBJ_VERSION")
    private int version = 0;

    @Column(name = "ITEM_NAME", length = 255, nullable = false, updatable = false)
    private String name;

    @Column(name = "DESCRIPTION", length = 4000, nullable = false)
    private String description = "";

    private BigDecimal reservePrice;

    @ManyToMany(fetch = FetchType.EAGER, cascade= {CascadeType.PERSIST,CascadeType.MERGE}, mappedBy="items")
    private Set<Category> categories = new HashSet<Category>();

    @OneToMany( mappedBy= "item", fetch=FetchType.EAGER, cascade = CascadeType.PERSIST)
    List<Comment> comments;
```

```java
 @Entity
@Table(
   name = "CATEGORY")
public class Category implements Serializable {

    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    @Column(name = "CATEGORY_ID")
    private Long id = null;

    @Version
    @Column(name = "OBJ_VERSION")
    private int version = 0;

    @Column(name = "CATEGORY_NAME", length = 255, nullable = false)
    private String name;

    @ManyToMany(fetch = FetchType.EAGER, cascade = CascadeType.ALL)
    @JoinTable ( name="Category_Items", joinColumns={@JoinColumn(name="Category_ID")},
    inverseJoinColumns={ @JoinColumn(name="Item_ID")} )
    private List<Item> items = new ArrayList<Item>();


```

5. [15 points] Explain the concept of locking. Include the definition of the two strategies covered in class. Give the details of Version-Based Optimistic Concurrency [Locking], how it relates to isolation levels, how it is implemented in JPA.

1

Lock Mode for Data Consistency

- Locking refers to actions taken to prevent data in a relational database from changing between the time it is read and the time that it is used.

Locking strategies

- **Optimistic Lock**
- **Concurrent transactions can complete without affecting each other,**
- **Transactions do not need to lock the data resources that they affect.**

- **Pessimistic Lock**
- Concurrent transactions will conflict with each other
- Transactions require that data is locked when read and unlocked at commit.

14

Version-Based Optimistic Concurrency[Locking]

- High-volume systems
- No Connection maintained to the Database [Detached Objects]
- Effective in ***Read-Often Write-Sometimes*** scenario

- Uses **read committed isolation level**
  Guarantees **repeatable read isolation level**
  An exception is thrown if a conflict occurs

- **@Version Long version;**
- When entity is updated - version field is incremented.

15

Version-Based Optimistic Concurrency

- Optimistic concurrency assumes that lost update conflicts generally don't occur
  - Keeps version #s so that it knows when they do
  - Guarantees best performance and scalability
  - The default way to deal with concurrency
  - There is no locking anywhere
  - It works well with very long conversations, including those that span multiple transactions
- First commit wins instead of last commit wins
  - An exception is thrown if a conflict would occur
    - ObjectOptimisticLockingFailureException

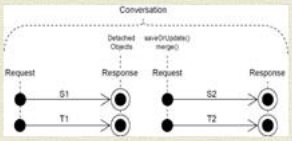
16

Optimistic Locking [Cont.]

- It works well with long conversations, including those that span multiple transactions

- **LONG CONVERSATION Use Case:**
  A multi-step dialog, for example a wizard dialog interacts with the user in several request/response cycles.

- ***session-per-request-with-detached-objects***
-
- @Version Locking manages
- Detached object consistency



17

JPA Optimistic Locking

JPA version-based concurrency control

- ***Simply add @Version to an Integer field – JPA takes care of the rest***

```
@Id
 @GeneratedValue(strategy = GenerationType.AUTO)
   private Long id = null;
   @Version
   @Column(name = "version")
   private int version = 0;
Hibernate:
    update purchaseOrder
    set orderNumber=?,version=?
    where id=? and version=?
```

@Version property is incremented on every DB write

Checked for consistency on every update

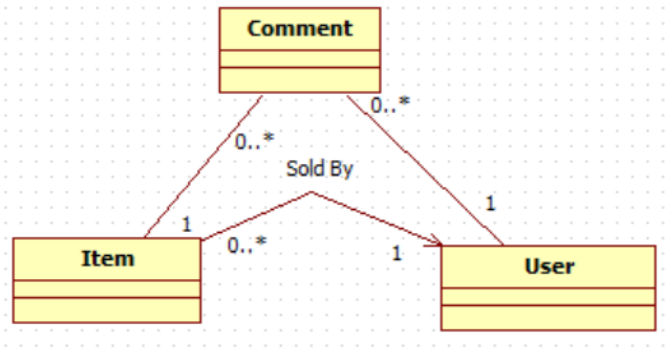
6. [15 points] Implement a JQPL **parameterized** query that looks up a User **by email** who is selling a specific Item with an initial price greater than a specified dollar value.
   For example:

Find John Smith who is selling an Item, "cardboard box" with an initial price greater than 70.00.

Another example:

Find Will Henry who is selling an Item named "Pencil Set" with an initial price greater than 100.00.

Item – User relationship [See relevant class properties in previous problem]:




```
In Item.java:
        @ManyToOne(fetch=FetchType.LAZY)
        @JoinColumn (name="user_id")
        private User seller;
```

Remember the Query is a ***parameterized query***. Also identify all the classes in the specific packages that need to be modified to implement the query in accordance with the N-Tier architecture convention. Describe the "pattern" that exists at the persistence layer.

ANSWER:

edu.mum.dao. UserDao

public User findBySoldItemInitialPrice(String email,String itemName, `BigDecimal initialPrice`);

edu.mum.dao.impl. UserDaoImpl

```
public User findBySoldItemInitialPrice(String email, String boughtItem, BigDecimal initialPrice) {
  Query query=entityManager.createQuery("select u from User u, Item i where i.seller.email=:email"
                          + "and i.name = :boughtItem and i.initialPrice > :initialPrice");
  return (User) query.setParameter("boughtItem", boughtItem).
       setParameter("email", email). setParameter("initialPrice", initialPrice).getSingleResult();

}
```

edu.mum.service. UserService

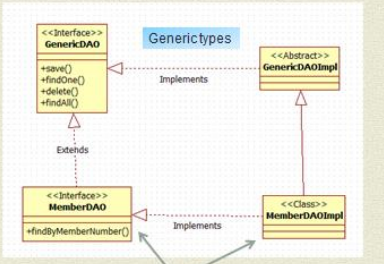
public User findBySoldItemInitialPrice(String email,String itemName, `BigDecimal initialPrice`);

edu.mum.service.impl. UserServiceImpl

public User findBySoldItemInitialPrice(String email,String itemName, `BigDecimal initialPrice`) {
return userDao.findBySoldItemInitialPrice(email,itemName, `BigDecimal initialPrice`);

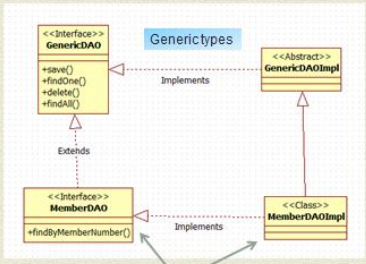
}

"Classic" ORM GenericDAO



1

"Classic" ORM GenericDAO



Generic DAO Implementation

```
public abstract class GenericDaoImpl<T> implements GenericDao<T> {

@PersistenceContext
    protected EntityManager entityManager;
    protected Class<T> daoType;

     public void setDaoType(Class<T> type) {
            daoType = type;
     }
    @Override
    public void save( T entity ){
        entityManager.persist( entity );
     }
    public void delete( T entity ){
        entityManager.remove( entity );
```


Domain Class specific DAO

```
public interface MemberDao extends GenericDao<Member> {
    public Member findByMemberNumber(Integer number);

public class MemberDaoImpl extends GenericDaoImpl<Member> implements MemberDao
public MemberDaoImpl() {
super.setDaoType(Member.class);
}
 public Member findByMemberNumber(Integer number) {
   Query query = entityManager.createQuery("select m from MEMBER m
                                where m.memberNumber =:number");
  return (Member) query.setParameter("number", number).getSingleResult();
}
```

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.**

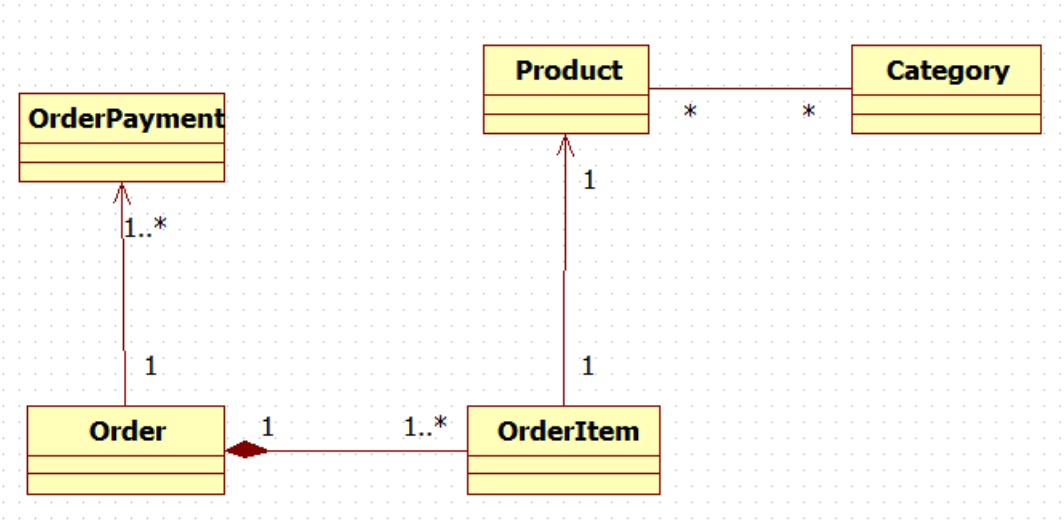
   T **F** Spring @Transaction has no built-in metadata for managing any of the DB ACID properties.

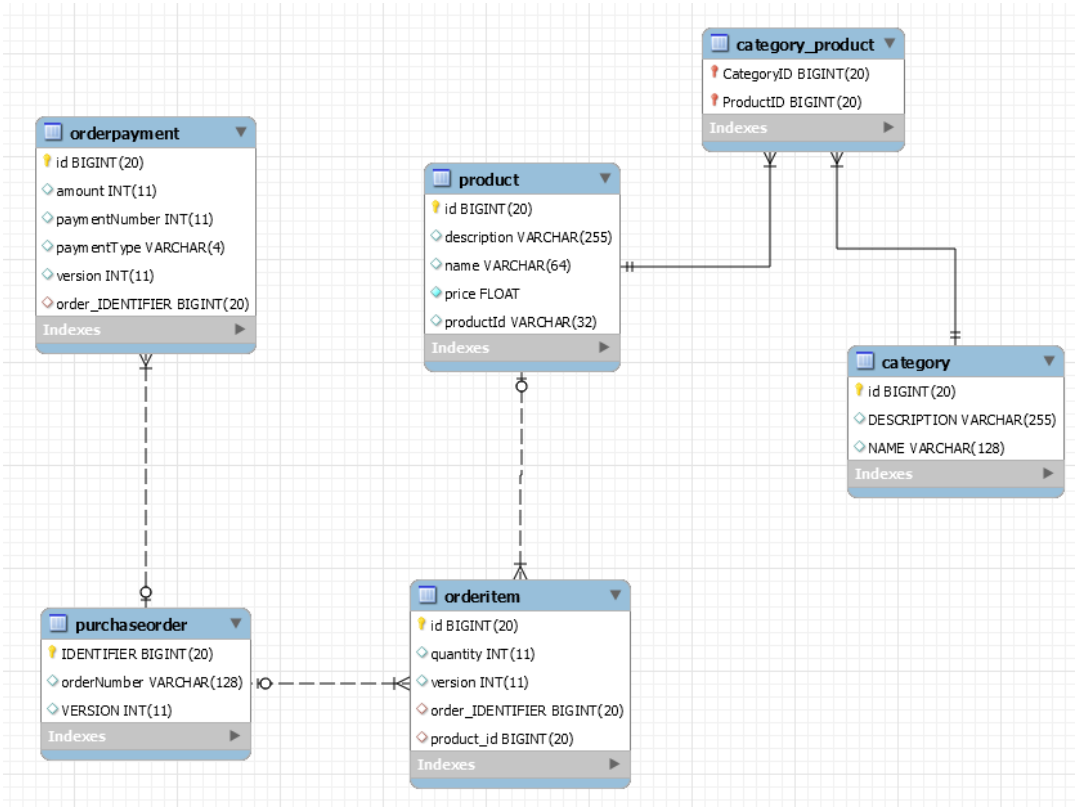
   EXPLAIN:____ **@Transactional has an optional parameter for indicating isolation levels. For example** - **@Transactional (*isolation=Isolation.READ_COMMITTED)***

   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

```
20 @Entity
21 public class Product implements Serializable {
22     private static final long serialVersionUID = 5784L;
23 
24     @Id
25     @GeneratedValue(strategy=GenerationType.AUTO)
26      private long id;
27     @Column(length=64)
28     private String name;
29     @Column
30     private String description;
31     @Column(length=32)
32     private String productId;
33     @Column
34     private float price;
35 
36     @ManyToMany(mappedBy="products",fetch = FetchType.EAGER, cascade = CascadeType.ALL)
37     private Set<Category> categories;
```

Category.java

```
18 @Entity
19 public class Category {
20 
21     @Id
22     @GeneratedValue(strategy=GenerationType.AUTO)
23     private long id;
24 
25     @Column(name="NAME",length=128)
26     String name;
27 
28     @Column(name="DESCRIPTION")
29     String description;
30 
31     // If using a List INSTEAD of a SET - less efficient
32     @ManyToMany(fetch = FetchType.EAGER, cascade = CascadeType.ALL)
33     @JoinTable ( name="Category_Product", joinColumns={@JoinColumn(name="CategoryID")},
34     inverseJoinColumns={ @JoinColumn(name="ProductID")} )
35     Set<Product> products = new HashSet<Product>();
```

Order.java

```
21 @Entity
22 @Table(name = "purchaseOrder")
23  public class Order {
24      @Id
25       @GeneratedValue(strategy = GenerationType.AUTO)
26       @Column(name = "IDENTIFIER", updatable = false, nullable = false)
27       private Long id = null;
28       @Version
29       @Column(name = "VERSION")
30       private int version = 0;
31 
32       @Column(length=128)
33       private String orderNumber;
34 
35       @OneToMany(mappedBy = "order", fetch = FetchType.LAZY, cascade = { CascadeType.PERSIST, CascadeType.MERGE })
36       private Set<OrderItem> items = new HashSet<OrderItem>();
37 
38       //mappedBy = "order",
39       @OneToMany( fetch = FetchType.LAZY, cascade = { CascadeType.PERSIST, CascadeType.MERGE })
40       @JoinColumn(name="order_IDENTIFIER")
41       private Set<OrderPayment> payments = new HashSet<OrderPayment>();
```

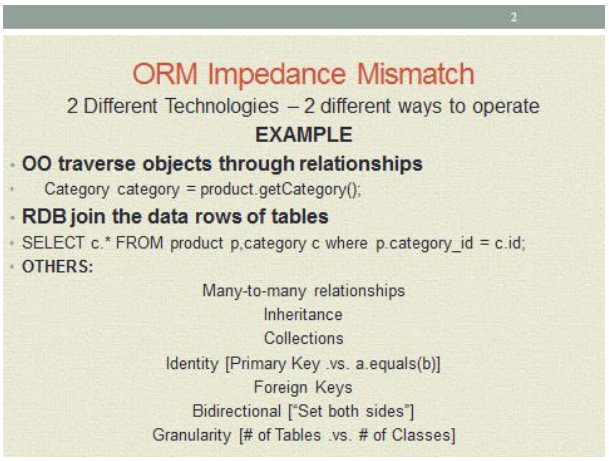
OrderItem.java

```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;
```

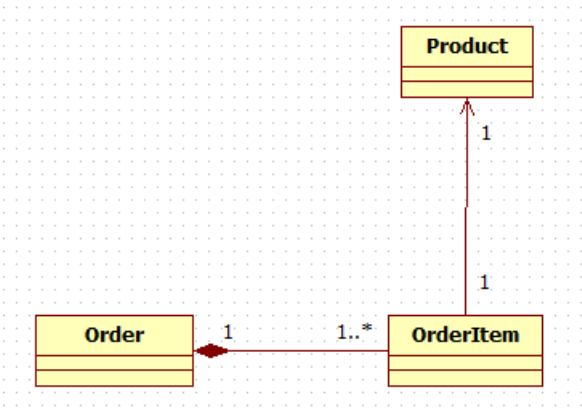
OrderPayment.java

```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.



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



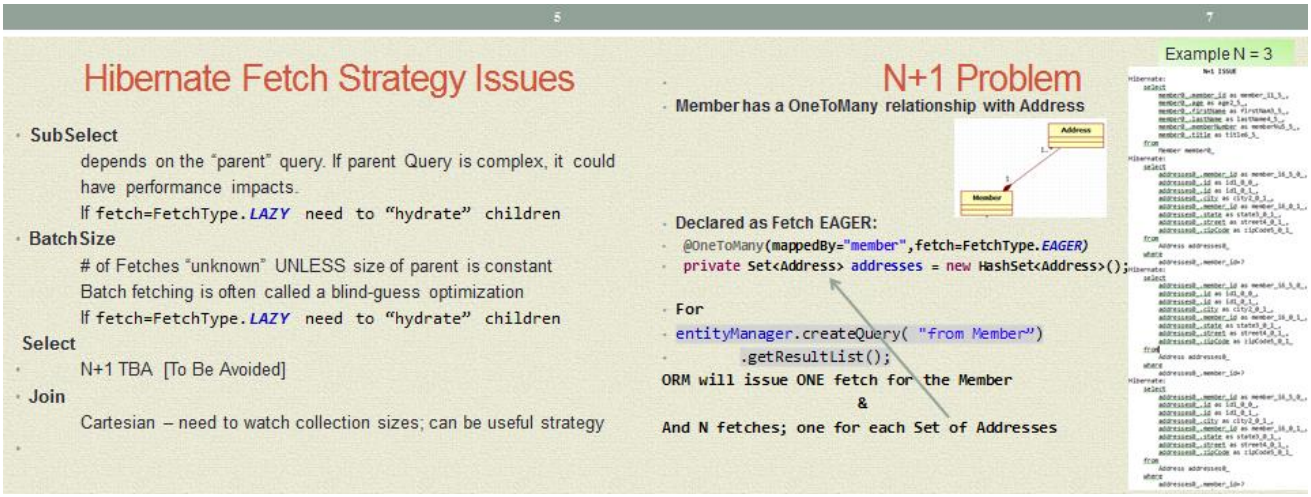
What performance problem does the Join fetch address? Give details.
What performance problem does it cause? Give details.
What can be done to "clean up" the data returned by the fetch?

In OrderDaoImpl.Java

```
23    public List<Order> findAllJoinFetch() {
24          Query query =  entityManager.createQuery("SELECT   o FROM Order AS o JOIN FETCH o.items AS i");
25          List<Order> orders =  query.getResultList();
26          return orders;
27    }
```

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.



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

"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]

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

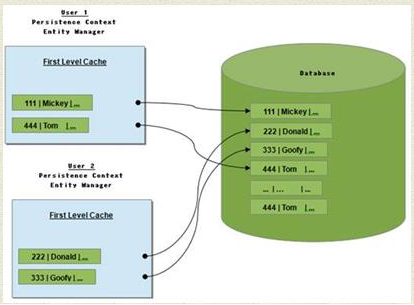
ORM caching mechanisms

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.

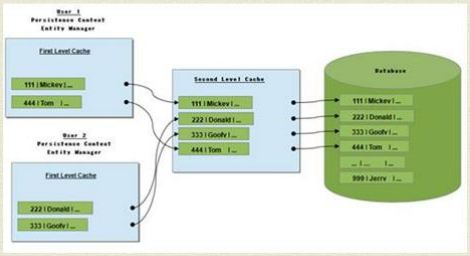
22

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

15

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");
```

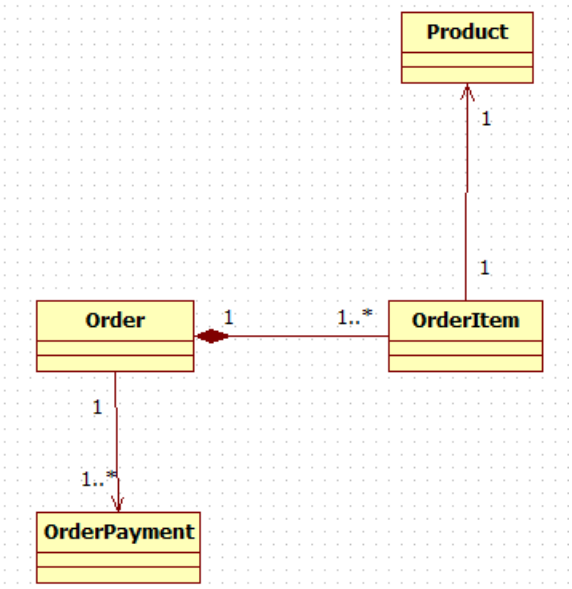
- **Query Cache**
  Applicable for accessing entities by specific query
  The two preceding queries are Query Cache "candidates"

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

```
52      public List<Product> findByAmountRangeAndQuantity(Integer minPayment,Integer maxPayment,Integer quantity) {
53          Query query = entityManager.createQuery("select p from Product p,OrderItem oi,OrderPayment op where oi.product =p and "
54                  + " oi.quantity > :quantity and op member of oi.order.payments  "
55                  + " and op.amount > :minPayment and op.amount < :maxPayment)") ;
56
57          return (List<Product>) query.setParameter("maxPayment", maxPayment)
58                  .setParameter("minPayment", minPayment)
59                  .setParameter("quantity", quantity)
60                  .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);
    }
```

```
52          public List<Product> findByAmountRangeAndQuantity(Integer minPayment,Integer maxPayment,Integer quantity) {
53              Query query = entityManager.createQuery("select p from Product p,OrderItem oi,OrderPayment op where oi.product =p and "
54                      + " oi.quantity > :quantity and op member of oi.order.payments  "
55                      + " and op.amount > :minPayment and op.amount < :maxPayment)") ;
56
57              return (List<Product>) query.setParameter("maxPayment", maxPayment)
58                      .setParameter("minPayment", minPayment)
59                      .setParameter("quantity", quantity)
60                      .getResultList();
61
62              /*                  + " oi.quantity > :quantity and op in (select opay from oi.order.payments opay "
63              + " where  opay.amount > :minPayment and opay.amount < :maxPayment)") ;
64  */
65  //This works also
66  //          + " oi.quantity > :quantity and op in (select opay from oi.order.payments opay ) "
67  //          + " and  op.amount > :minPayment and op.amount < :maxPayment");
68
```

CS544
Enterprise Architecture
Midterm December 2016

Name_____

Student ID _____

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

1. [10 points] **Circle w**hich of the following is TRUE/FALSE concerning Spring Inversion of Control/Dependency Injection:

   T **F** Only Managed Beans can be injected in Spring, a POJO or JavaBean cannot.

   EXPLAIN:____If a POJO or JavaBean can be a Spring Managed bean so they can be injected.

   T **F** @Autowired works only on interfaces. It cannot work directly on classes.

   EXPLAIN:__ It can work on classes. However you lose some of the value, testing; changing implementations

   T F In practice, IoC container is not exactly the same as Dependency Injection as it involves a discovery step concerning the dependency.

   EXPLAIN:__ IoC involves a "look up the dependency" step before injecting it. Spring has declarative configuration that identifies "where to find" the resource to inject.

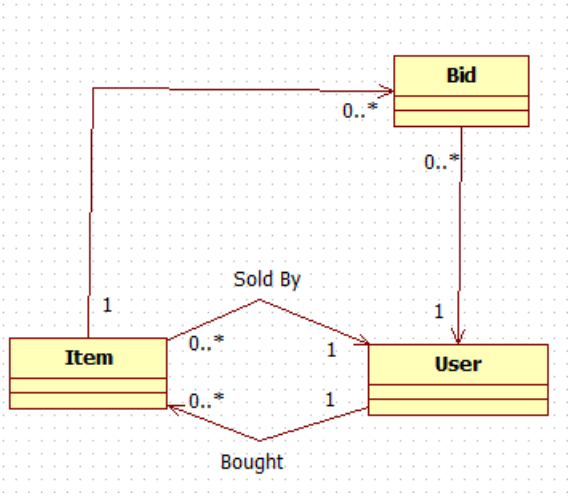
   T **F** A domain object needed in a @Service class is usually a good candidate for Dependency Injection

   EXPLAIN:____ DI is mainly used for "cross" layer transitions, access plumbing resources, etc. NOT for passing business data around…

   T **F** In Spring, DI can be done through either XML or through Annotations. They are mutually exclusive. That means, if you use XML for DI for one bean, they you should use it, exclusively for all beans.

   EXPLAIN:____ You could inject a DAO[memberDAO in memberService] through XML & inject another DAO[productDAO in productService] through @Autowired. Remember, XML configuration is the final word – if you configure the same bean twice, the XML confguration will take precedence.

2. [15 points] For the following relationships implement a SubSelect that fetches all items with their corresponding collection of bids.



What performance problem[s] does the SubSelect fetch address?
How does it work? – Explain the "algorithm" based on a universe of 10 Items each with a collection of 5-10 Bids.
Compare it to Join Fetch.

In Item.Java

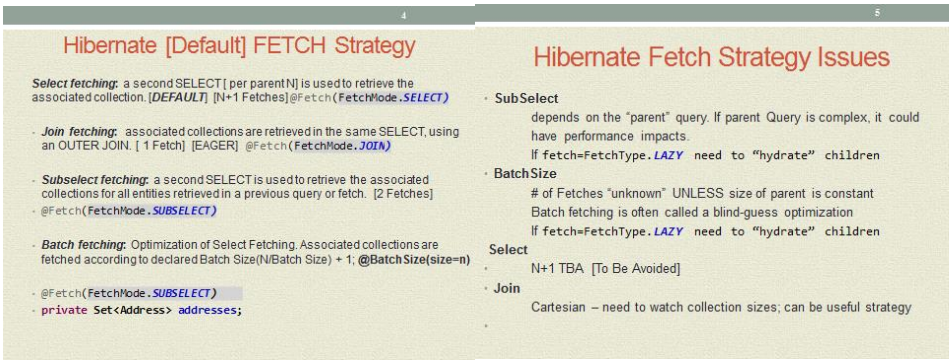
```
public class Item {

	@OneToMany(cascade = CascadeType.ALL, fetch = FetchType.LAZY)
	@Fetch(FetchMode.SUBSELECT)
	 private List<Bid> bids = new ArrayList<Bid>();
```

In ItemServiceImpl.Java

```
	public List<Item> findbySubSelect() {

		List<Item> items = (List<Item>)this.findAll();
		// hydrate since LAZY load
		items.get(0).getBids().get(0);

		return items;
	}
```

Subselect does ONE fetch for the Items and ONE fetch for ALL collections. Therefore the number of Items & Bids in the collections does not matter,
It solves the Cartesian issue && the N+1 issue.
If the collection is LAZY loaded, then doing a get all for the "parent" entities [Item as in above] will get the "parent"/Item list. Within the same transaction, a reference to ONE of the Bid collections will fetch ALL the collections.

The Join Fetch, on the other hand will get ALL the Bids AND Items in ONE Select/fetch.
The Join Fetch, however suffers from the Cartesian product issue [ItemsXBids] which means that more than one copy of each Item/Bid pair will be included in the fetch. The number of copies depends on the number of Bids in the collection. So if Item A has 3 bids, 3 copies of Item A will be present.



3. [15 points] The Core J2EE DAO pattern is fundamental to a well-organized ORM application.
   Explain the pattern, what is for, how it works. Include in the explanation, the sequence of interactions between the user, a DAO and the database. Include an explanation [& UML diagram] of the Generic DAO design.
   Be specific, give examples.

11

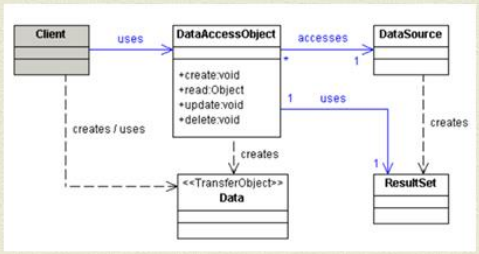
CRUD Services
Core J2EE DAO pattern

Data Access Object (DAO)

- Manage the connection with the data source
- Abstract and encapsulate access to data source
- Provides CRUD access:
  - Create
  - Read
  - Update
  - Delete
- Hides the data source implementation details
- Interface allows for different storage schemes
- Adapter between the client and the data source.

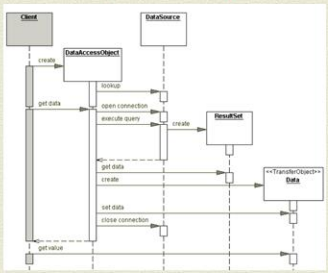
12

DAO Class Diagram



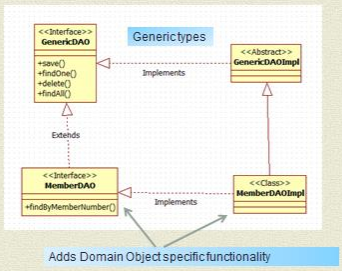
1

DAO Sequence Diagram



1

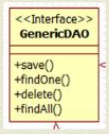
"Classic" ORM GenericDAO



15

Generic DAO Interface

```
public interface GenericDao<T> {
    void save(T t);
    void delete(Long id);
    T findOne(Long id);
    T update(T t);
    List<T> findAll();
```



16

Generic DAO Implementation

```
public abstract class GenericDaoImpl<T> implements GenericDao<T> {

@PersistenceContext
    protected EntityManager entityManager;
    protected Class<T> daoType;

     public void setDaoType(Class<T> type) {
           daoType = type;
     }
    @Override
    public void save( T entity ){
        entityManager.persist( entity );
     }
    public void delete( T entity ){
        entityManager.remove( entity );
```

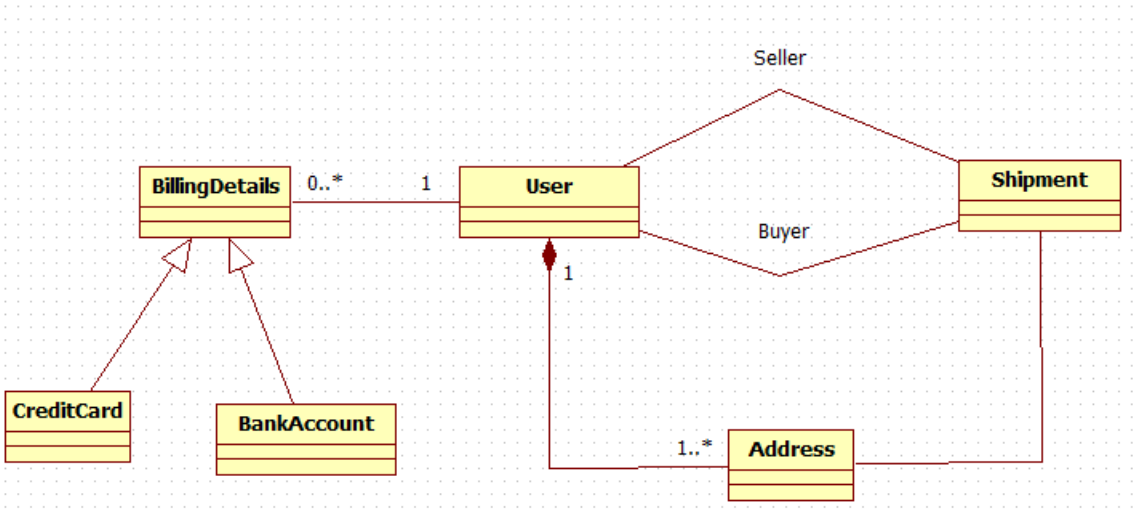
17

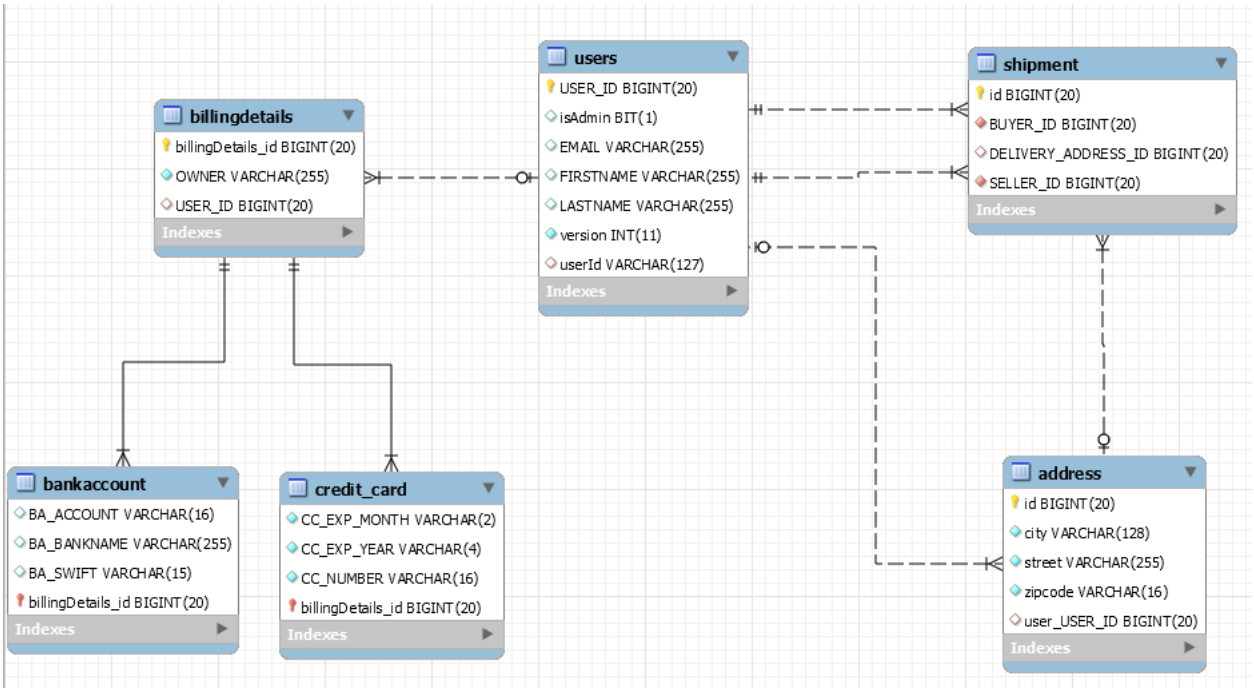
Domain Class specific DAO

```
public interface MemberDao extends GenericDao<Member> {
    public Member findByMemberNumber(Integer number);
```


```
public class MemberDaoImpl extends GenericDaoImpl<Member> implements MemberDao
public MemberDaoImpl() {
super.setDaoType(Member.class);
}
 public Member findByMemberNumber(Integer number) {
   Query query = entityManager.createQuery("select m from MEMBER m
                              where m.memberNumber =:number");
   return (Member) query.setParameter("number", number).getSingleResult();
}
```

4. [20 points] Annotate the Domain Objects based on the Domain Model and Entity Relationship Diagram provided. NOTE: All the Domain Objects are not listed. All the fields are not listed. Only annotate the objects and fields that are listed.





```java
@Entity
@Table(name = "USERS")
 public class User implements Serializable  {

    @Id @GeneratedValue(strategy=GenerationType.AUTO)
    @Column(name = "USER_ID")
    private Long id = null;

    @Column(name = "FIRSTNAME")
    private String firstName;

    @Column(name = "LASTNAME")
    private String lastName;

    @Column(name = "EMAIL")
    private String email;

     @Column(name = "isAdmin")
    private boolean admin = false;

    @OneToOne(fetch=FetchType.EAGER,  cascade = CascadeType.ALL)
    @JoinColumn(name="userId")
    private UserCredentials userCredentials;

       @OneToMany(fetch=FetchType.LAZY, cascade = CascadeType.PERSIST, mappedBy="user")
         private Set<Address> addresses = new HashSet<Address>();

       @OneToMany(fetch=FetchType.LAZY, cascade = CascadeType.PERSIST, mappedBy="buyer")
         private Set<Shipment> buyShipments = new HashSet<Shipment>();

       @OneToMany(fetch=FetchType.LAZY, cascade = CascadeType.PERSIST, mappedBy="seller")
         private Set<Shipment> sellShipments = new HashSet<Shipment>();

       @OneToMany(fetch=FetchType.LAZY, cascade = CascadeType.PERSIST, mappedBy="user")
      private Set<BillingDetails> billingDetails = new HashSet<BillingDetails>();
```

```java
 @Entity
@Table(name = "SHIPMENT")
public class Shipment {

    @Id @GeneratedValue
    private Long id = null;

    @ManyToOne(fetch = FetchType.EAGER)
    @JoinColumn(name="DELIVERY_ADDRESS_ID")
     private Address deliveryAddress;

     @ManyToOne(fetch = FetchType.EAGER)
    @JoinColumn(name="BUYER_ID")
    private User buyer;

    @ManyToOne(fetch = FetchType.EAGER)
    @JoinColumn(name="SELLER_ID")
     private User seller;
```

```java
@Entity
public class Address implements Serializable {

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

    @Column(length = 255)
    private String street;

    @Column(length = 16)
    private String zipcode;

    @Column(length = 128)
    private String city;

    @ManyToOne(fetch=FetchType.LAZY)
    private User  user;
```

```java
@Entity
@Table(name = "billingDetails")
@Inheritance(strategy = InheritanceType.JOINED)
public abstract class BillingDetails  {

    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    @Column(name = "billingDetails_id")
    private Long id = null;


    @Column(name = "OWNER")
    private String owner;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "USER_ID")
     private User user;
```

```java
@Entity
public class BankAccount extends BillingDetails {

    // Id Shared with Superclass - BillngDetails

    @Column(name = "BA_ACCOUNT",length = 16)
    private String account;

    @Column(name = "BA_BANKNAME",  length = 255)
    private String bankname;

    @Column(name = "BA_SWIFT", length = 15)
    private String swift;
```

5. [10 points] Explain the concept of locking. Include the definition of the two strategies covered in class. Give the details of Version-Based Optimistic Concurrency [Locking], how it relates to isolation levels, how it is implemented in JPA.

1

Lock Mode for Data Consistency

- Locking refers to actions taken to prevent data in a relational database from changing between the time it is read and the time that it is used.

Locking strategies

- **Optimistic Lock**
- **Concurrent transactions can complete without affecting each other,**
- **Transactions do not need to lock the data resources that they affect.**

- **Pessimistic Lock**
- Concurrent transactions will conflict with each other
- Transactions require that data is locked when read and unlocked at commit.

14

Version-Based Optimistic Concurrency[Locking]

- High-volume systems
- No Connection maintained to the Database [Detached Objects]
- Effective in ***Read-Often Write-Sometimes*** scenario

- Uses **read committed isolation level**
  Guarantees **repeatable read isolation level**
  An exception is thrown if a conflict occurs

- **@Version Long version;**
- When entity is updated - version field is incremented.

15

Version-Based Optimistic Concurrency

- Optimistic concurrency assumes that lost update conflicts generally don't occur
  - Keeps version #s so that it knows when they do
  - Guarantees best performance and scalability
  - The default way to deal with concurrency
  - There is no locking anywhere
  - It works well with very long conversations, including those that span multiple transactions
- First commit wins instead of last commit wins
  - An exception is thrown if a conflict would occur
    - ObjectOptimisticLockingFailureException

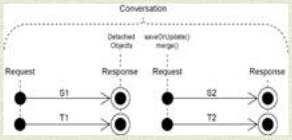
16

Optimistic Locking [Cont.]

- It works well with long conversations, including those that span multiple transactions

- **LONG CONVERSATION Use Case:**
  A multi-step dialog, for example a wizard dialog interacts with the user in several request/response cycles.

- ***session-per-request-with-detached-objects***
-
- @Version Locking manages
- Detached object consistency



17

JPA Optimistic Locking

JPA version-based concurrency control

- *Simply add @Version to an Integer field – JPA takes care of the rest*

```
@Id
 @GeneratedValue(strategy = GenerationType.AUTO)
   private Long id = null;
   @Version
   @Column(name = "version")
   private int version = 0;
Hibernate:
    update purchaseOrder
    set orderNumber=?,version=?
    where id=? and version=?
```

@Version property is incremented on every DB write

Checked for consistency on every update

6. [15 points] Implement a JQPL query that looks up a User by email who bought an Item with a shipping address that has a specific zip code. [Reference UML in problem #4]
   For instance:

Find User who has an email address of JohnDoe@mail.com who bought an Item that was shipped to a zip code equal to 52556

OR

Find User who has an email address of JBean@post.com who bought an Item that was shipped to a zip code equal to 12345.

The Query should be a parameterized query. Also identify all the classes in the specific packages that need to be modified to adhere to the N-Tier architecture convention.

ANSWER:

edu.mum.dao. UserDao

public User findByBoughtItemShippedZip(String email,String zipCode);

edu.mum.dao.impl. UserDaoImpl

```
public User findByBoughtItemShippedZip(String email,String zipCode) {

        Query query = entityManager.createQuery("select u from User u,Shipment s
                     where u.email =:email and s.buyer = u "
                     + " and s.deliveryAddress.zipcode = :zipCode");
        return (User) query.setParameter("email", email)
                .setParameter("zipCode", zipCode).getSingleResult();
}
```

edu.mum.service. UserService

public User findByBoughtItemShippedZip(String email,String zipCode);

edu.mum.service.impl. UserServiceImpl

```
public User findByBoughtItemShippedZip(String email,String zipCode) {
    return  userDao. findByBoughtItemShippedZip (email,zipCode);

}
```

CS544
Enterprise Architecture
Midterm July 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 ORM technologies:

   T F An example of impedance mismatch is the fact that a RDB puts information in rows and an OO language puts information in Objects

   EXPLAIN:____A RDB consists of individual columns which represent the fields/properties of an object. This requires some way to BIND the DB data to the Object & make sure that the data types match.

   T F A good use case for using an ORM is complex interactions between entities

   EXPLAIN:__ This is a major advantage of an ORM. The ORM "automatically" maps these relationships, reducing boiler plate code.

   T **F** The value of a good ORM is that it automatically takes care of all the issues relating to a RDB

   EXPLAIN:__It is NOT possible for an ORM to do everything. It covers mapping and CRUD services. However there are situations where "manual" intervention is necessary

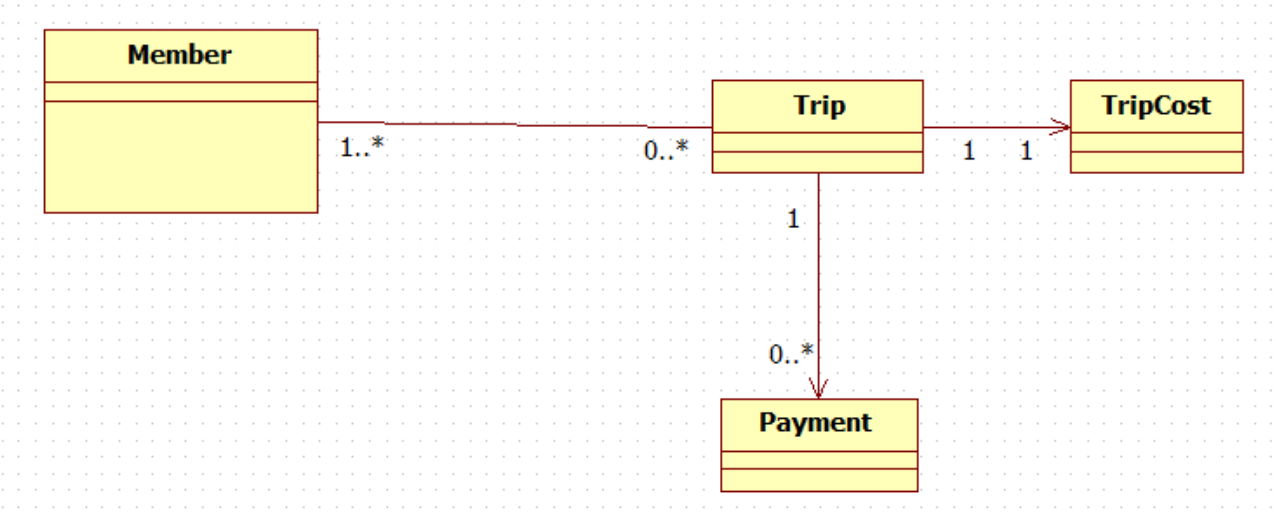
   T **F** JPA is an industry standard for ORMs and has made Hibernate obsolete.

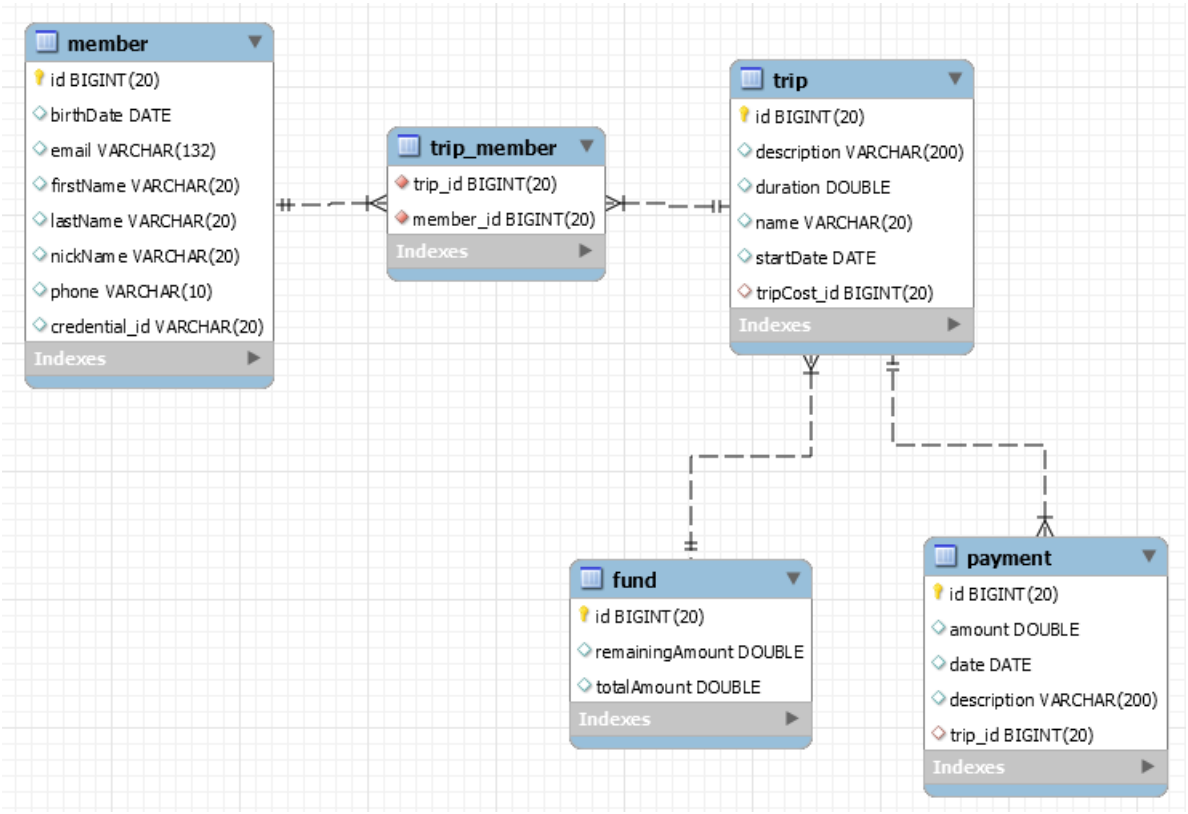
   EXPLAIN:____JPA is an industry standard but does not replace Hibernate. In fact, Hibernate implements JPA As JPA is "only" an API.

   T **F** Native SQL Queries are supported by a good ORM solution and are recommended as the first choice way to access entity relationships.

   EXPLAIN:____Native queries are fallback mechanism to be used when queries are complex and cannot be adequately implemented in JPQL.

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.





Trip.java

```java
@Entity
public class Trip {

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

    @Column(length = 20)
    private String name;

    @Column(length = 200)
    private String description;

    @Column
    private Double duration;

     @Temporal(TemporalType.DATE)
     private Date startDate;

    @Transient
    private Date endDate;

    @ManyToMany(cascade = { CascadeType.PERSIST, CascadeType.MERGE })
    @JoinTable(name = "trip_member", joinColumns = { @JoinColumn(name = "trip_id") }, inverseJoinColumns = {
            @JoinColumn(name = "member_id") })
    List<Member> members = new ArrayList<>();

    @OneToOne(fetch = FetchType.LAZY, cascade = { CascadeType.PERSIST, CascadeType.MERGE, CascadeType.REMOVE })
    @JoinColumn(name="fund_id")
    Fund fund;

    @OneToMany(fetch = FetchType.EAGER, cascade = { CascadeType.PERSIST, CascadeType.MERGE })
    @JoinColumn(name="trip_id")
    List<Payment> payments = new ArrayList<>();
```

TripCost.java

```java
@Entity(name="TripCost")
public class Fund {

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

    private Double remainingAmount;

    private Double totalAmount;
```

Payment.java

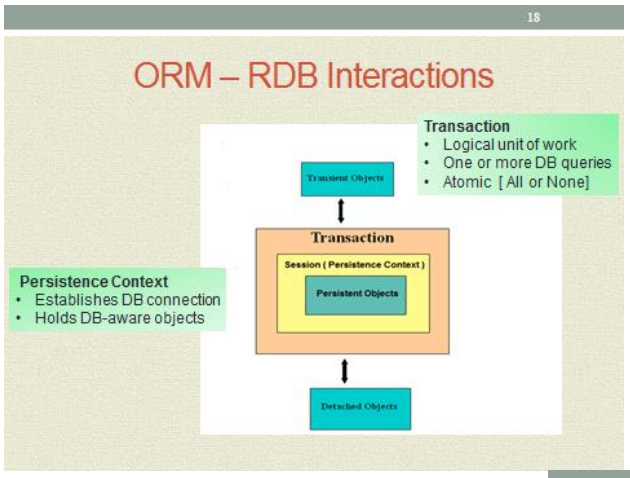
```java
@Entity
public class Payment {

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

    @Column(length = 200)
    private String description;

    @Column
    private Double amount;

    @Temporal(TemporalType.DATE)
    private Date date;
```

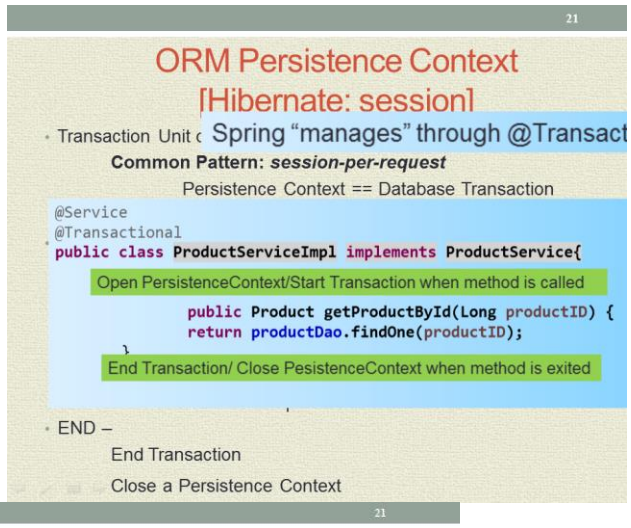
Member.java

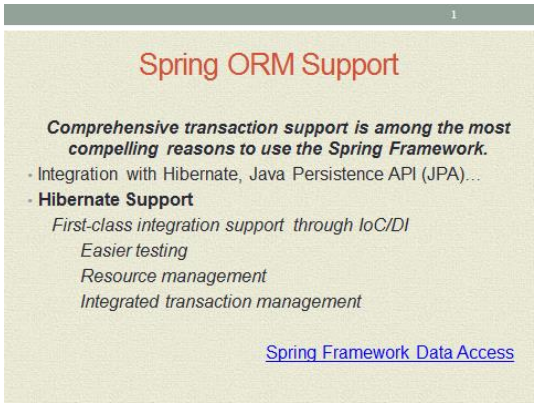
```java
@Entity
public class Member {

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

    @Column(length = 20)
    private String firstName;

    @Column(length = 20)
    private String lastName;

    @Column(length = 20)
    private String nickName;

    @Transient
    private Gender gender;

    @Column(length = 132)
    private String email;

    @Column(length = 10)
    private String phone;

    @Temporal(TemporalType.DATE)
    private Date birthDate;

     @ManyToMany( cascade= {CascadeType.PERSIST,CascadeType.MERGE}, mappedBy="members")
    List<Trip> trips;
```

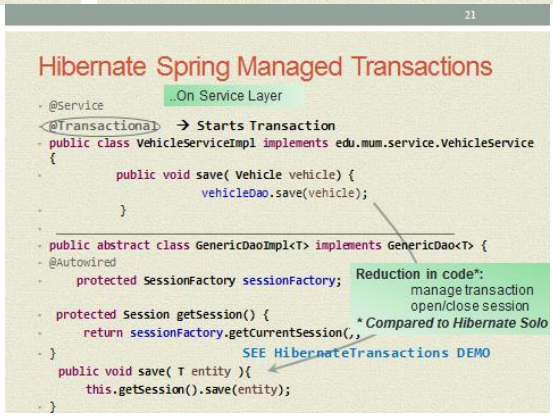
3. [15 points] Transaction management is an important part of RDBMS oriented enterprise applications Spring provides core functionality to assist in transaction management. Describe the Spring transaction functionality, how it is implemented, how it facilitates ORM Transaction management. Include an explanation on how it supports the RDBMS ACID properties of Consistency and Isolation. Be specific. Give Examples.

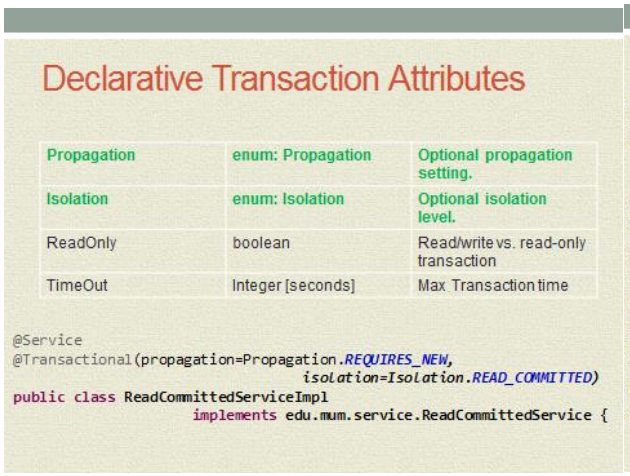
ANSWER:

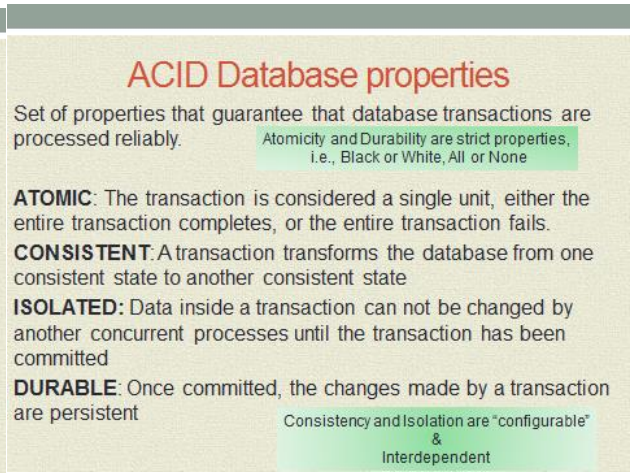












Isolation in a Relational Database

- The challenge is to maximize concurrent transactions, while maintaining consistency
- The shorter the lock acquisition interval, the more requests a database can process.

- **Isolation Levels:**

Spring @transactional isolation enum reflects these values

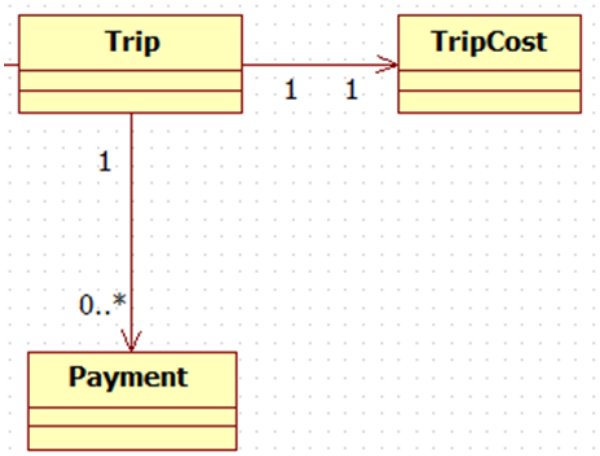
  - SERIALIZABLE (NO dirty, non-repeatable OR phantom reads)
  - REPEATABLE READ (NO dirty OR non-repeatable reads)
  - READ COMMITTED (NO dirty reads)
  - READ UNCOMMITTED (ANYTHING Goes)

- Most databases default to READ COMMITTED

Isolation Types

	dirty reads	non-repeatable reads	phantom reads
READ_UNCOMMITTED	yes	yes	yes
READ_COMMITTED	no	yes	yes
REPEATABLE_READ	no	no	yes
SERIALIZABLE	no	no	no

4. [15 points] For the following relationships implement a Batch fetch of all Trips with their Payments collection. Assume the Payment collection is fetch LAZILY.



What performance problem[s] does the batch fetch address?
How does it work? – Explain the "algorithm" based on a universe of 20 Trips each with a collection of 5-10 Payments.

One fetch for ALL the Trips PLUS N Payment collection fetches where N is based on batch Size & # of Trips.
For example, 20 Trips with batch size = 2 results in 10 collection fetches.
For example, 20 Trips with batch size = 3 results in 7 collection fetches. [6 fetches of 3 PLUS 1 fetch of 2].
For example, 20 Trips with batch size = 4 results in 5 collection fetches, etc…

In TripServiceImpl.Java

```
52    public List<Trip> findAllBatch() {
53        List<Trip> trips =  (List<Trip>)this.findAll();
54        // hydrate - need to access ALL since we don't know batch Size
55        // e.g. if size =2  AND there are 20 trips we need to hydrate trips #1 & #3 & #5 ... & #19
56        for (Trip trip : trips)
57            if (!trip.getPayments().isEmpty()) trip.getPayments().get(0);
58
59         return trips;
60
61    }
```

In Trip.Java

```
61    @OneToMany(fetch = FetchType.LAZY, cascade = { CascadeType.PERSIST, CascadeType.MERGE })
62    @JoinColumn(name="trip_id")
63       @org.hibernate.annotations.Fetch(org.hibernate.annotations.FetchMode.SELECT)
64       @org.hibernate.annotations.BatchSize(size = 2)
65    List<Payment> payments = new ArrayList<>();
```

Solves N+1.. AND Cartesian Product BUT # of collection fetches is "unknown"…

5

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
- **BatchSize**
  - # 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

7

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

8

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

9

"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]

11

Batch Size Fetch

- **Declared as Fetch LAZY:**

```
@OneToMany(mappedBy="member",fetch=FetchType.LAZY)
@Fetch(FetchMode.SELECT)
@BatchSize(size = 3)
private Set<Address> addresses = new HashSet<Address>();
```

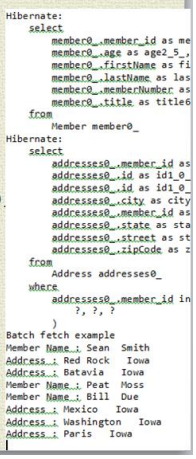
- **Need to "Hydrate" collections:**

```
List<Member> members =  (List<Member>)this.findAll();
for (Member member : members)
        if (!member.getAddresses().isEmpty())
                member.getAddresses().get(0);
```

- **In example, ORM will do ONE Collection Fetch [based on batch size = # parents]**

NOTE:

In example member = 3; BatchSize = 3; 1 Collection fetch
If member = 3; BatchSize = 2; 2 Collection fetches
If member = 3; BatchSize = 1; 3 Collection fetches



5. [15 points] IoC and DI are part of the Spring Core Technologies. Explain in detail what they are and how they work. Explain it in terms of the "Essence of a Spring Application" and the basic component s that make up a Spring Application.

Spring Core Technologies

- **IoC** ***
  Inversion of Control Container
- **AOP** ***
  Aspect-Oriented Programming
- **SpEL** **
  Spring Expression Language that supports querying and manipulating an object graph at runtime.
- **Resource** **
  Common API that abstracts the type of underlying resource such as a URL, file or class path resource.

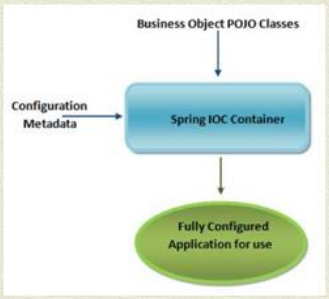
Inversion of Control [IOC]

Objects do not create other objects that they depend on.

- Promotes loose coupling between classes and subsystems
- Adds potential flexibility to a codebase for future changes.
- Classes are easier to unit test in isolation.
- Enable better code reuse.

Spring Core – IoC Container

The Essence of a Spring Application



Spring Core - CORE

The **HEART** of the Spring Framework is the
Spring Inversion Of Control [IOC] **
Container

The IOC container is used to Manage & Configure
Plain Old Java Objects [POJO]
Through
Interfaces

JavaBeans .vs. POJO .vs. Spring Bean

- JavaBean
  Adhere to Sun's JavaBeans specification
  Implements Serializable
  public no-argument [default] constructor
  Reusable Java classes for visual application composition

POJO
'Fancy' way to describe ordinary Java Objects
D
D
S s

Spring Documentation:
"Bean" is used interchangeably with POJO instance
Both mean object instance created from a Java class.

Spring Documentation:
"Component" is used interchangeably with POJO class
Both mean a Java class from which an object instance is created

Spring Bean
Spring managed - configured, instantiated and injected

A Java object can be a JavaBean, a POJO and a Spring bean all at the same time.

Inversion of Control [IOC]

"Hollywood Principle: Don't call us, we'll call you".

The terms IOC and Dependency Injection [DI] are often used interchangeably.

IOC actually refers to:
Dependency Lookup
&
Dependency Injection

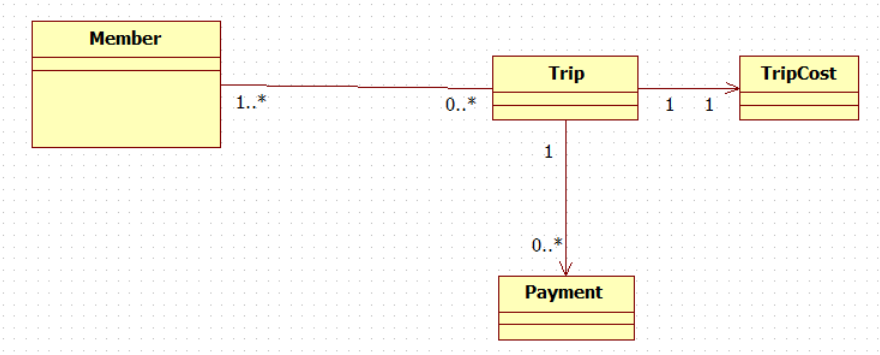
Lookup involves a "pull" of the dependency from a resource: e.g. Registry Lookup

For injection, IOC container "pushes" the dependency...

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

```
public Member findByEmailAndTotalCost(String email,Double amount)
```

The query looks up all Members[s] by email that has a Trip associated with it that has a Trip Cost greater than the supplied amount value. Refer to Problem #2 for field names.



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. MemberDao

```
public Member findByEmailAndTotalCost(String email,Double amount);
```

edu.mum.dao.impl. MemberDaoImpl

```
18    public Member findByEmailAndTotalCost(String email,Double amount) {
19
20        Query query = entityManager.createQuery("select m from Member m,Trip t " +
21                          " where m.email =:email and t member of m.trips "
22                          + " and t.fund.totalAmount > :amount");
23        return (Member) query.setParameter("email", email)
24                      .setParameter("amount", amount).getSingleResult();
25  }
```

edu.mum.service. MemberService

```
public Member findByEmailAndTotalCost(String email,Double amount);
```

edu.mum.service.impl. MemberService Impl

```
public Member findByEmailAndTotalCost(String email,Double amount) {
      return memberDao.findByEmailAndTotalCost(email, amount);
}
```

CS544
Enterprise Architecture
Midterm July 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 ORM technologies:

   T F An example of impedance mismatch is the fact that a RDB puts information in rows and an OO language puts information in Objects

   EXPLAIN:____A RDB consists of individual columns which represent the fields/properties of an object. This requires some way to BIND the DB data to the Object & make sure that the data types match.

   T F A good use case for using an ORM is complex interactions between entities

   EXPLAIN:__ This is a major advantage of an ORM. A RDB entity-entity relationship uses Foreign keys. **The ORM "automatically" maps these relationships, reducing boiler plate code.**

   T **F** The value of a good ORM is that it automatically takes care of all the issues relating to a RDB

   EXPLAIN:__It is NOT possible for an ORM to do everything. It covers mapping and CRUD services. However there are situations where "manual" intervention is necessary [e.g. custom SQL queries]

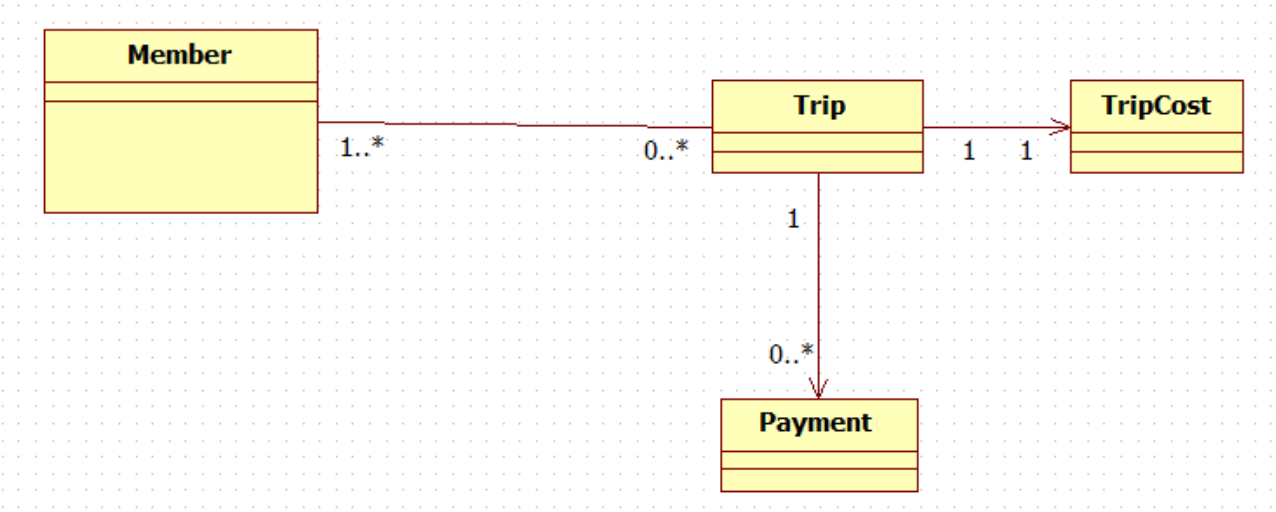
   T **F** JPA is an industry standard for ORMs and has made Hibernate obsolete.

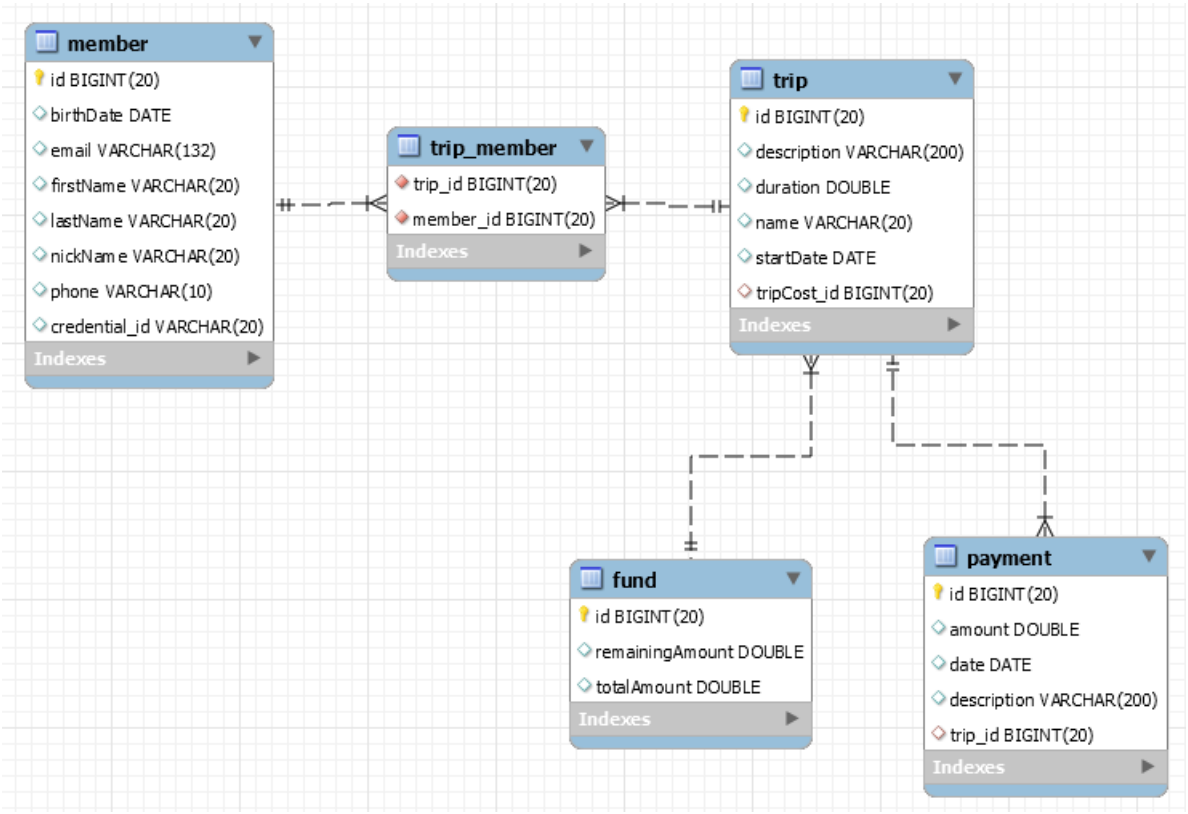
   EXPLAIN:____JPA is an industry standard but does not replace Hibernate. In fact, Hibernate implements JPA As JPA is "only" an API.

   T **F** Native SQL Queries are supported by a good ORM solution and are recommended as the first choice way to access entity relationships.

   EXPLAIN:____Native queries are fallback mechanism to be used when queries are complex and cannot be adequately implemented in JPQL.

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.





Trip.java

```java
@Entity
public class Trip {

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

    @Column(length = 20)
    private String name;

    @Column(length = 200)
    private String description;

    @Column
    private Double duration;

     @Temporal(TemporalType.DATE)
     private Date startDate;

    @Transient
    private Date endDate;

    @ManyToMany(cascade = { CascadeType.PERSIST, CascadeType.MERGE })
    @JoinTable(name = "trip_member", joinColumns = { @JoinColumn(name = "trip_id") }, inverseJoinColumns = {
            @JoinColumn(name = "member_id") })
    List<Member> members = new ArrayList<>();


    @OneToOne(fetch = FetchType.LAZY, cascade = { CascadeType.PERSIST, CascadeType.MERGE, CascadeType.REMOVE })
    @JoinColumn(name="tripCost_id")
    TripCost tripCost;

    @OneToMany(fetch = FetchType.LAZY, cascade = { CascadeType.PERSIST, CascadeType.MERGE })
    @JoinColumn(name="trip_id")
       @org.hibernate.annotations.Fetch(org.hibernate.annotations.FetchMode.SELECT)
       @org.hibernate.annotations.BatchSize(size = 1)
    List<Payment> payments = new ArrayList<>();
```

TripCost.java

```java
@Entity(name="Fund")
public class TripCost {

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

    @Column
    private Double remainingAmount;

    @Column
    private Double totalAmount;

```

Payment.java

```java
@Entity
public class Payment {

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

    @Column(length = 200)
    private String description;

    @Column
    private Double amount;

    @Temporal(TemporalType.DATE)
    private Date date;
```

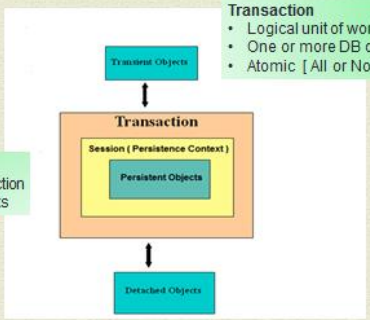
Member.java

```java
@Entity
public class Member {

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

    @Column(length = 20)
    private String firstName;

    @Column(length = 20)
    private String lastName;

    @Column(length = 20)
    private String nickName;

    @Transient
    private Gender gender;

    @Column(length = 132)
    private String email;

    @Column(length = 10)
    private String phone;

    @Temporal(TemporalType.DATE)
    private Date birthDate;

     @ManyToMany( cascade= {CascadeType.PERSIST,CascadeType.MERGE}, mappedBy="members")
    List<Trip> trips;
```

3. [15 points] Transaction management is an important part of RDBMS oriented enterprise applications Spring provides core functionality to assist in transaction management. Describe the Spring transaction functionality, how it is implemented, how it facilitates ORM Transaction management. Include an explanation on how it supports the RDBMS ACID properties of Consistency and Isolation. Be specific. Give Examples.

ANSWER:

ORM – RDB Interactions



Transaction
- Logical unit of work
- One or more DB queries
- Atomic [All or None]

Persistence Context
- Establishes DB connection
- Holds DB-aware objects

ORM Persistence Context [Hibernate: session]

- Transaction Unit o Spring "manages" through @Transact
  - **Common Pattern: *session-per-request***
    - Persistence Context == Database Transaction

```
@Service
@Transactional
public class ProductServiceImpl implements ProductService{
```

Open PersistenceContext/Start Transaction when method is called

```
    public Product getProductById(Long productID) {
    return productDao.findOne(productID);
    }
```

End Transaction/ Close PesistenceContext when method is exited

- END –
  - End Transaction
  - Close a Persistence Context

Spring ORM Support

Comprehensive transaction support is among the most compelling reasons to use the Spring Framework.

- Integration with Hibernate, Java Persistence API (JPA)…
- **Hibernate Support**
  - *First-class integration support through IoC/DI*
    - *Easier testing*
    - *Resource management*
    - *Integrated transaction management*

Spring Framework Data Access

Hibernate Spring Managed Transactions

..On Service Layer

```
@Service
@Transactional   → Starts Transaction
public class VehicleServiceImpl implements edu.mum.service.VehicleService
{
        public void save( Vehicle vehicle) {
                vehicleDao.save(vehicle);
        }
```

```
public abstract class GenericDaoImpl<T> implements GenericDao<T> {
@Autowired
    protected SessionFactory sessionFactory;

 protected Session getSession() {
     return sessionFactory.getCurrentSession();
}
 public void save( T entity ){
     this.getSession().save(entity);
}
```

Reduction in code*:
manage transaction
open/close session
** Compared to Hibernate Solo*

SEE HibernateTransactions DEMO

Declarative Transaction Attributes

Propagation	enum: Propagation	Optional propagation setting.
Isolation	enum: Isolation	Optional isolation level.
ReadOnly	boolean	Read/write vs. read-only transaction
TimeOut	Integer [seconds]	Max Transaction time

```
@Service
@Transactional(propagation=Propagation.REQUIRES_NEW,
                              isolation=Isolation.READ_COMMITTED)
public class ReadCommittedServiceImpl
                    implements edu.mum.service.ReadCommittedService {
```

ACID Database properties

Set of properties that guarantee that database transactions are processed reliably.

Atomicity and Durability are strict properties, i.e., Black or White, All or None

ATOMIC: The transaction is considered a single unit, either the entire transaction completes, or the entire transaction fails.

CONSISTENT: A transaction transforms the database from one consistent state to another consistent state

ISOLATED: Data inside a transaction can not be changed by another concurrent processes until the transaction has been committed

DURABLE: Once committed, the changes made by a transaction are persistent

Consistency and Isolation are "configurable" & Interdependent

Isolation in a Relational Database

- **The challenge is to maximize concurrent transactions, while maintaining consistency**
- **The shorter the lock acquisition interval, the more requests a database can process.**

- **Isolation Levels:**

Spring @transactional isolation enum reflects these values

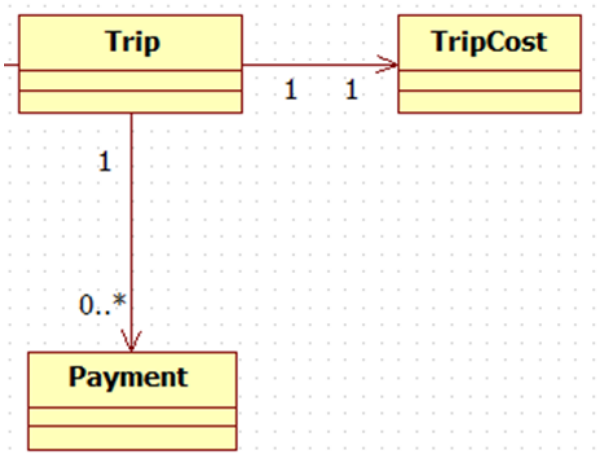
  - **SERIALIZABLE** (NO dirty, non-repeatable OR phantom reads)
  - **REPEATABLE READ** (NO dirty OR non-repeatable reads)
  - **READ COMMITTED** (NO dirty reads)
  - **READ UNCOMMITTED** (ANYTHING Goes)

- **Most databases default to READ COMMITTED**

Isolation Types

	dirty reads	non-repeatable reads	phantom reads
READ_UNCOMMITTED	yes	yes	yes
READ_COMMITTED	no	yes	yes
REPEATABLE_READ	no	no	yes
SERIALIZABLE	no	no	no

4. [15 points] For the following relationships implement a Batch fetch of all Trips with their Payments collection. Assume the Payment collection is fetch LAZILY.



What performance problem[s] does the batch fetch address?
How does it work? – Explain the "algorithm" based on a universe of 20 Trips each with a collection of 5-10 Payments.

One fetch for ALL the Trips PLUS N Payment collection fetches where N is based on batch Size & # of Trips.
For example, 20 Trips with batch size = 2 results in 10 collection fetches.
For example, 20 Trips with batch size = 3 results in 7 collection fetches. [6 fetches of 3 PLUS 1 fetch of 2].
For example, 20 Trips with batch size = 4 results in 5 collection fetches, etc…

In TripServiceImpl.Java

```
52    public List<Trip> findAllBatch() {
53        List<Trip> trips =  (List<Trip>)this.findAll();
54        // hydrate - need to access ALL since we don't know batch Size
55        // e.g. if size =2  AND there are 20 trips we need to hydrate trips #1 & #3 & #5 ... & #19
56        for (Trip trip : trips)
57            if (!trip.getPayments().isEmpty()) trip.getPayments().get(0);
58
59         return trips;
60
61    }
```

In Trip.Java

```
61    @OneToMany(fetch = FetchType.LAZY, cascade = { CascadeType.PERSIST, CascadeType.MERGE })
62    @JoinColumn(name="trip_id")
63       @org.hibernate.annotations.Fetch(org.hibernate.annotations.FetchMode.SELECT)
64       @org.hibernate.annotations.BatchSize(size = 2)
65    List<Payment> payments = new ArrayList<>();
```

Solves N+1.. AND Cartesian Product BUT # of collection fetches is "unknown"…

5

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
- **BatchSize**
  - # 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

7

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

8

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

9

"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]

11

Batch Size Fetch

- **Declared as Fetch LAZY:**

```
@OneToMany(mappedBy="member",fetch=FetchType.LAZY)
@Fetch(FetchMode.SELECT)
@BatchSize(size = 3)
private Set<Address> addresses = new HashSet<Address>();
```

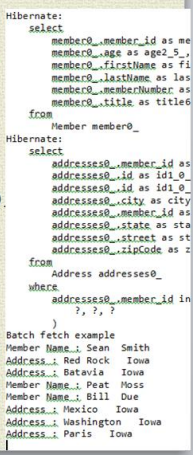
- **Need to "Hydrate" collections:**

```
List<Member> members =  (List<Member>)this.findAll();
for (Member member : members)
        if (!member.getAddresses().isEmpty())
                member.getAddresses().get(0);
```

- **In example, ORM will do ONE Collection Fetch [based on batch size = # parents]**

NOTE:

In example member = 3; BatchSize = 3; 1 Collection fetch
If member = 3; BatchSize = 2; 2 Collection fetches
If member = 3; BatchSize = 1; 3 Collection fetches



5. [15 points] IoC and DI are part of the Spring Core Technologies. Explain in detail what they are and how they work. Explain it in terms of the "Essence of a Spring Application" and the basic component s that make up a Spring Application.

Spring Core Technologies

- **IoC** ***
  Inversion of Control Container
- **AOP** ***
  Aspect-Oriented Programming
- **SpEL** **
  Spring Expression Language that supports querying and manipulating an object graph at runtime.
- **Resource** **
  Common API that abstracts the type of underlying resource such as a URL, file or class path resource.

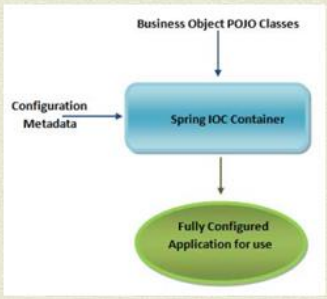
Inversion of Control [IOC]

Objects do not create other objects that they depend on.

- Promotes loose coupling between classes and subsystems
- Adds potential flexibility to a codebase for future changes.
- Classes are easier to unit test in isolation.
- Enable better code reuse.

Spring Core – IoC Container

The Essence of a Spring Application



Spring Core - CORE

The **HEART** of the Spring Framework is the
Spring Inversion Of Control [IOC] **
Container

The IOC container is used to Manage & Configure
Plain Old Java Objects [POJO]
Through
Interfaces

JavaBeans .vs. POJO .vs. Spring Bean

- JavaBean
  Adhere to Sun's JavaBeans specification
  Implements Serializable
  public no-argument [default] constructor
  Reusable Java classes for visual application composition

POJO
'Fancy' way to describe ordinary Java Objects
D
D
S s

Spring Documentation:
"Bean" is used interchangeably with POJO instance
Both mean object instance created from a Java class.

Spring Documentation:
"Component" is used interchangeably with POJO class
Both mean a Java class from which an object instance is created

Spring Bean
Spring managed - configured, instantiated and injected

A Java object can be a JavaBean, a POJO and a Spring bean all at the same time.

Inversion of Control [IOC]

"Hollywood Principle: Don't call us, we'll call you".

The terms IOC and Dependency Injection [DI] are often used interchangeably.

IOC actually refers to:
Dependency Lookup
&
Dependency Injection

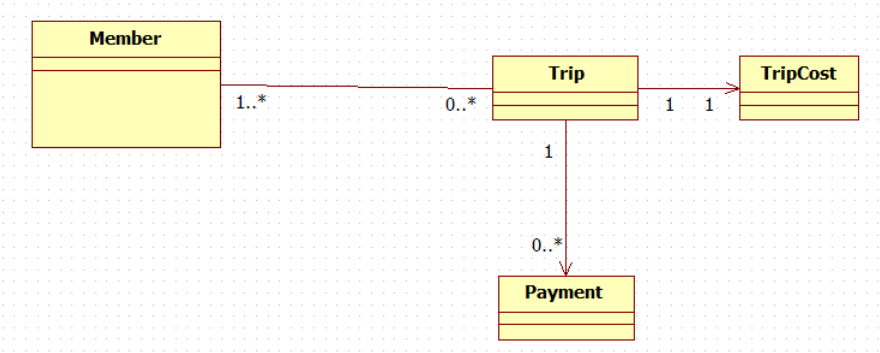
Lookup involves a "pull" of the dependency from a resource: e.g. Registry Lookup

For injection, IOC container "pushes" the dependency...

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

```
public Member findByEmailAndTotalCost(String email,Double amount)
```

The query looks up all Members[s] by email that has a Trip associated with it that has a Trip Cost greater than the supplied amount value. Refer to Problem #2 for field names.



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. MemberDao

```
public Member findByEmailAndTotalCost(String email,Double amount);
```

edu.mum.dao.impl. MemberDaoImpl

```
19    public Member findByEmailAndTotalCost(String email,Double amount) {
20
21        Query query = entityManager.createQuery("select m from Member m,Trip t " +
22                        " where m.email =:email and t member of m.trips "
23                        + " and t.tripCost.totalAmount > :amount");
24
25        Member member =  (Member) query.setParameter("email", email)
26                        .setParameter("amount", amount).getSingleResult();
27        return member;
28  }
29
30
```

edu.mum.service. MemberService

```
public Member findByEmailAndTotalCost(String email,Double amount);
```

edu.mum.service.impl. MemberService Impl

```
public Member findByEmailAndTotalCost(String email,Double amount) {
      return memberDao.findByEmailAndTotalCost(email, amount);
}
```