CS544

Enterprise Architecture

Exam 1 Example

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].

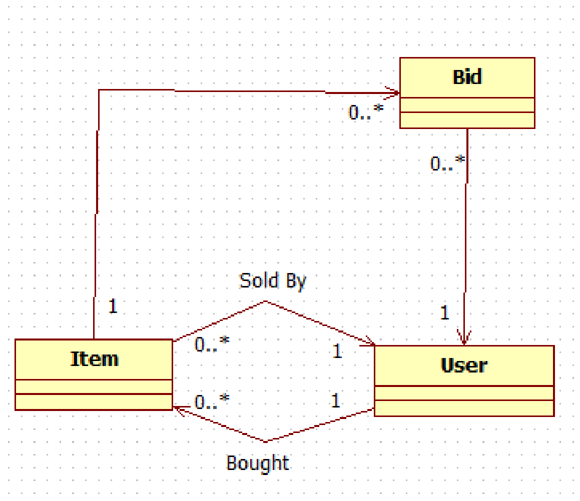
   T **F** A domain object needed in a @Service class is usually a good candidate for Dependency Injection

   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

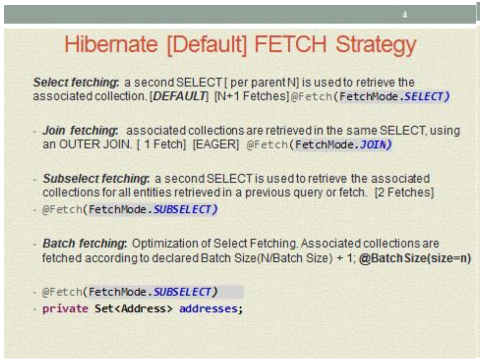
```
public class Item {

    @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.



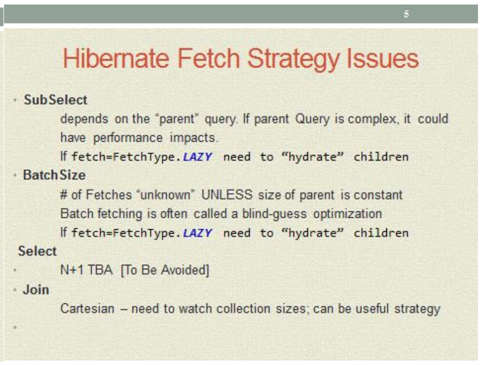
4

Hibernate [Default] FETCH Strategy

Select fetching: a second SELECT [per parent N] is used to retrieve the associated collection. [***DEFAULT***] [N+1 Fetches] `@Fetch(FetchMode.SELECT)`

- ***Join fetching:*** associated collections are retrieved in the same SELECT, using an OUTER JOIN. [1 Fetch] [EAGER] `@Fetch(FetchMode.JOIN)`
- ***Subselect fetching:*** a second SELECT is used to retrieve the associated collections for all entities retrieved in a previous query or fetch. [2 Fetches]
- `@Fetch(FetchMode.SUBSELECT)`
- ***Batch fetching:*** Optimization of Select Fetching. Associated collections are fetched according to declared Batch Size(N/Batch Size) + 1; **@BatchSize(size=n)**

```
@Fetch(FetchMode.SUBSELECT)
private Set<Address> addresses;
```



5

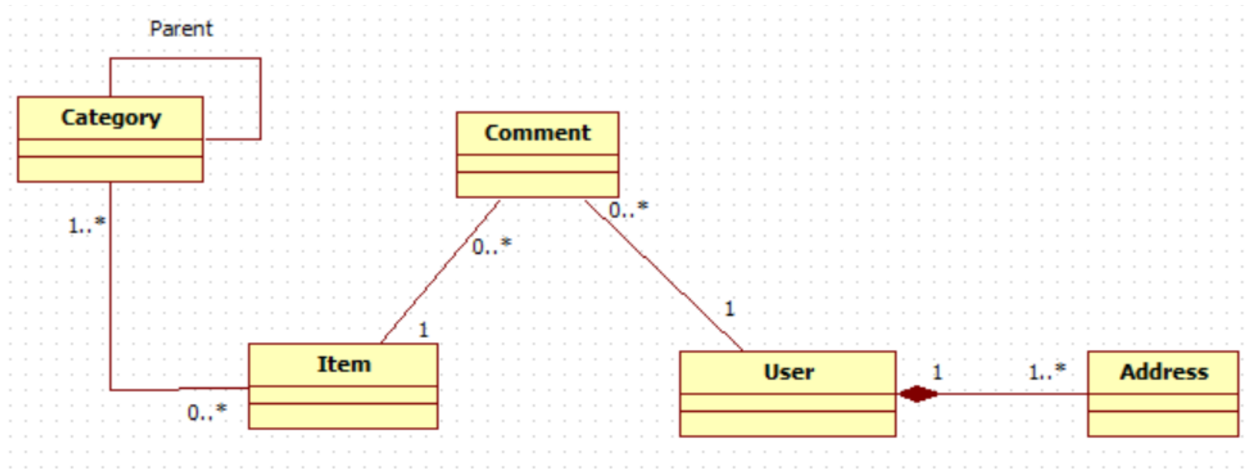
Hibernate Fetch Strategy Issues

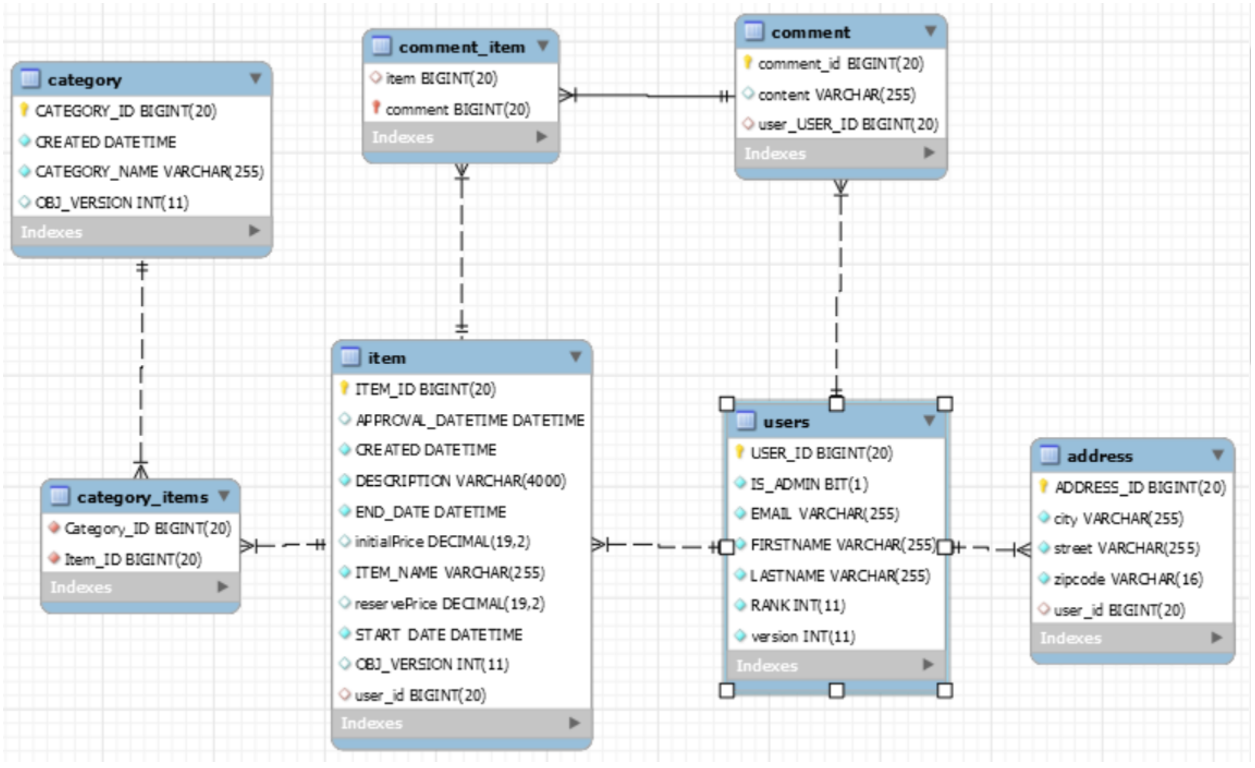
- **SubSelect**
  - depends on the "parent" query. If parent Query is complex, it could have performance impacts.
  - If `fetch=FetchType.LAZY need to "hydrate" children`
- **Batch Size**
  - # of Fetches "unknown" UNLESS size of parent is constant
  - Batch fetching is often called a blind-guess optimization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

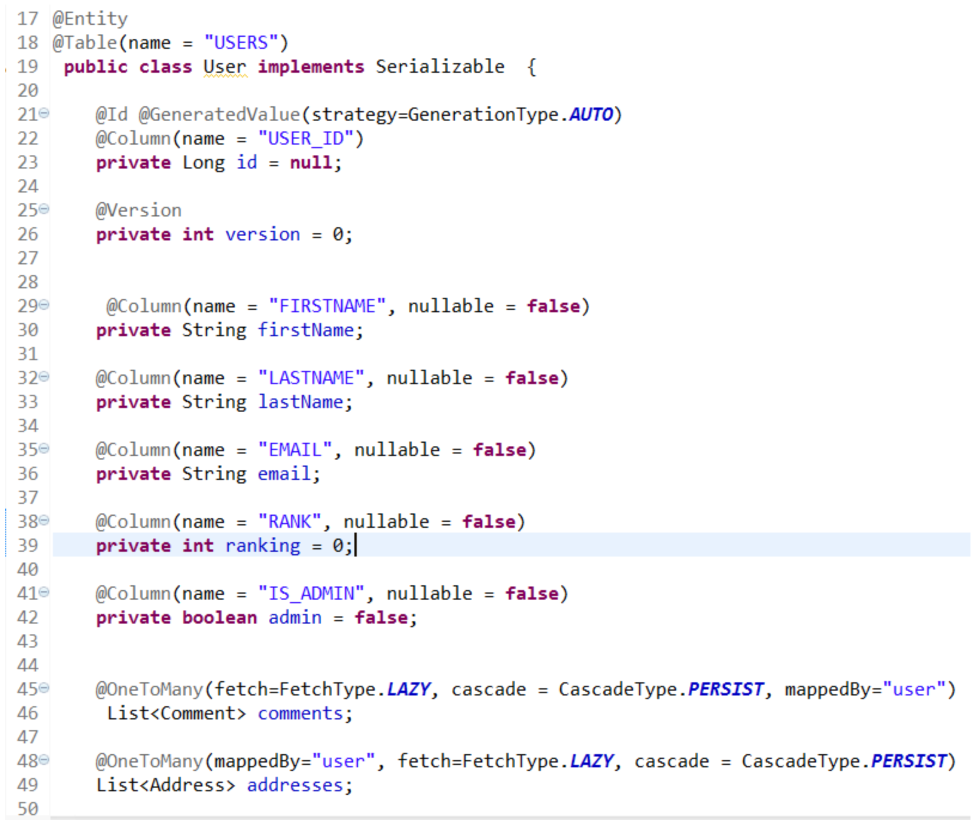
Select

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

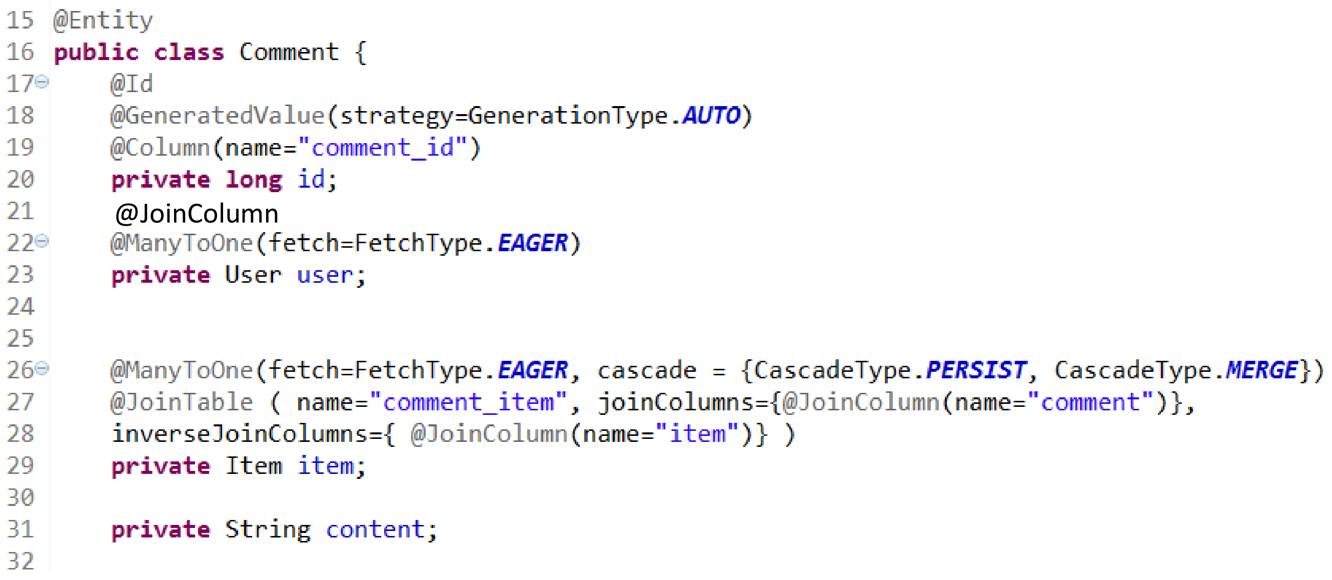
3. [20 points] Annotate the Domain Objects based on the Domain Model and Entity Relationship Diagram provided. NOTE: All the Domain Objects are not listed. All the fields are not listed. Only annotate the objects and fields that are listed.



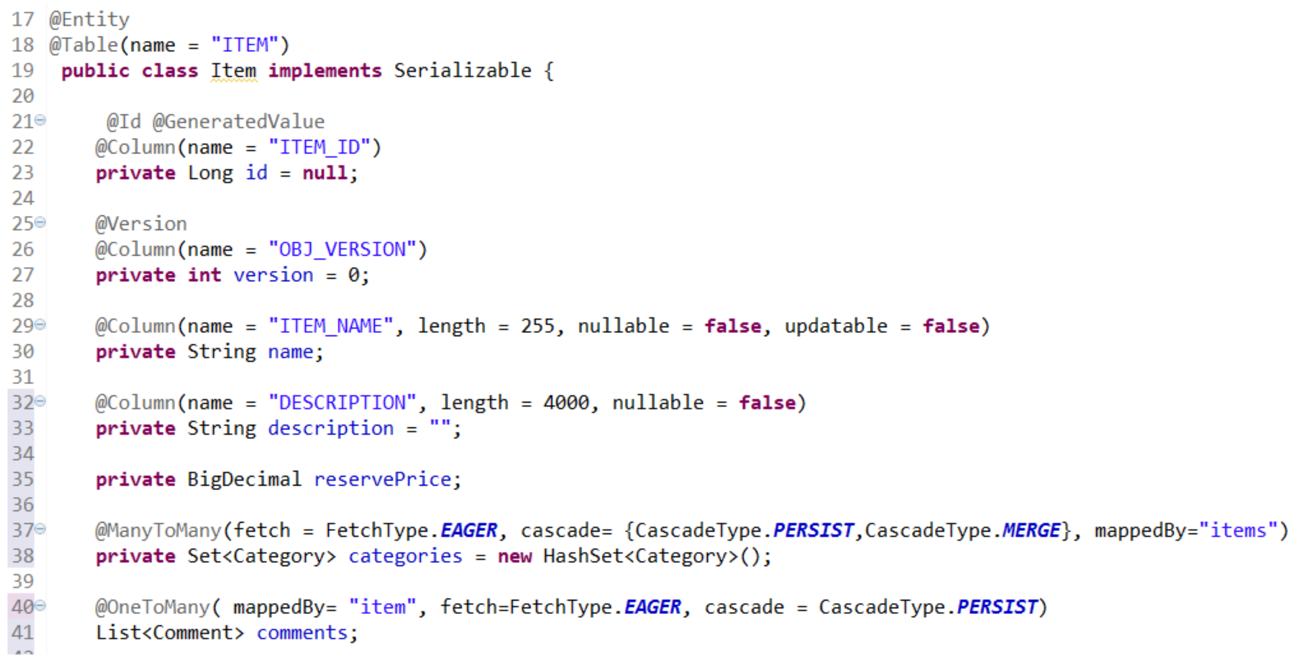




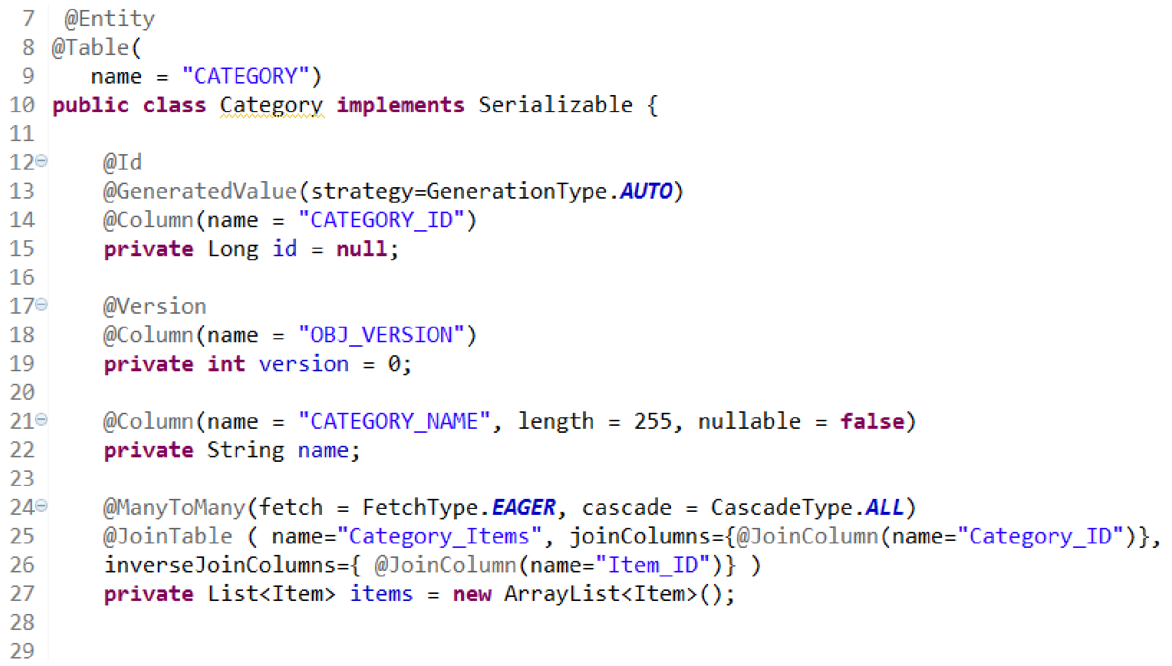
```java
@Entity
@Table(name = "USERS")
 public class User implements Serializable  {

    @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;

```

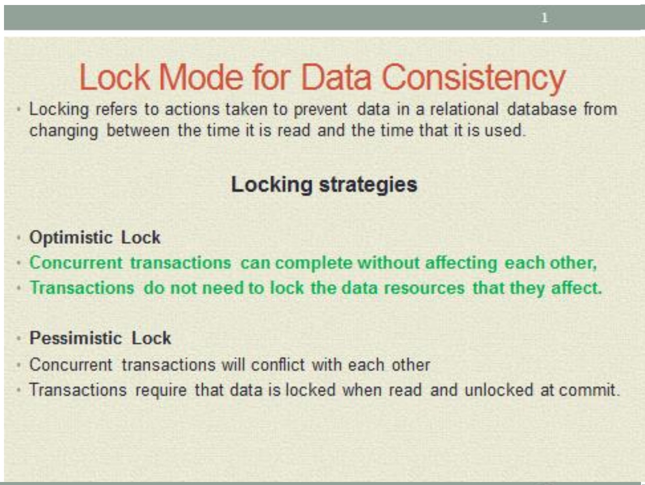


```java
@Entity
@Table(name = "ITEM")
 public class Item implements Serializable {

     @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>();


```

4. [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.



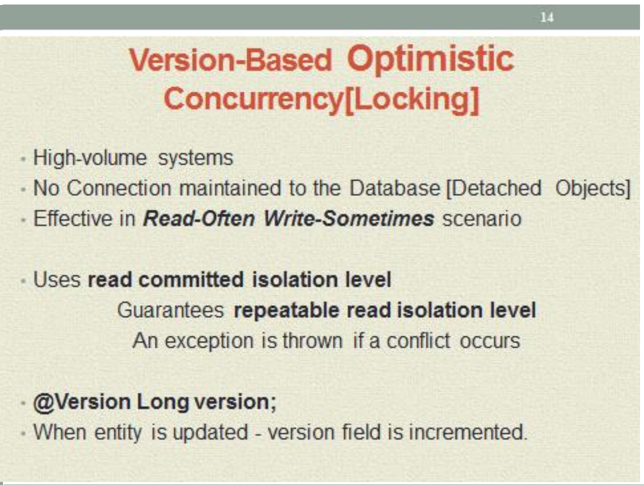
Lock Mode for Data Consistency

- 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.

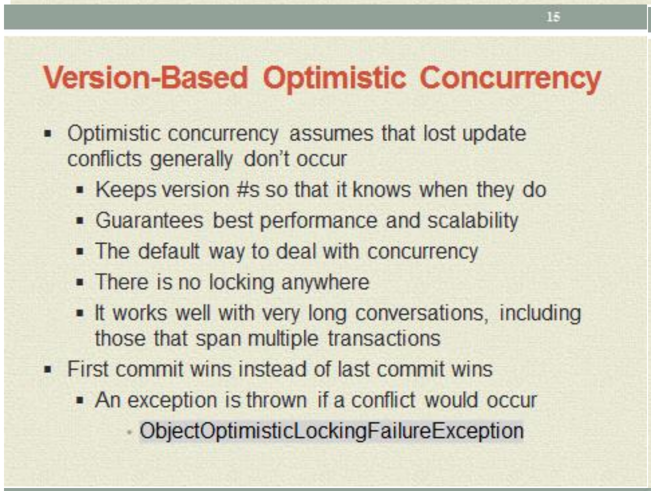


Version-Based Optimistic Concurrency[Locking]

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

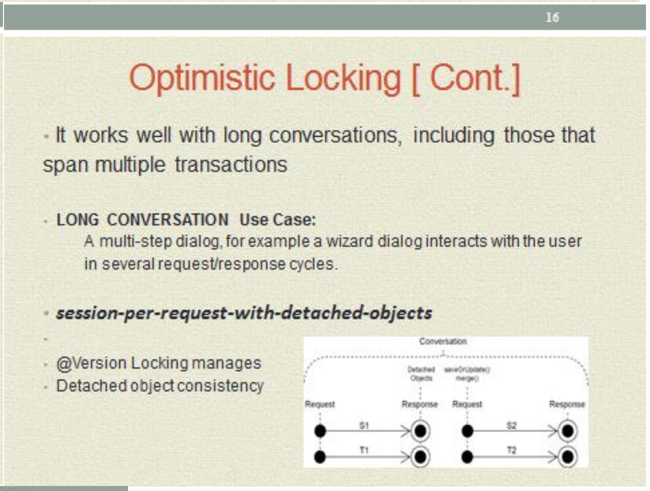
- Uses **read committed isolation level**
  Guarantees **repeatable read isolation level**
  An exception is thrown if a conflict occurs

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



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



Optimistic Locking [Cont.]

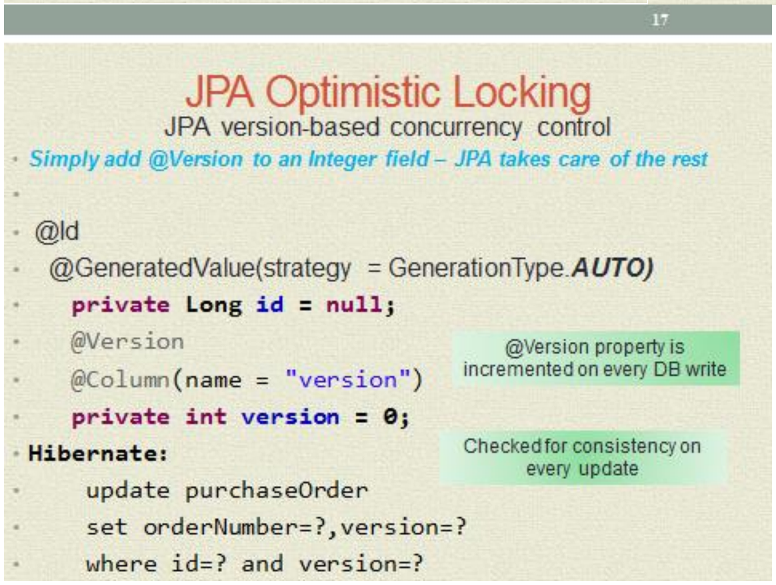
- It works well with long conversations, including those that span multiple transactions

- **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

Conversation
Detached Objects
saveOrUpdate() merge()
Request
Response
Request
Response
S1
T1
S2
T2



JPA Optimistic Locking

JPA version-based concurrency control

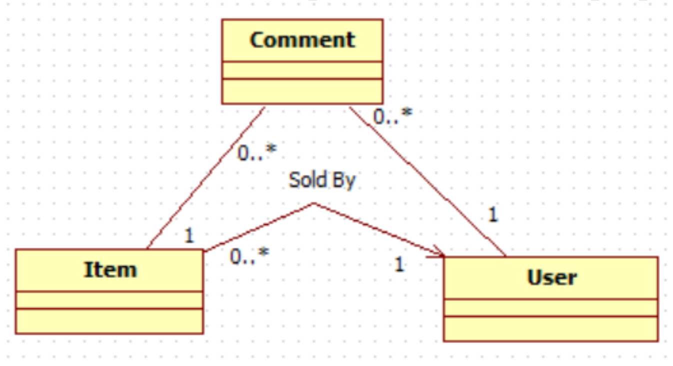
- *Simply add @Version to an Integer field – JPA takes care of the rest*

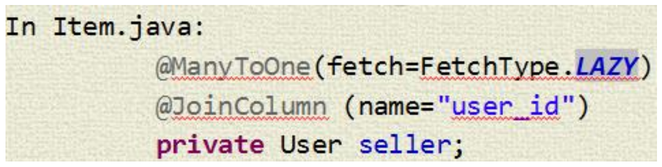
```
@Id
 @GeneratedValue(strategy = GenerationType.AUTO)
  private Long id = null;
  @Version
  @Column(name = "version")
  private int version = 0;
Hibernate:
    update purchaseOrder
    set orderNumber=?,version=?
    where id=? and version=?
```

@Version property is incremented on every DB write

Checked for consistency on every update

5. [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

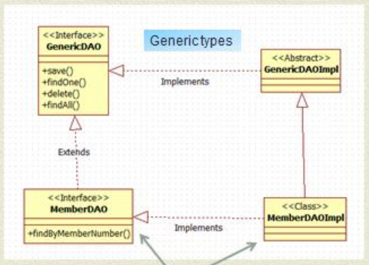
    public User findBySoldItemInitialPrice(String email,String itemName, BigDecimal initialPrice);

edu.mum.service.impl. UserServiceImpl

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

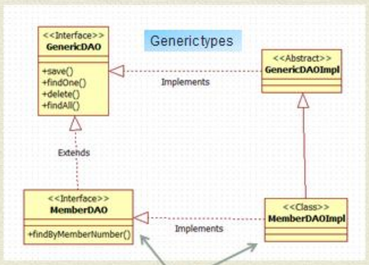
}

“Classic” ORM GenericDAO



Adds Domain Object specific functionality

“Classic” ORM GenericDAO



Adds Domain Object specific functionality

Generic DAO Implementation

<<Interface>>
GenericDAO
+save()
+findOne()
+delete()
+findAll()
Implements
<<Abstract>>
GenericDAOImpl

```
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);
```

<<Interface>>
MemberDAO
+findByMemberNumber()
Implements
<<Class>>
MemberDAOImpl

```
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();
}
```