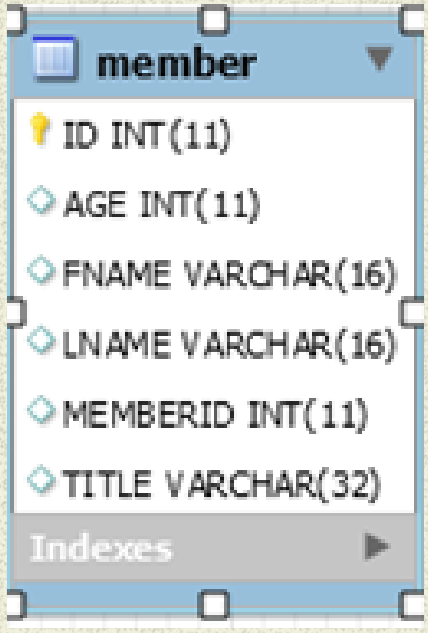
BASIC ORM MAPPING & CRUD OPERATIONS

How to Get

From Here

```
public class Member {
private long id;

private String firstName;

private String lastName;
private int age;

private String title;
private int memberNumber;
```

To Here



Basic Class to Database Table Mapping

- Use a Domain Class POJO
  - Mutators & Accessors [getters/setters]
  - Default Constructor

Entities instantiated using Reflection API Constructor.newInstance()

- Annotate the Domain Class
  - @Entity declares a class as a persistent class
  - @Table identifies the RDB table name

Default: class name of the entity

  - @Id declares the identifier property of the entity.

For ID - use a nullable (i.e., non-primitive) type.

- @Id on field means field annotations
- @Id on setter means property annotations

@Id Annotation
Identifier Generation Strategy

@**GeneratedValue**(strategy=GenerationType.***AUTO***)

Generating the identifier property:

- **AUTO** either identity, sequence or table depending on the underlying DB
- **TABLE** Special table holds the id
- I**DENTITY** Identity column […in entity table]

  DB provides facility to generate ID value during insertion
- **SEQUENCE**

  DB has facility to create a special/custom sequence generator

AUTO is the preferred type for portability (across DB vendors).

For MySQL AUTO preferred type is IDENTITY
Which looks like this:
ID INT PRIMARY KEY AUTO_INCREMENT

MySQL GenerationType.*AUTO* with Hibernate 5

Uses GenerationType.***TABLE*** NOT GenerationType.***IDENTITY***

Table has performance and scalability issues

SOLUTION:

@GeneratedValue(strategy= GenerationType.***IDENTITY***)

OR

```
@GeneratedValue(strategy=GenerationType.AUTO,generator="native")
@GenericGenerator(name = "native", strategy = "native")
```

NOTE:

`strategy = "native"` - selects ***identity***, sequence or hilo depending upon the capabilities of the underlying database.

MySQL GenerationType Performance

```
Hibernate: create table Authority (id bigint not null auto
Hibernate: create table MEMBER (ID bigint not null auto_in
Hibernate: create table user (USERNAME varchar(255) not nu
Hibernate: select member0_.ID as ID1_1_, member0_.AGE as A
Member count: 0
Hibernate: insert into MEMBER (AGE, FNAME, lastLogin, LNAM
Member inserted!
```

Hibernate 4 AUTO

Hibernate 5 IDENTITY

```
Hibernate: create table Authority (id bigint not null, auth
Hibernate: create table hibernate_sequence (next_val bigint
Hibernate: insert into hibernate_sequence values ( 1 )
Hibernate: insert into hibernate_sequence values ( 1 )
Hibernate: create table MEMBER (ID bigint not null, AGE int
Hibernate: create table user (USERNAME varchar(255) not nul
Hibernate: select member0_.ID as ID1_1_, member0_.AGE as AG
Member count: 0
Hibernate: select next_val as id_val from hibernate_sequenc
Hibernate: update hibernate_sequence set next_val= ? where
Hibernate: insert into MEMBER (AGE, FNAME, lastLogin, LNAME
Member inserted!
```

Hibernate 5 AUTO

```
Hibernate: create table Authority (id bigint not null, auth
Hibernate: create table hibernate_sequence (next_val bigint
Hibernate: insert into hibernate_sequence values ( 1 )
Hibernate: create table MEMBER (ID bigint not null auto_inc
Hibernate: create table user (USERNAME varchar(255) not nul
Hibernate: select member0_.ID as ID1_1_, member0_.AGE as AG
Member count: 0
Hibernate: insert into MEMBER (AGE, FNAME, lastLogin, LNAME
Member inserted!
```

Hibernate 5 native

@Column Annotation

@Column is optional

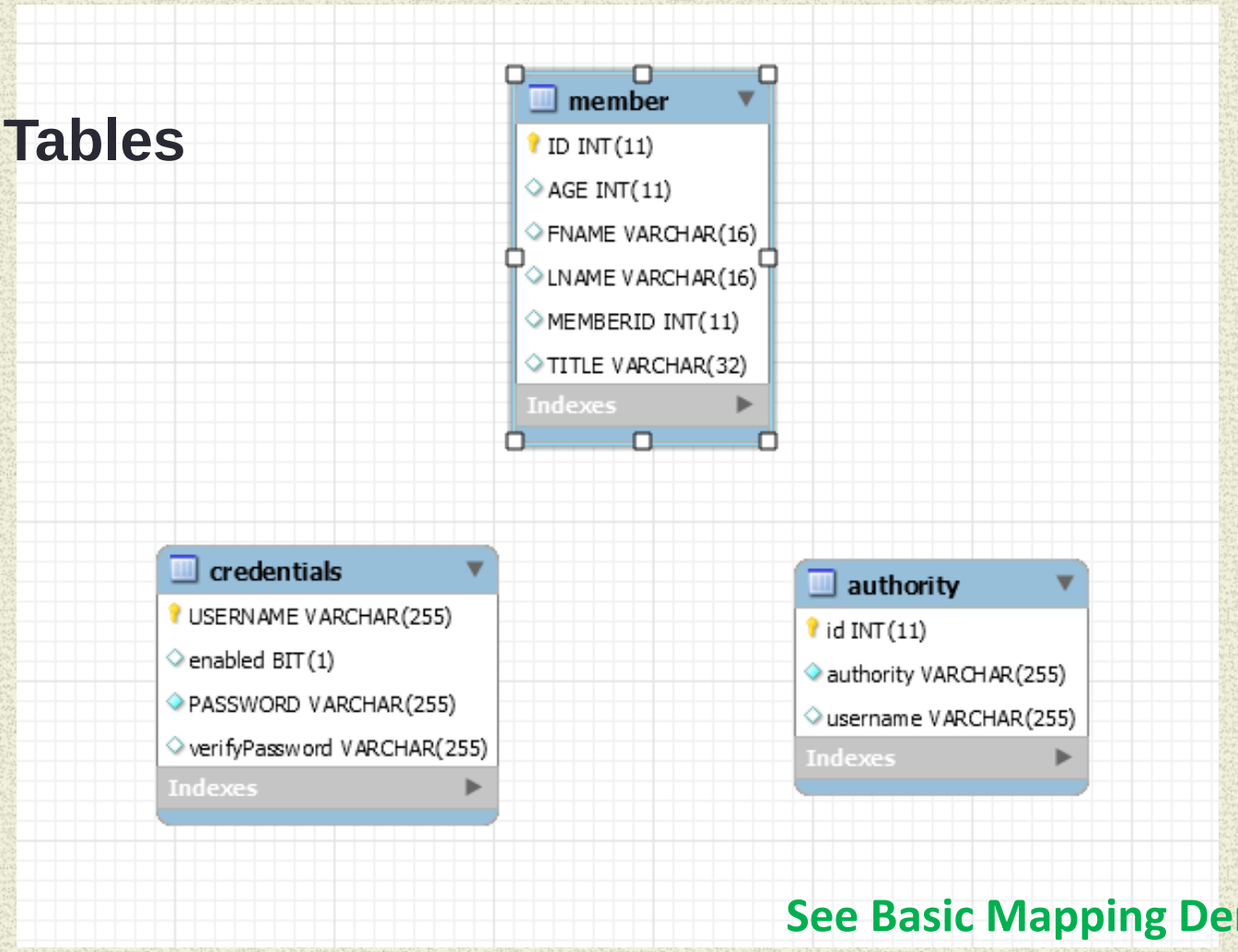
```
@Column(
    name="columnName";
    boolean unique() default false;
    boolean nullable() default true;
    boolean insertable() default true;
    boolean updatable() default true;
    String columnDefinition() default "";  //SQL to generate column
    String table() default "";
    int length() default 255;   // String
    int precision() default 0; // Decimal precision
    int scale() default 0; // Decimal scale
```

Default: field name

```
@Column(columnDefinition="Decimal(10,2) NOT NULL UNIQUE")
@Column(precision = 10, scale = 2, nullable = false, unique = true)
```

Class to Table Mapping Examples

Generated Tables



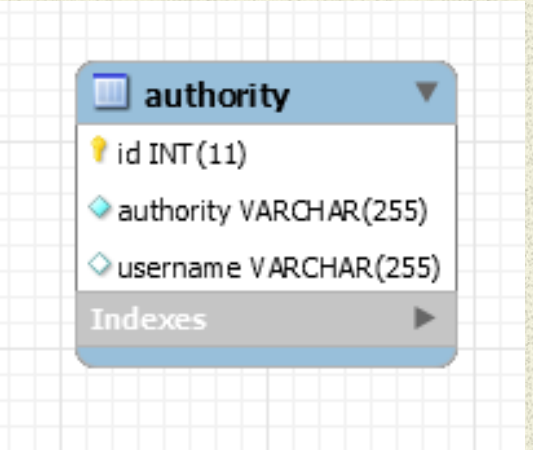
[Java - JDBC types]

See Basic Mapping Demo

Authority Example

Table = Class ; lower case first letter
Field name = column name
String default 255

```
@Entity
public class Authority {

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

private String username;
  @Column(nullable = false)
 private String authority;
```

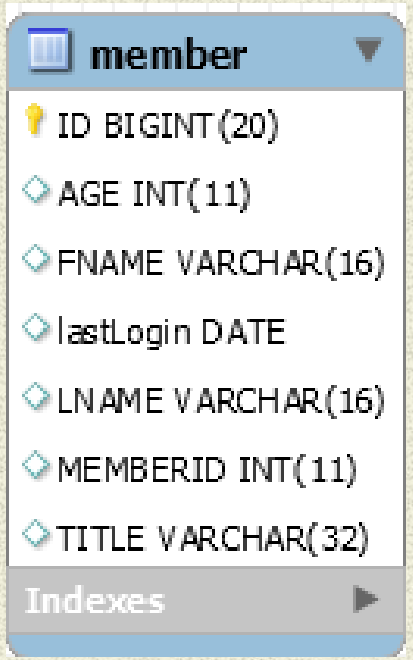


Column	Type	Nullable
id	int(11)	NO
authority	varchar(255)	NO
username	varchar(255)	YES

Member Example

```
@Entity(name = "MEMBER")
public class Member {

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

@Column(name="FNAME", length = 16)
private String firstName;
@Column(name="LNAME", length = 16)
private String lastName;
@Column(name="AGE")
private int age;
@Column(name="TITLE", length = 32)
private String title;
@Column(name="MEMBERID")
private int memberNumber;
@Temporal(TemporalType.DATE)
@Column
private Date lastLogin;
```

Long == BigInt
TABLE = Entity name
Explicit String length
Field name= Declared name



@Temporal

- @Temporal converts date and time values from Java object to compatible database type and retrieving back to the application.
- java.util.Date or java.util.Calendar require @Temporal to map to database types
- Not required when using java.sql.Date or java.sql.Time

]

- `@Temporal(TemporalType.DATE)` — *Same as:*
- `private java.util.Date lastLogin;` — `java.sql.Date lastLogin;`
- `@Temporal(TemporalType.TIME)` — *Same as:*
- `private java.util.Date lastLogin;` — `java.sql.Time lastLogin;`
- `@Temporal(TemporalType.TIMESTAMP)` — *Same as:*
- `private java.util.Date lastLogin;` — `java.sql.Timestamp lastLogin;`
- ***WITHOUT*** `@Temporal` — *Same as:*
- `private java.util.Date lastLogin;` — `java.sql.Timestamp lastLogin;`

UserCredentials Example

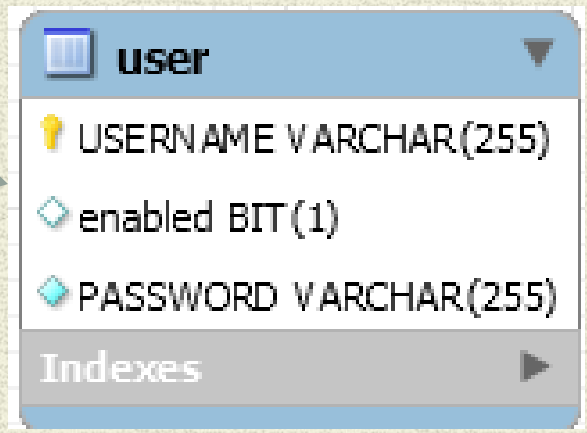
```
@Entity(name = "USER")
public class UserCredentials

 @Id
 @Column(name = "USERNAME", nullable = false, unique = true)
 String username;
 @Column(name = "PASSWORD", nullable = false)
String password;
@Transient
String verifyPassword;
Boolean enabled;
```

Entity/Table name different from Class name
ID – String - USERNAME - unique
Password not NULL

Key	Type	Unique	Columns
PRIMARY	BTREE	YES	USERNAME

@Column annotation can be optional

Column	Type	Nullable
enabled	bit(1)	YES
PASSWORD	varchar(255)	NO
USERNAME	varchar(255)	NO



Main Point

- The mapping of simple object structures to a database is done through configuration files and/or annotations. This simple configuration is enough to instruct the framework about the objects it has to control and store.

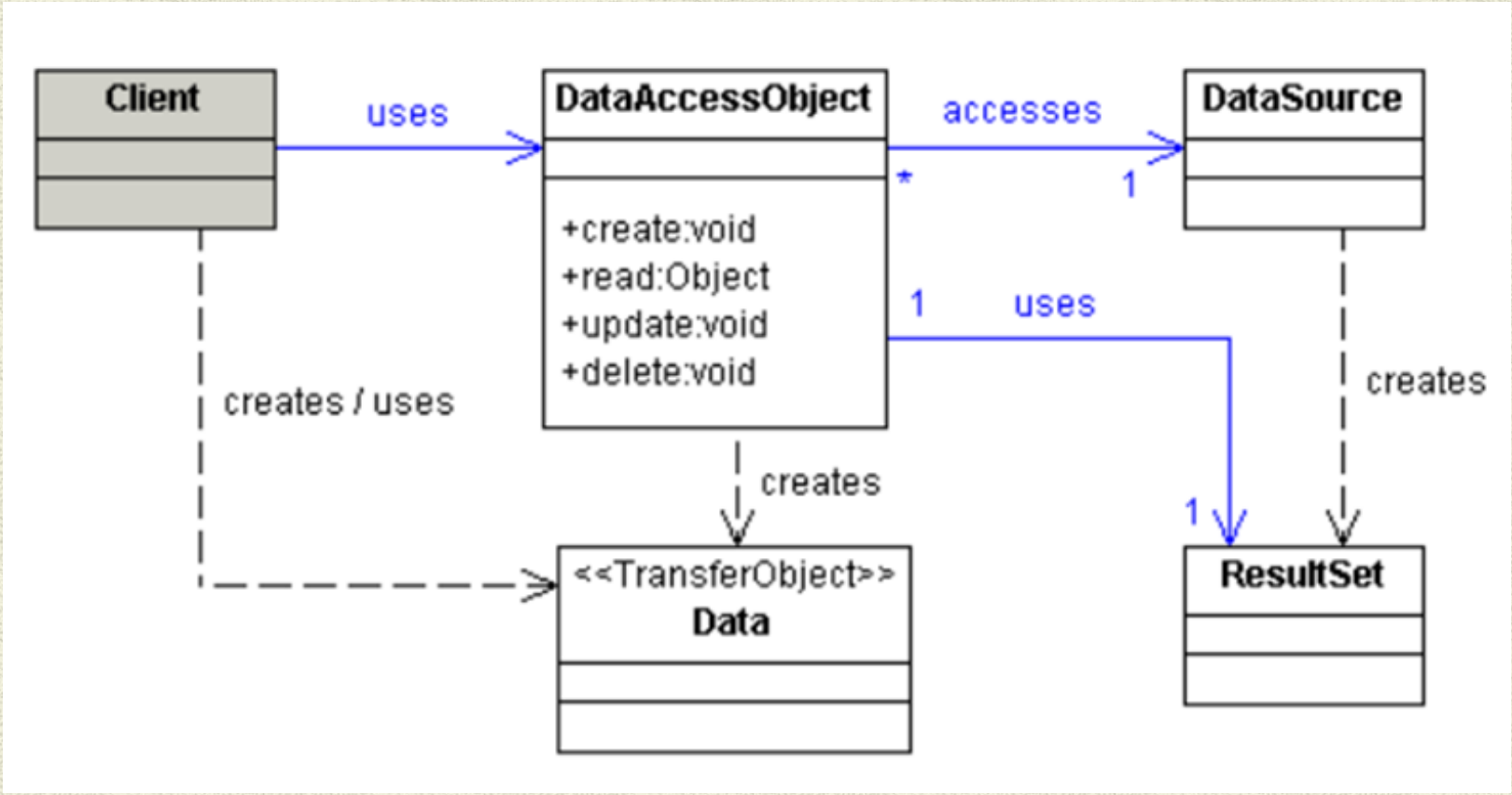
- ***Science of Consciousness:*** *The simple mechanics of the TM technique allow [instruct] the mind to transcend to the home [store] of all knowledge.*

CRUD Services
Core J2EE DAO pattern

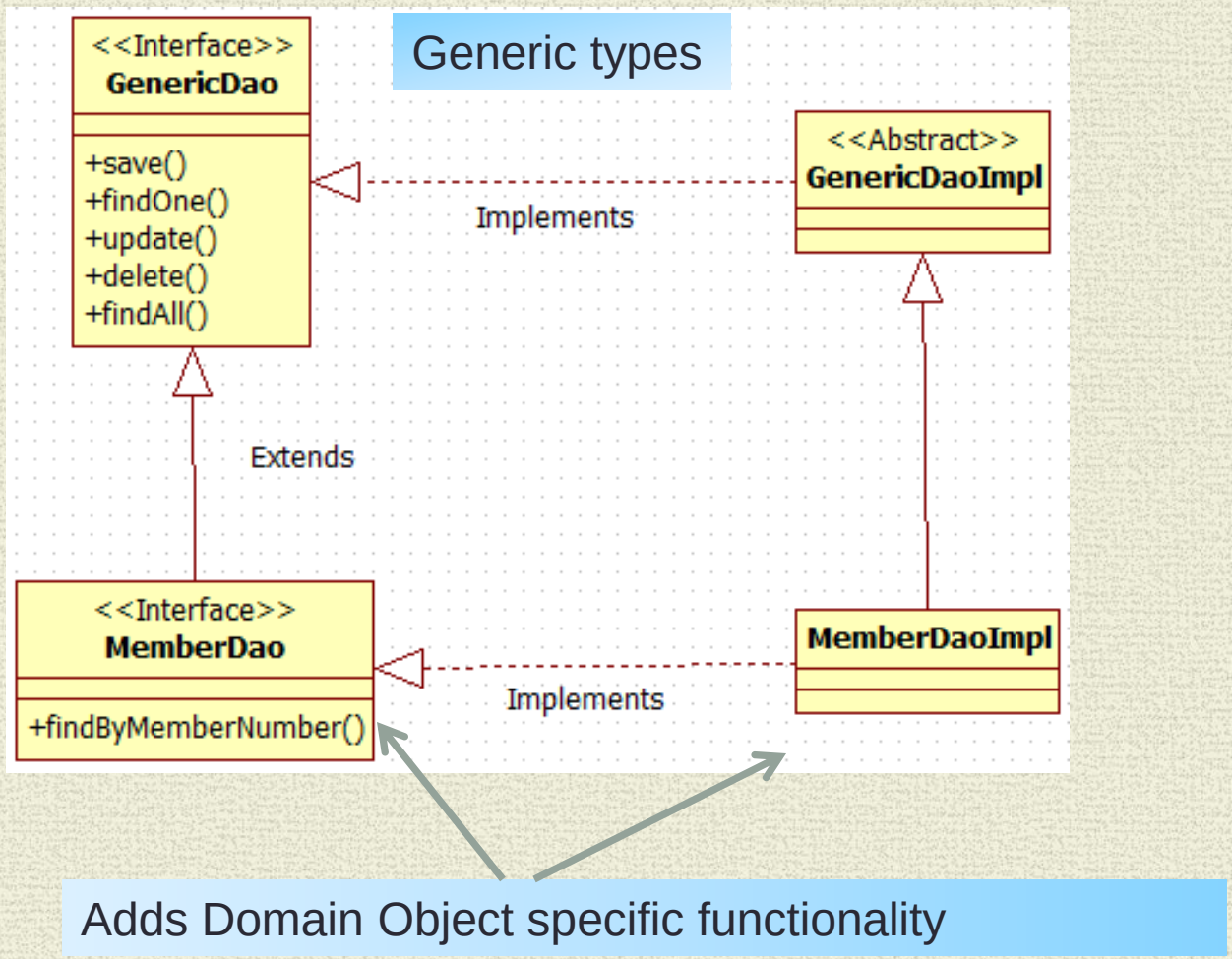
Data Access Object (DAO)

- 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.

DAO Interactions

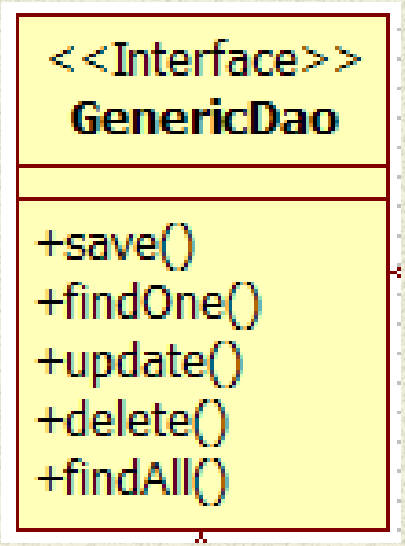


“Classic” ORM GenericDAO



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();
```



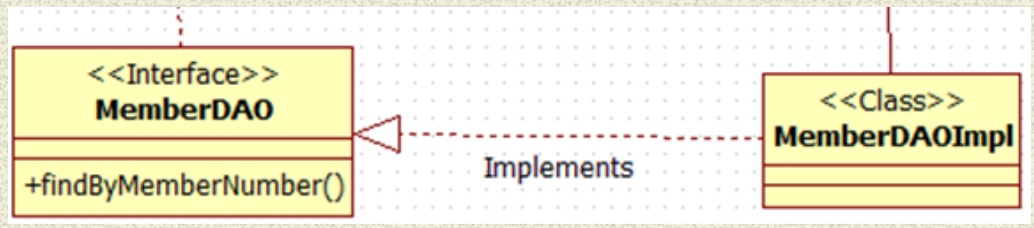
Generic DAO Implementation

```
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

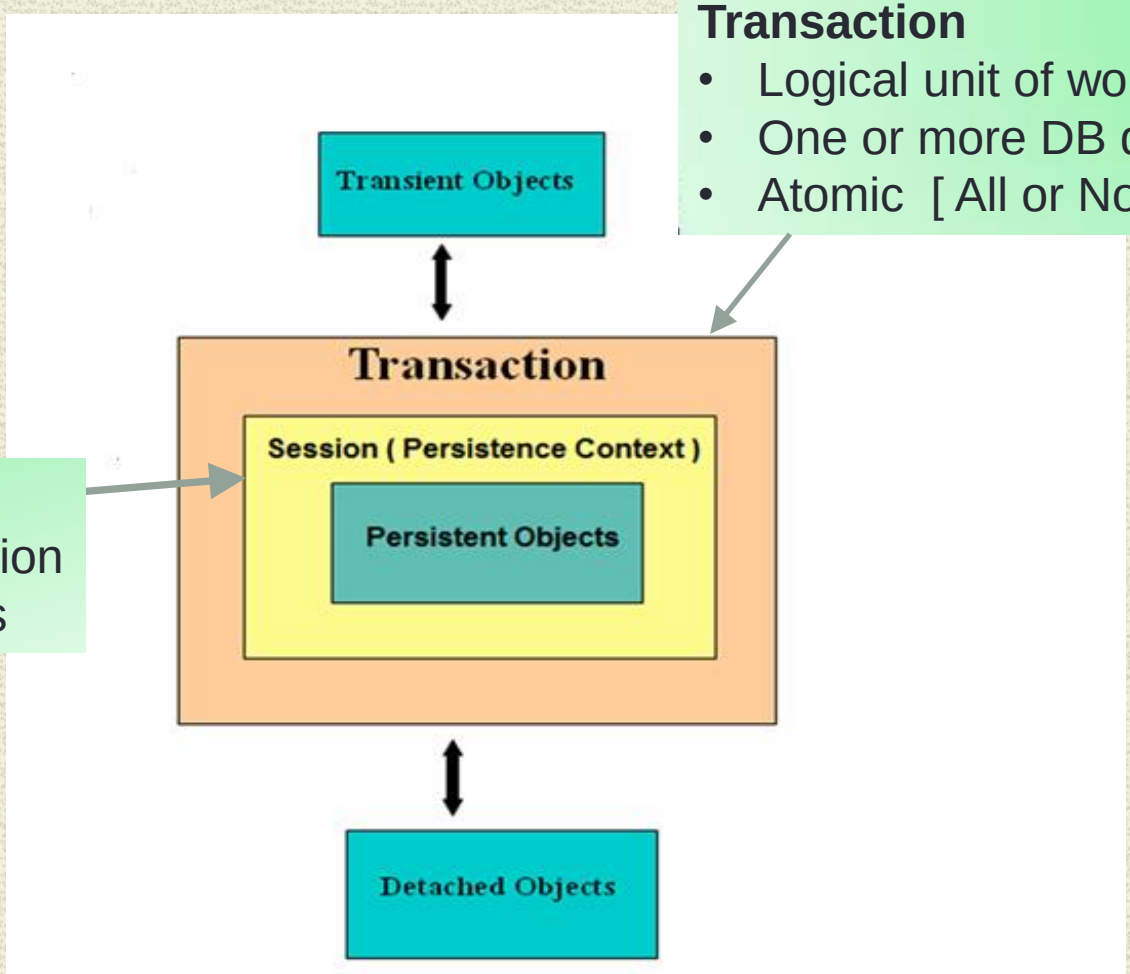
```
public interface MemberDao extends GenericDao<Member> {
    public Member findByMemberNumber(Integer number);
```




```
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();
}
```

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

Persistence Context ~= Hibernate Session

ORM Persistence Context

- Transaction Unit of work

  Spring "manages" through @Transactional

  Common Pattern: *session-per-request*

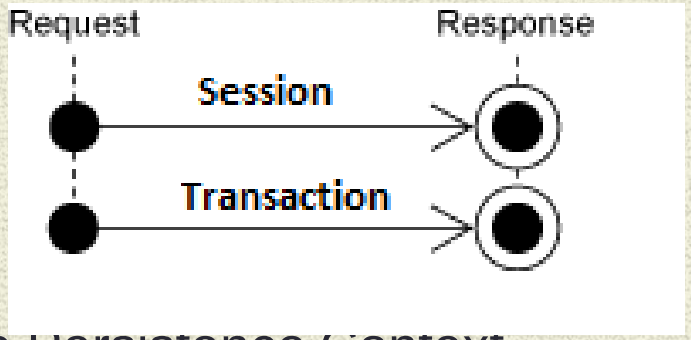
  Persistence Context == Database Transaction

- START –

  Open a Persistence Context

  Open a single database connection

  Start a Transaction



  Associate & Manage entities W/R the Persistence Context

  Exercise DB CRUD operations

- END –

  End Transaction

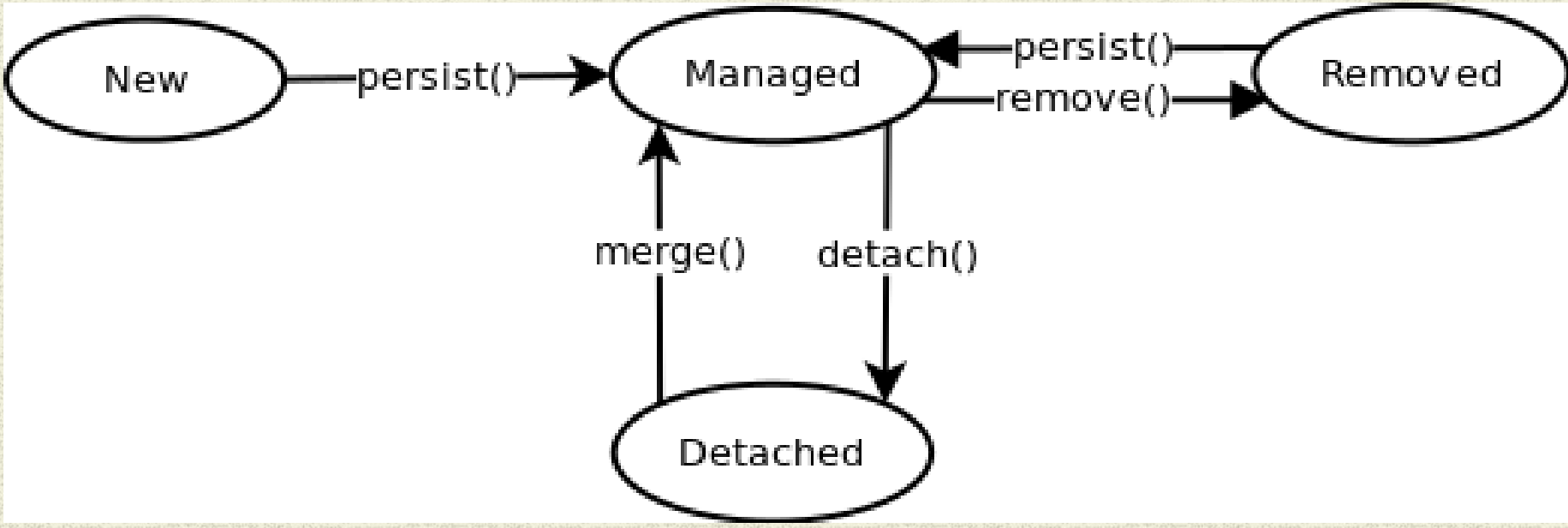
  Close a Persistence Context

ORM-related Entity States

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

ORM Entity Lifecycle

- **Transient** **Persistent** **Removed**



Detached

ORM caching mechanisms

Persistence provider manages local store of entity data

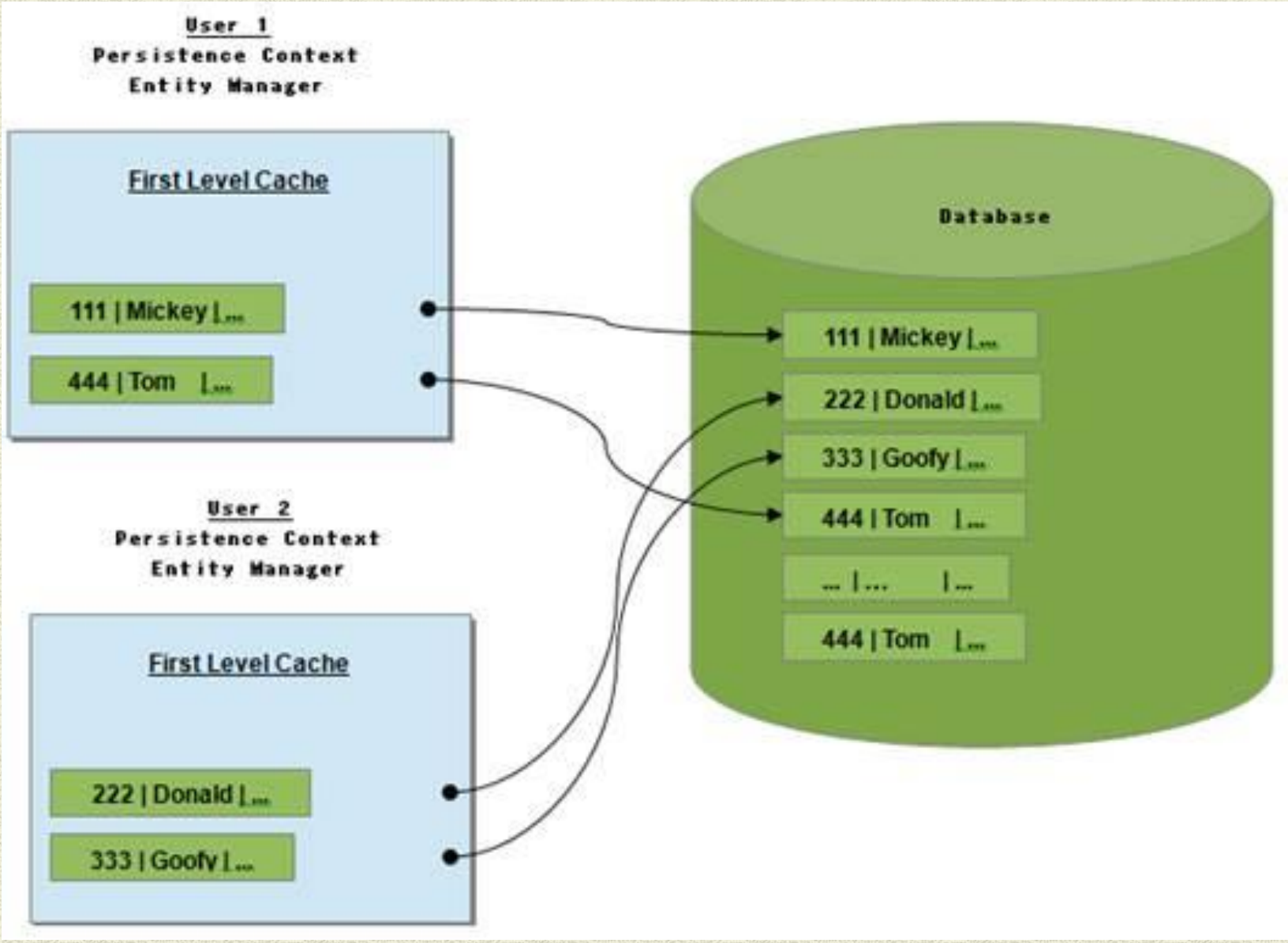
- Leverages performance by avoiding expensive database calls
-
- CRUD operation can be performed through normal entity manager functions
- Application can remain oblivious of the underlying cache and do its job without concern

- Level 1 Cache

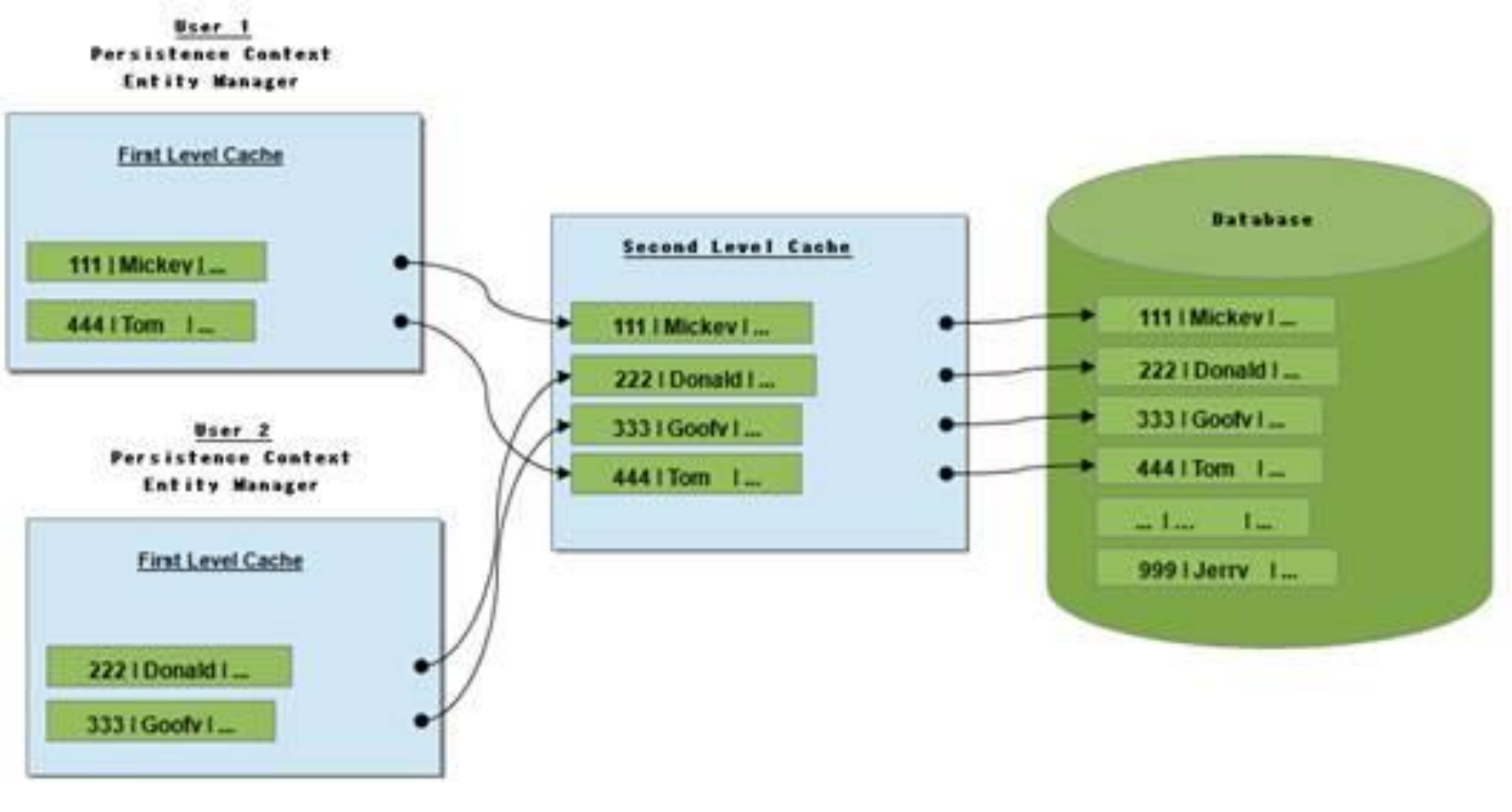
  Available within the same transaction [Persistence Context]
- Level 2 Cache

  Available throughout the application.

Level 1 Cache



Level 2 Cache



Managing the CRUD operations

CRUD	JPA
Create	Persist
Read	Find
Update	Merge
Delete	Remove
	Flush
	Refresh

Create == Persist

Same as Hibernate session.persist()

- Does not create SQL insert statement immediately
- The ***commit*** of the transaction sends insert to DB

EFFICIENT

- If you change multiple fields, add a relation while an entity is in managed state, only one update statement at the end.

Hibernate save – insert immediately less efficient
JPA has no such method

- Minimum row lock time

Hibernate **saveOrupdate** -- attaches the passed entity to the persistence context
ISSUE: If entity ALREADY in persistence context with the same ID,
a NonUniqueObjectException is thrown.
JPA has no such method

READ == FIND

Object fetch [through Level 1 cache]

That is, if Object is in cache return that value,

if NOT in cache, fetch Object from DB

Hibernate get() identical to JPA find

entityManager.findOne(Member.class,memberId);

READ [Lazy] == getReference()

Similar to Hibernate load()

- Doesn’t load the full object state - just gets a reference to it
- Only fetches from DB when an object field is referenced.
- NOTE: Reference **MUST** be done in the PersistenceContext

- **User user = new User();**
- **user.SetName("Henry");**
- **Member member=**

  entityManager.getReference(Member.class, Id);
- **user.setMember(member);**
- **em.persist(user);**

no query to the DB

Update == Merge

- Entity is in Detached state
- ORM fetches the ”current” entity from the database.
- ORM copies values of detached object to the fetched object
- The fetched object is **THE** managed object
- The detached object is **NOT** a managed object.
- Return/ operate on the fetched object.

NOTE: Calling **MERGE** on a **TRANSIENT** object will **PERSIST** it..

- **Member fetchedMember =**

entityManager.merge(detachedMember);

Hibernate has **update** which attaches the passed entity to the persistence context
ISSUE: If entity ALREADY in persistence context with the same ID,
a NonUniqueObjectException is thrown.

JPA has no such method

Update a managed object

- Results in a Implicit save/merge….at commit…

Open Session/Start Transaction

```
Member member = entityManager.find(Member.class, 4711);
member.setName(“Frank Lee");
```

End Transaction/Close Session

Member is UPDATED with name = “Frank Lee” in DB

Delete = Remove

- Need to Make entity managed[merge] BEFORE delete

- **Member mergedMember = entityManager.merge(member);**
- **entityManager.remove(mergedMember);**
-

A detached entity is possibly stale. You need to “synchronize” [merge] it with the DB before removing…

Flush

- Flush() sends SQL instructions to the database
  [e.g.,INSERT, UPDATE]
- Synchronizes the managed objects with the database

- FlushMode default is **AUTO**
- [**AUTO**] Flush occurs :

1. Before query execution
2. Transaction commit
3. EntityManager.flush() is explicitly called

- EntityManager.setFlushMode(FlushModeType.COMMIT)
  disables flush before query execution {#1 above}

Refresh

- Refresh will reread the object from the database.
- **USE CASE:**

  Database triggers are used to initialize some of the properties of the entity

  …Can also be used to undo updates

- **em.persist(member);**
- **em.flush(); // force the SQL insert and triggers to run**
- **em.refresh(member);**

Main Point

1. The persistence framework has a set of simple common operations . We simply configure and use them improving flexibility, overall accuracy and performance of the system.
2. ***Science of Consciousness****: Research found that participants in the Transcendental Meditation program showed greater activation of the appropriate hemisphere of the brain. This means the brain responds more flexibly and dynamically.*