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]

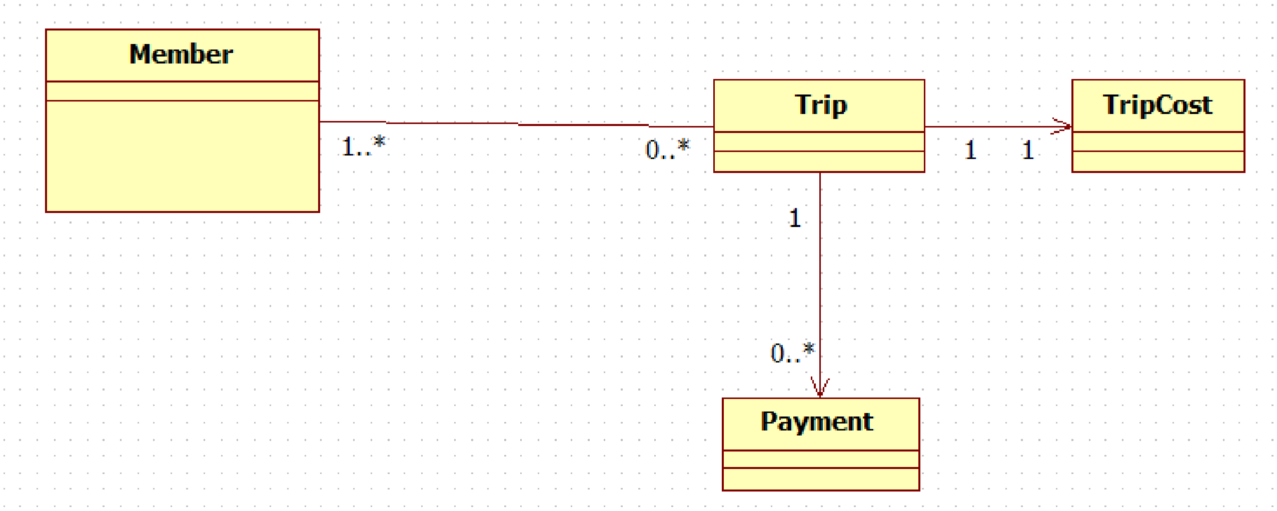
T [✓ **F**] JPA is an industry standard for ORMs and has made Hibernate obsolete.

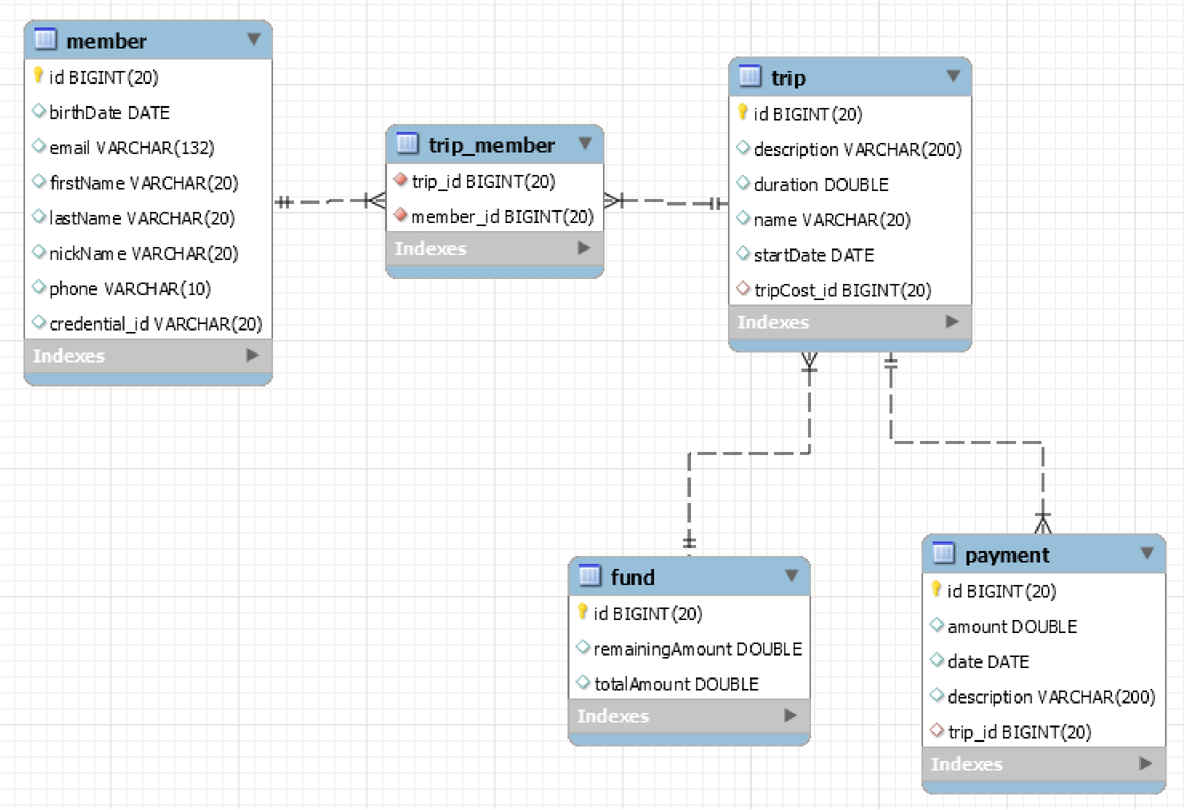
EXPLAIN:____JPA is an industry standard but does not replace Hibernate. In fact, Hibernate implements JPA As JPA is "only" an API.

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

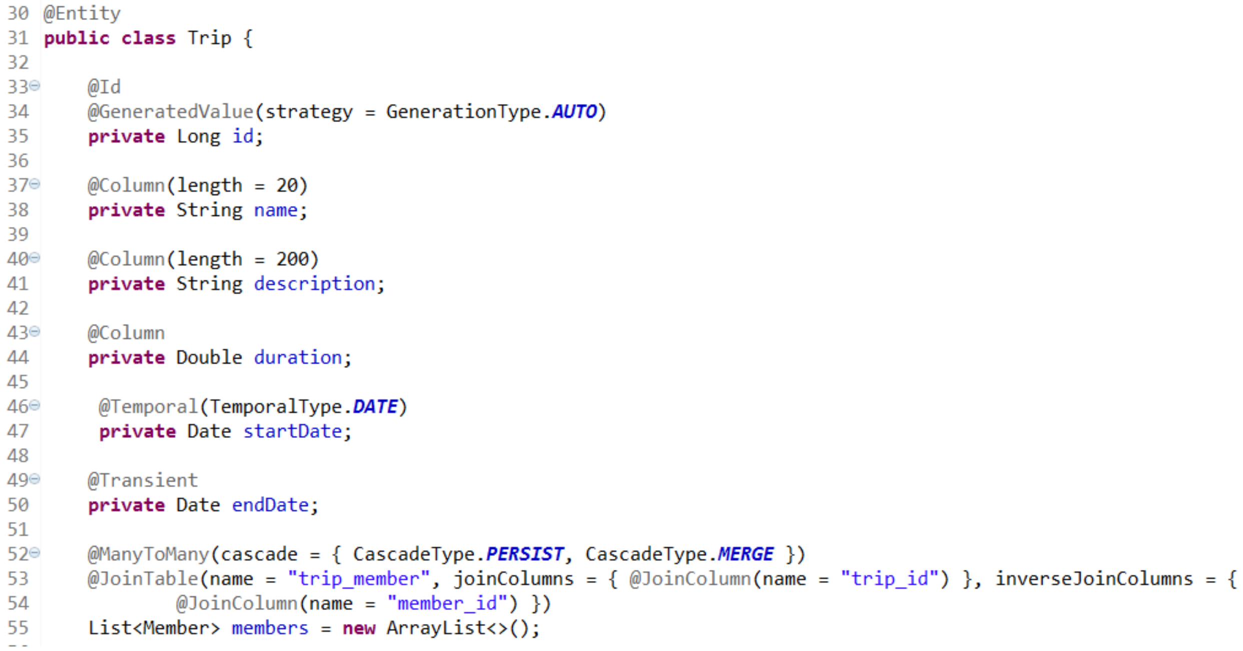
EXPLAIN:____Native queries are fallback mechanism to be used when queries are complex and cannot be adequately implemented in JPQL.

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.

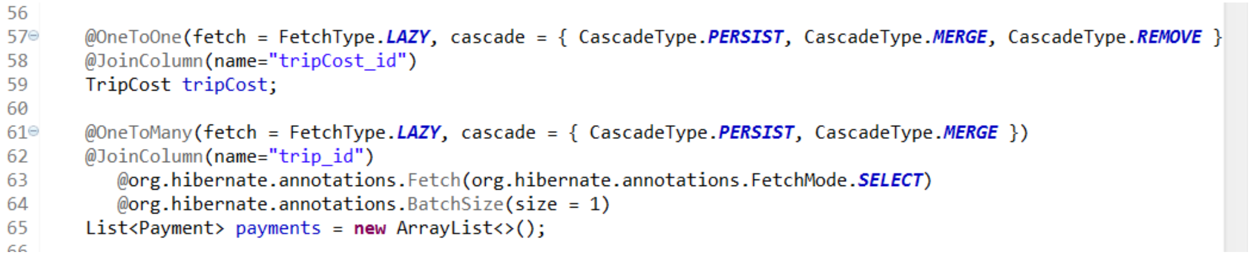




Trip.java

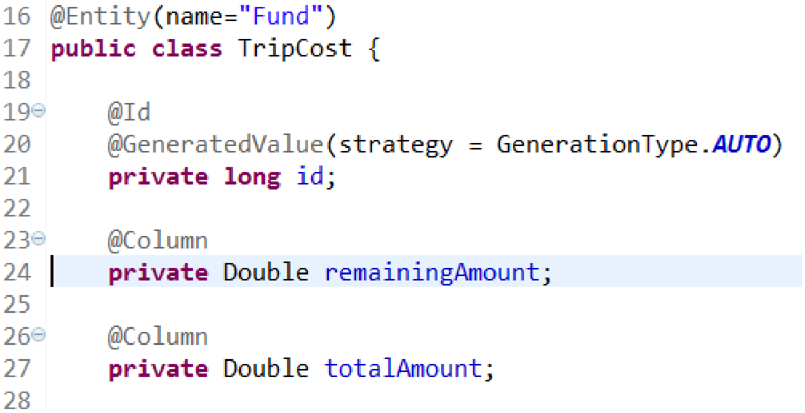


```java
@Entity
public class Trip {

    @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<>();
```



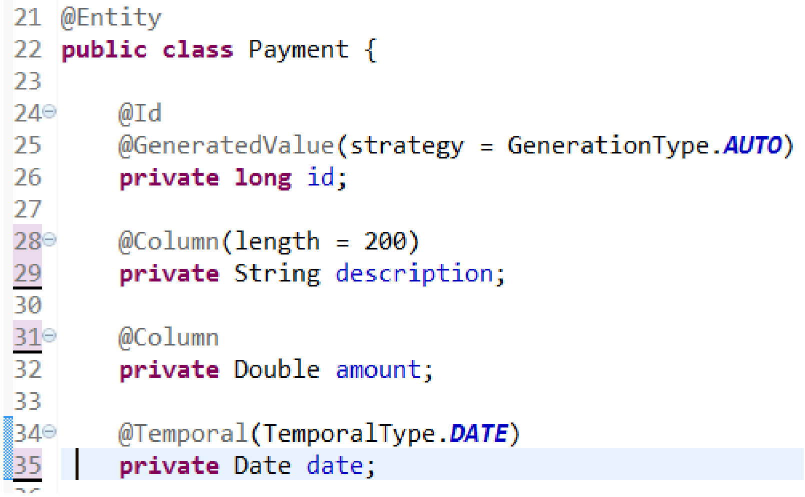
```java

    @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



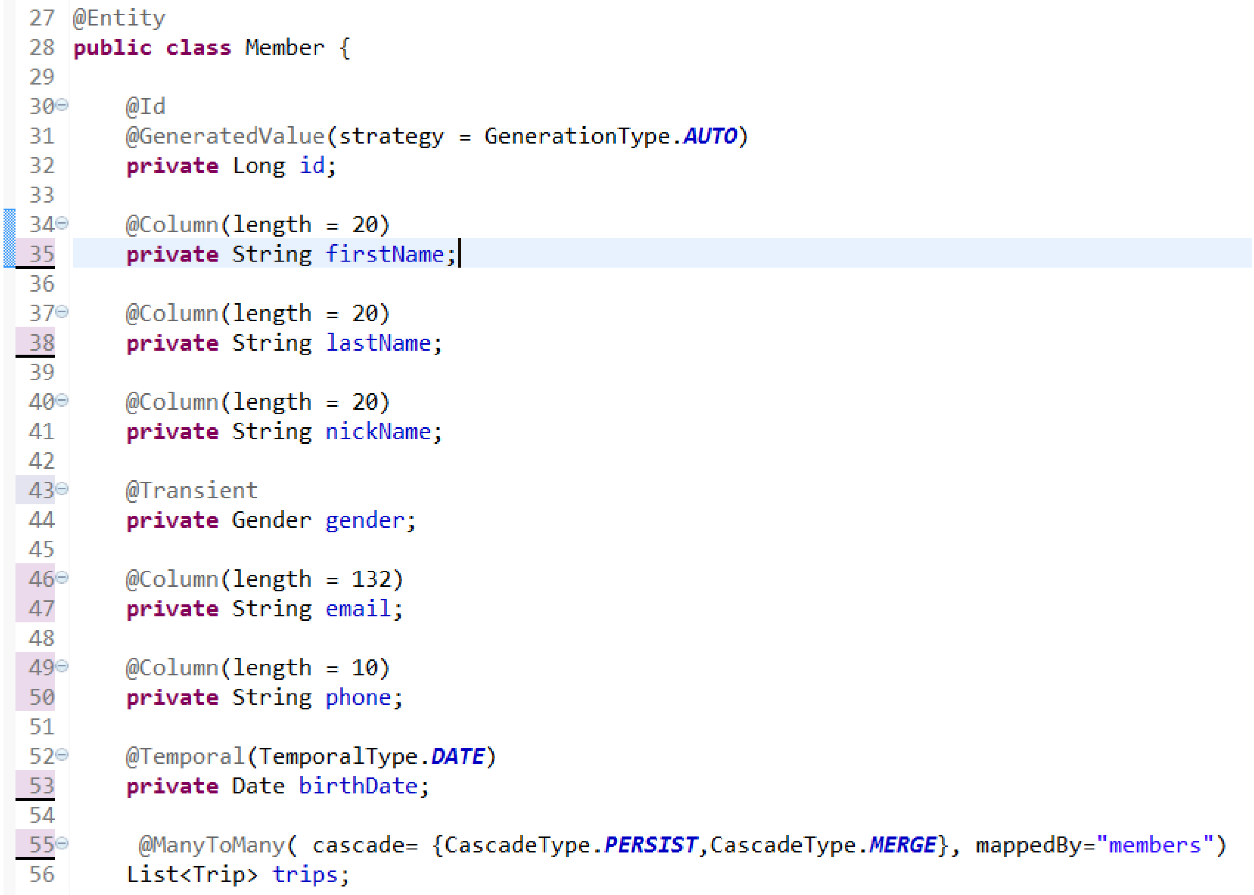
```java
@Entity(name="Fund")
public class TripCost {

    @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;
```

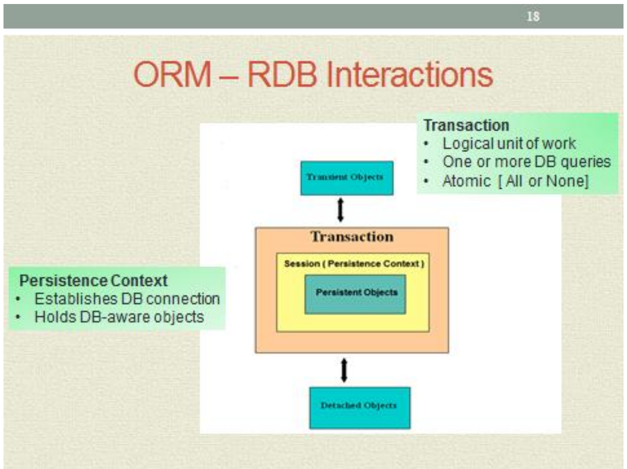
Member.java



```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.

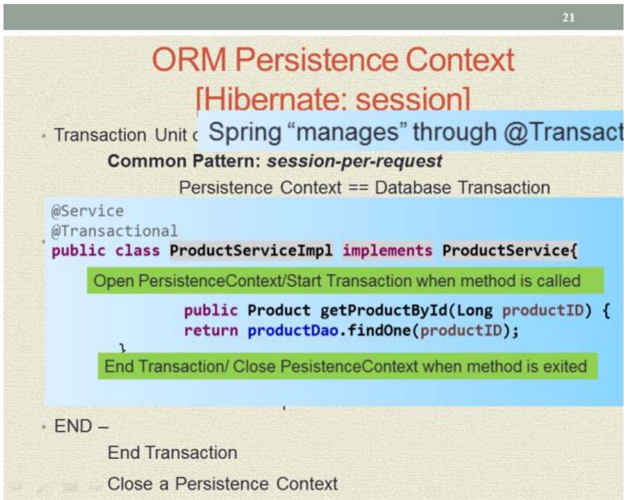
ANSWER:



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 c

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

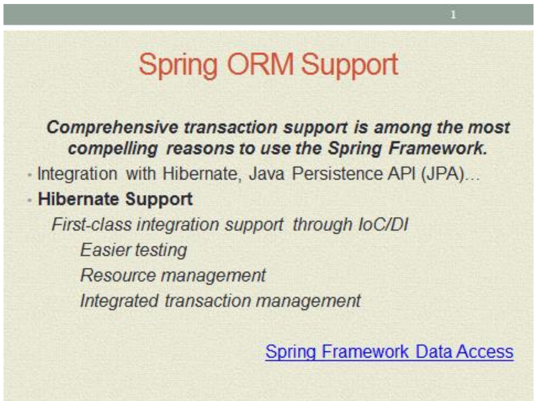
```
        public Product getProductById(Long productID) {
        return productDao.findOne(productID);
   }
```

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**

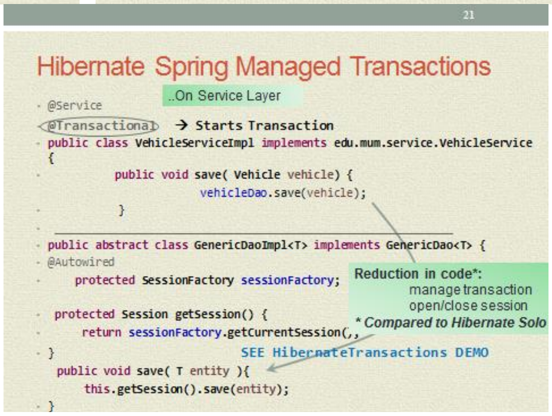
  First-class integration support through IoC/DI

  Easier testing

  Resource management

  Integrated transaction management

Spring Framework Data Access



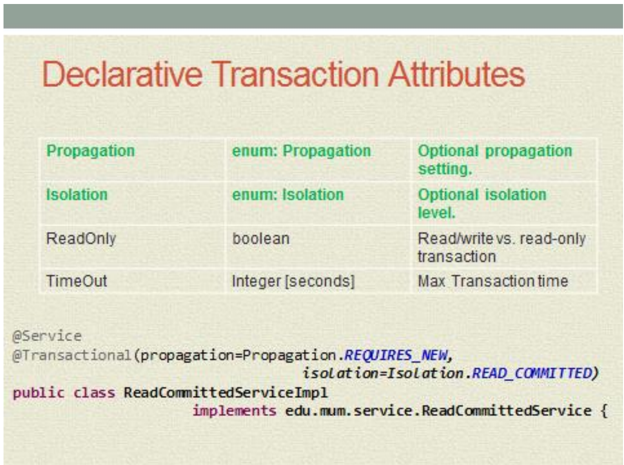
Hibernate Spring Managed Transactions

..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);
}
```

SEE HibernateTransactions DEMO

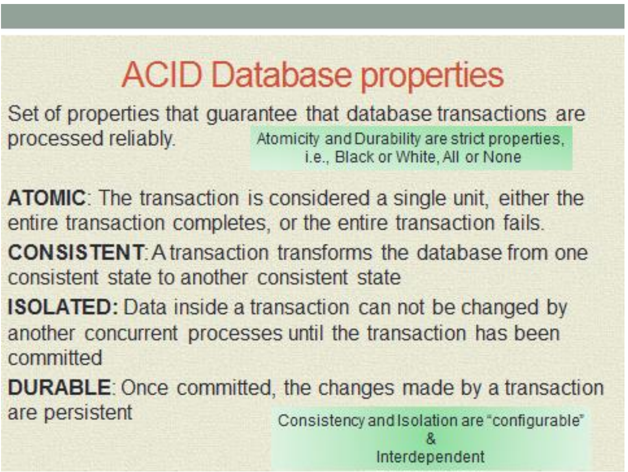
Reduction in code*:
manage transaction
open/close session
** Compared to Hibernate Solo*



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

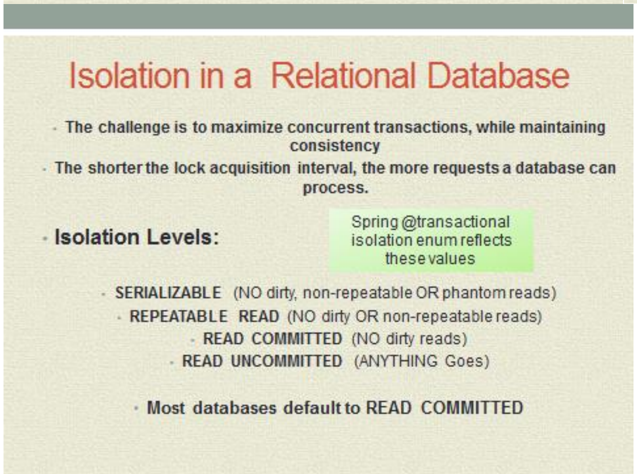
ATOMIC: The transaction is considered a single unit, either the entire transaction completes, or the entire transaction fails.

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

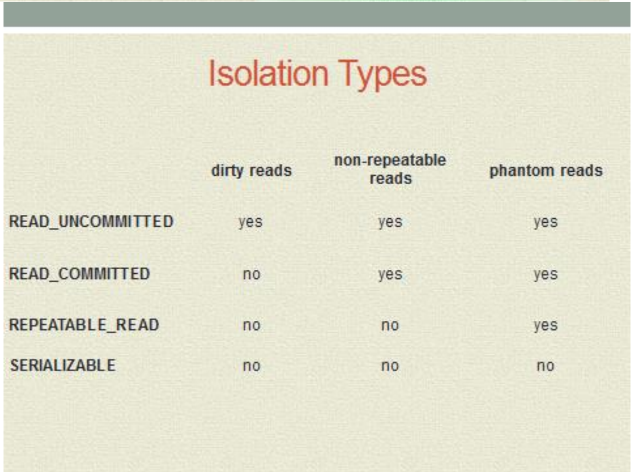


Isolation in a Relational Database

- **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

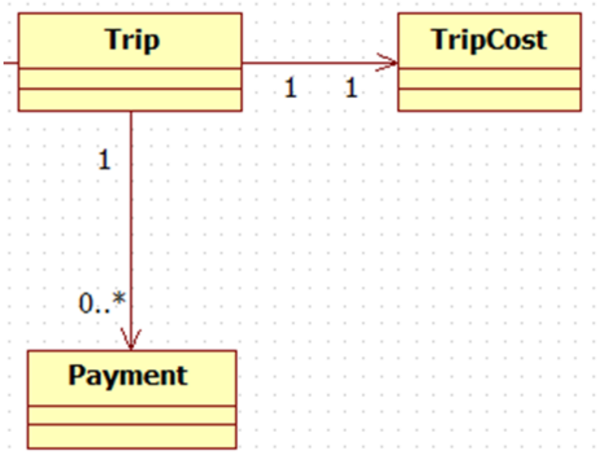
  - **SERIALIZABLE** (NO dirty, non-repeatable OR phantom reads)
  - **REPEATABLE READ** (NO dirty OR non-repeatable reads)
  - **READ COMMITTED** (NO dirty reads)
  - **READ UNCOMMITTED** (ANYTHING Goes)
- **Most databases default to READ COMMITTED**



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.

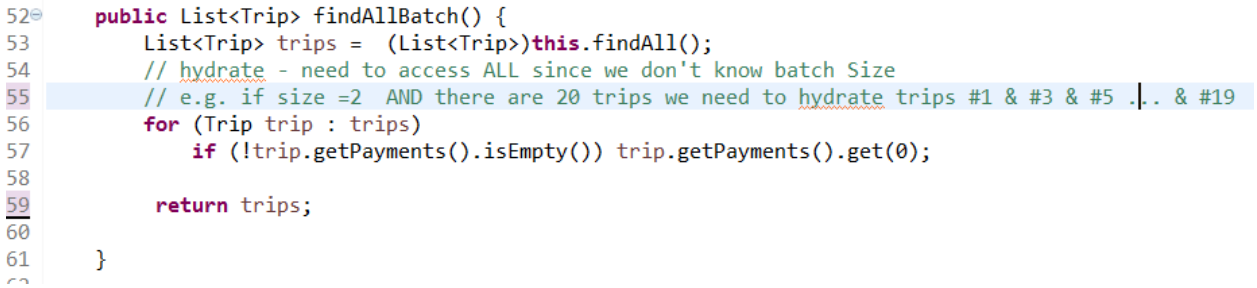


What performance problem[s] does the batch fetch address?

How does it work? – Explain the “algorithm” based on a universe of 20 Trips each with a collection of 5-10 Payments.

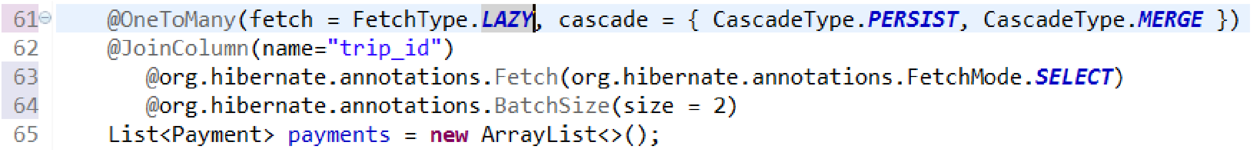
One fetch for ALL the Trips PLUS N Payment collection fetches where N is based on batch Size & # of Trips.
For example, 20 Trips with batch size = 2 results in 10 collection fetches.
For example, 20 Trips with batch size = 3 results in 7 collection fetches. [6 fetches of 3 PLUS 1 fetch of 2].
For example,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

In TripServiceImpl.Java



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

         return trips;

    }
```

In Trip.Java



```
    @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 = 2)
    List<Payment> payments = new ArrayList<>();
```

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

Hibernate Fetch Strategy Issues

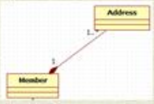
- SubSelect
  - depends on the "parent" query. If parent Query is complex, it could have performance impacts.
  - If `fetch=FetchType.LAZY need to "hydrate" children`
- BatchSize
  - # of Fetches "unknown" UNLESS size of parent is constant
  - Batch fetching is often called a blind-guess optimization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

N+1 Problem

- Member has a OneToMany relationship with Address

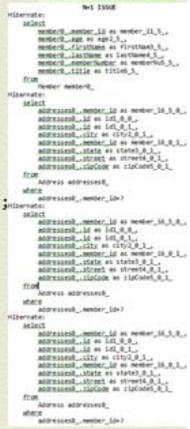


- Declared as Fetch EAGER:
- `@OneToMany(mappedBy="member",fetch=FetchType.EAGER)`
- `private Set<Address> addresses = new HashSet<Address>();`

- For
- `entityManager.createQuery( "from Member")`
- `.getResultList();`

```
ORM will issue ONE fetch for the Member
                    &
And N fetches; one for each Set of Addresses
```

Example N = 3



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

Cartesian Product Problem

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

2 Members
X
Address/Member

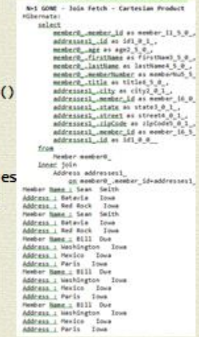
- Member [STILL] has a OneToMany relationship with Address
- NOW - Declared as Fetch LAZY:
- `@OneToMany(mappedBy="member",fetch=FetchType.LAZY)`
- `private Set<Address> addresses = new HashSet<Address>()`
- For:
- `Query query=entityManager.createQuery("SELECT m FROM Member AS m JOIN FETCH m.addresses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

of "root" table results

X

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

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



```
 N+1 GONE - Join Fetch - Cartesian Product
Hibernate:
    select
        member0_.member_id as member_i1_5_0_,
        addresses1_.id as id1_0_1_,
        member0_.age as age2_5_0_,
        member0_.firstName as firstNam3_5_0_,
        member0_.lastName as lastName4_5_0_,
        member0_.memberNumber as memberNu5_5_
        member0_.title as title6_5_0_,
        addresses1_.city as city2_0_1_,
        addresses1_.member_id as member_i6_0_
        addresses1_.state as state3_0_1_,
        addresses1_.street as street4_0_1_,
        addresses1_.zipCode as zipCode5_0_1_,
        addresses1_.member_id as member_i6_5_
        addresses1_.id as id1_0_0__
    from
        Member member0_
    inner join
        Address addresses1_
            on member0_.member_id=addresses1_
Member Name : Sean  Smith
Address : Batavia   Iowa
Address : Red Rock   Iowa
Member Name : Sean  Smith
Address : Batavia   Iowa
Address : Red Rock   Iowa
Member Name : Bill  Due
Address : Washington   Iowa
Address : Mexico   Iowa
Address : Paris   Iowa
Member Name : Bill  Due
Address : Washington   Iowa
Address : Mexico   Iowa
Address : Paris   Iowa
Member Name : Bill  Due
Address : Washington   Iowa
Address : Mexico   Iowa
Address : Paris   Iowa
```

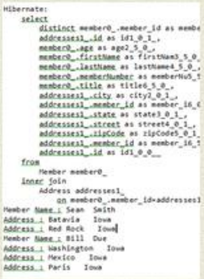
"Reduce" the Cartesian Product

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

- DISTINCT keyword removes duplicates

However

It accomplishes it in Memory [After DB fetch]



```
Hibernate:
    select
        distinct member0_.member_id as membe
        addresses1_.id as id1_0_1_,
        member0_.age as age2_5_0_,
        member0_.firstName as firstNam3_5_0_
        member0_.lastName as lastName4_5_0_,
        member0_.memberNumber as memberNu5_5
        member0_.title as title6_5_0_,
        addresses1_.city as city2_0_1_,
        addresses1_.member_id as member_i6_0
        addresses1_.state as state3_0_1_,
        addresses1_.street as street4_0_1_,
        addresses1_.zipCode as zipCode5_0_1_
        addresses1_.member_id as member_i6_5
        addresses1_.id as id1_0_0__
    from
        Member member0_
    inner join
        Address addresses1_
            on member0_.member_id=addresses1
Member Name : Sean  Smith
Address : Batavia   Iowa
Address : Red Rock   Iowa
Member Name : Bill  Due
Address : Washington   Iowa
Address : Mexico   Iowa
Address : Paris   Iowa
```

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>();`

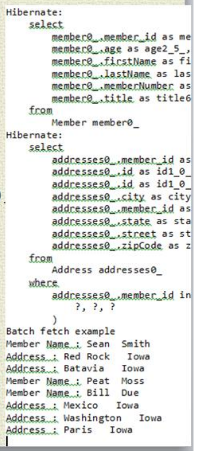
- Need to "Hydrate" collections:

```
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]

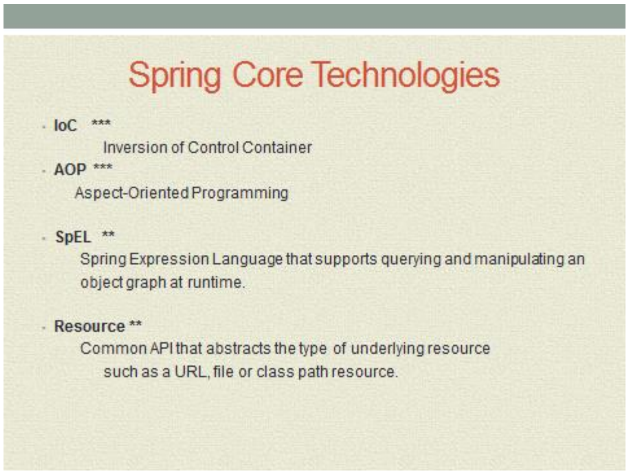
NOTE:

```
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
```



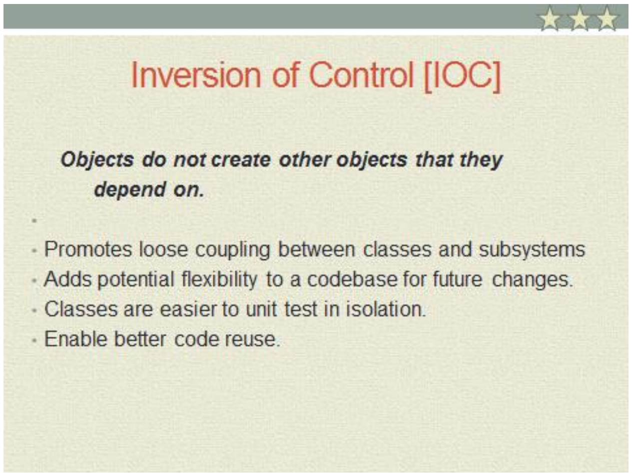
```
Hibernate:
    select
        member0_.member_id as me
        member0_.age as age2_5_,
        member0_.firstName as fi
        member0_.lastName as las
        member0_.memberNumber as
        member0_.title as title6
    from
        Member member0_
Hibernate:
    select
        addresses0_.member_id as
        addresses0_.id as id1_0_
        addresses0_.id as id1_0_
        addresses0_.city as city
        addresses0_.member_id as
        addresses0_.state as sta
        addresses0_.street as st
        addresses0_.zipCode as z
    from
        Address addresses0_
    where
        addresses0_.member_id in
            ?, ?, ?
        )
Batch fetch example
Member Name : Sean  Smith
Address : Red Rock   Iowa
Address : Batavia   Iowa
Member Name : Peat  Moss
Member Name : Bill  Due
Address : Mexico   Iowa
Address : Washington   Iowa
Address : Paris   Iowa
```

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

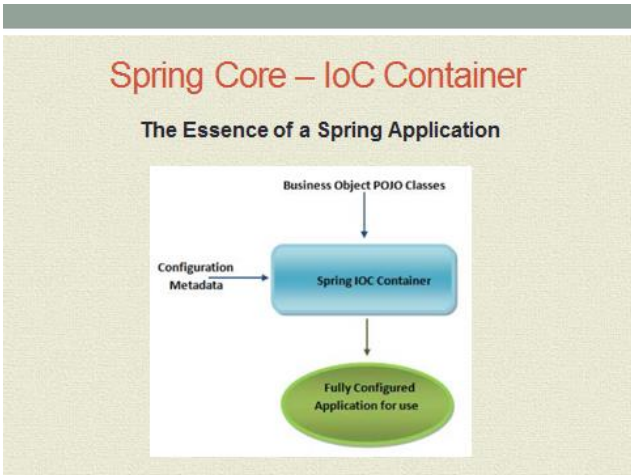
- IoC ***
  Inversion of Control Container
- AOP ***
  Aspect-Oriented Programming
- SpEL **
  Spring Expression Language that supports querying and manipulating an object graph at runtime.
- Resource **
  Common API that abstracts the type of underlying resource such as a URL, file or class path resource.



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

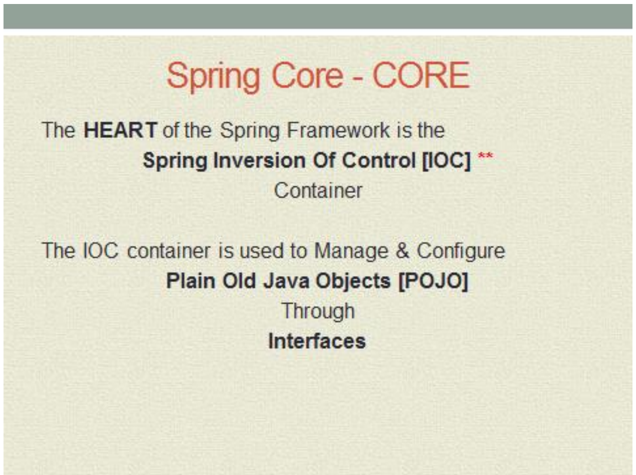
The Essence of a Spring Application

Business Object POJO Classes

Configuration Metadata

Spring IOC Container

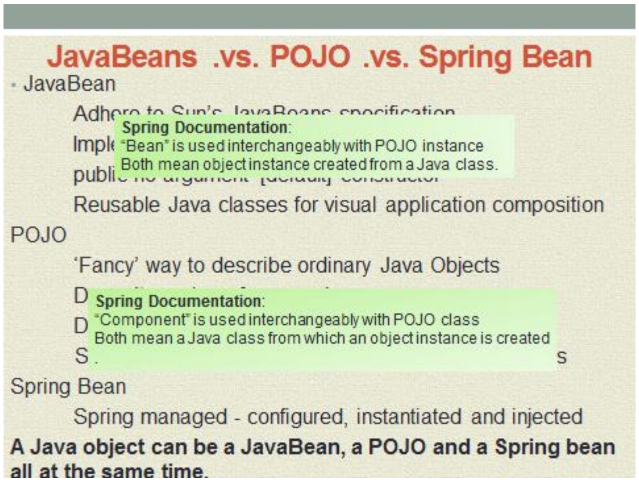
Fully Configured Application for use



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
  Imple
  public no argument [default] constructor
  Reusable Java classes for visual application composition

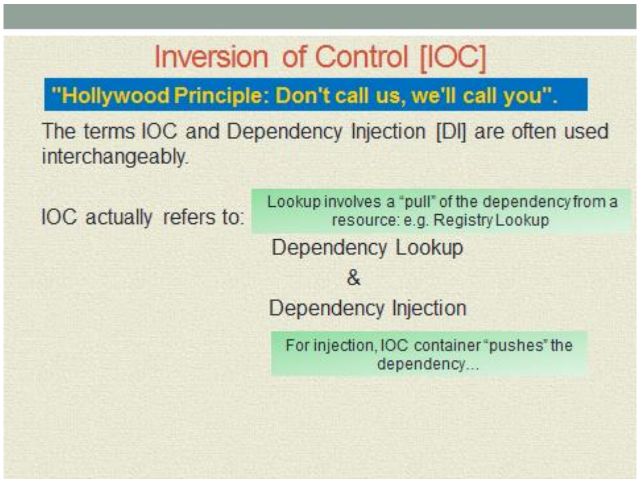
POJO
  ‘Fancy’ way to describe ordinary Java Objects
  D
  D
  S s

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

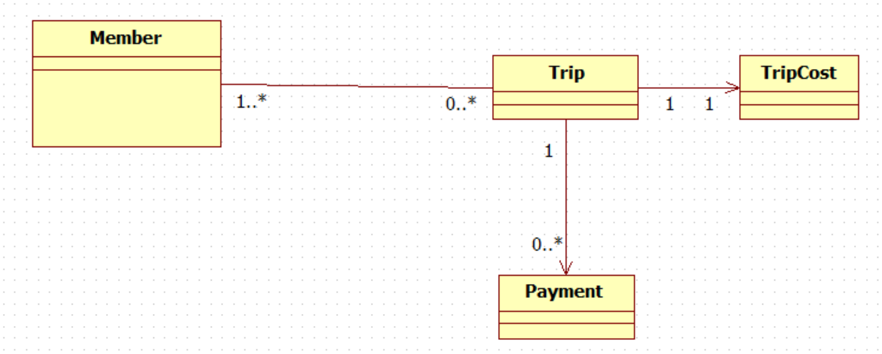
Lookup involves a “pull” of the dependency from a resource: e.g. Registry Lookup

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.

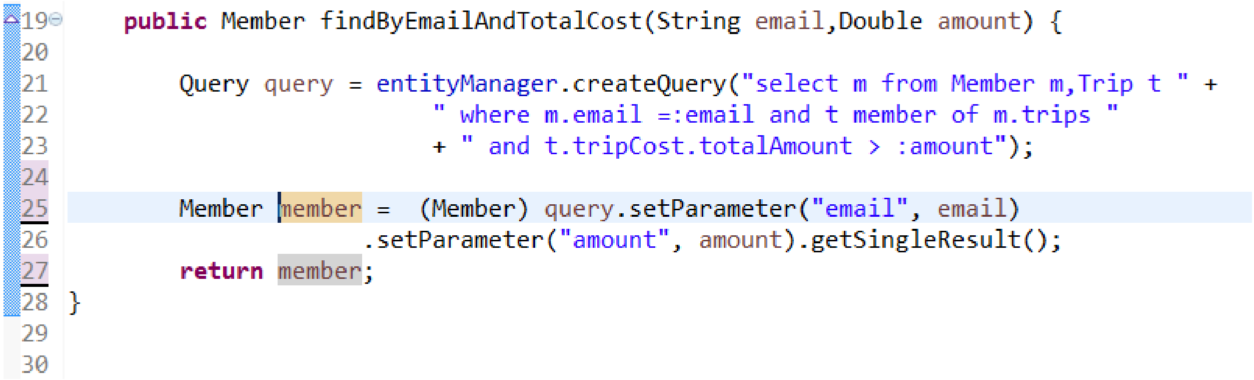


The Query should be a parameterized query. Also show the modifications to all classes in order to adhere to the N-Tier architecture convention. Identify the specific packages that each modified class is in.

edu.mum.dao. MemberDao

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

edu.mum.dao.impl. MemberDaoImpl



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

        Query query = entityManager.createQuery("select m from Member m,Trip t " +
                      " where m.email =:email and t member of m.trips "
                      + " and t.tripCost.totalAmount > :amount");

        Member member =  (Member) query.setParameter("email", email)
                .setParameter("amount", amount).getSingleResult();
        return member;
}
```

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);
}
```