ASSOCIATIONS MANAGING THE IMPEDANCE MISMATCH

ORM Impedance Mismatch

2 Different Technologies – 2 different ways to operate

EXAMPLE

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

Many-to-many relationships

Inheritance

Collections

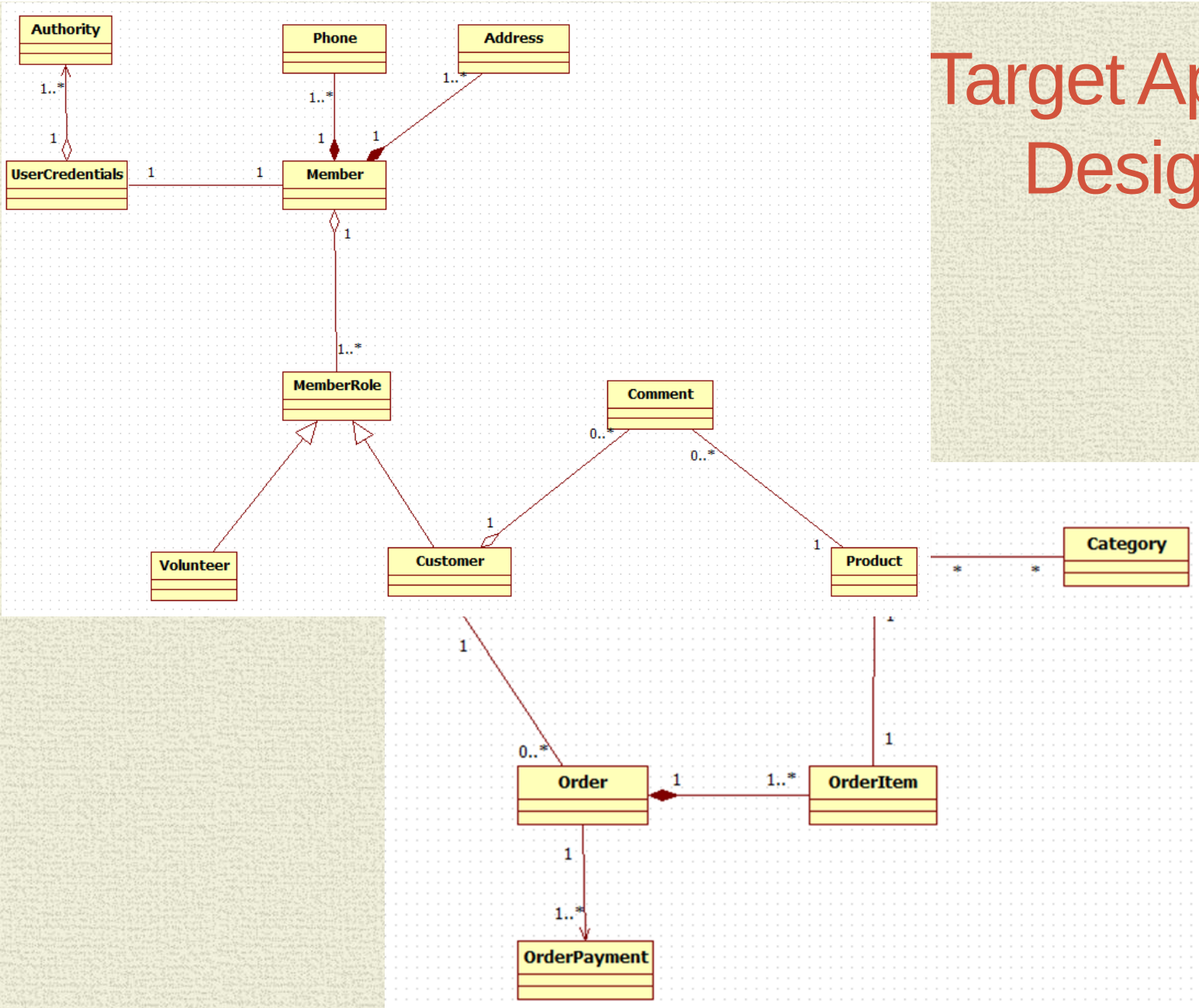
Identity [Primary Key .vs. a.equals(b)]

Foreign Keys

Bidirectional [“Set both sides”]

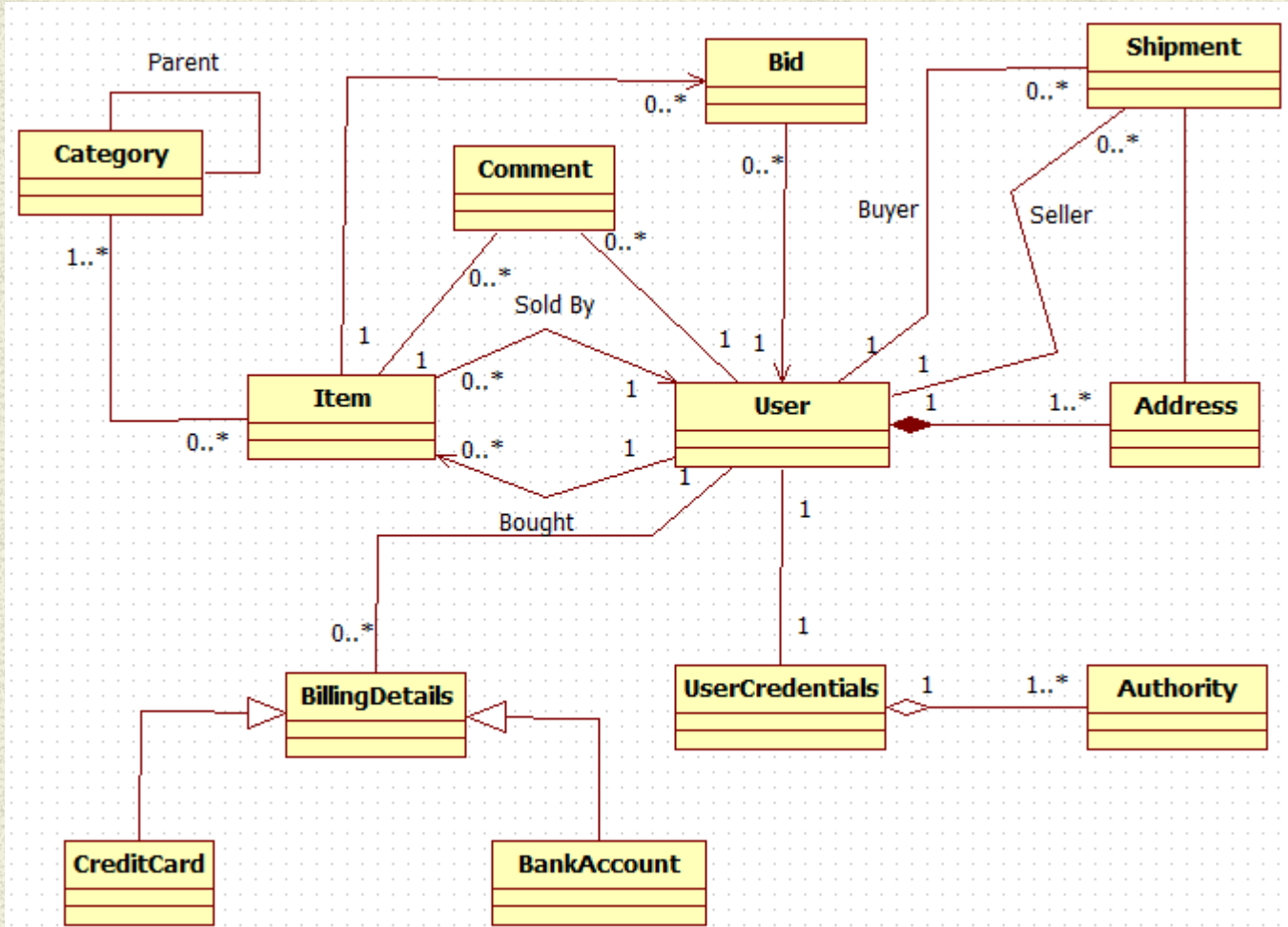
Granularity [# of Tables .vs. # of Classes]

Target Application Design Model



Hibernate Caveat Emptor

Student Target Application Design Model



Member Entity Example

```
@Entity
public class Member {
@Id
@GeneratedValue(strategy=GenerationType.AUTO)
@Column(name="member_id")
private Long id;
@Column(length = 16)
private String firstName;
@Column(length = 16)
private String lastName;

. . .
@OneToOne(fetch=FetchType.LAZY,  cascade = CascadeType.ALL)
@JoinColumn(name="username")
 UserCredentials userCredentials;

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

Configurable Parent-Child operations JPA Cascade Types

- **PERSIST**

  Cascading calls to EntityManager.persist() - persists children.

- **MERGE**

  Cascading calls to EntityManager.merge() - updates children.

- **If orphanRemoval=true** then a disconnected child is automatically removed.
  e.g., userCredentials.getAuthority.remove(0);
  userCredentialsDao.update(userCredentials);

- *Use: Composition relationship[Child does **NOT** exist without Parent relationship]*
  see OneToManyUniCol Demo

- **ALL**

  Shortcut for cascade={***PERSIST, MERGE, REMOVE, REFRESH***}

Example:

```
@OneToMany(cascade = CascadeType.PERSIST)
```

Configurable Parent-Child operations Fetching Strategies

- *Immediate fetching*: an association, collection or attribute is fetched immediately when the owner is loaded.

[(JPA)Default for one-to-one]

```
@OneToOne(fetch=FetchType.EAGER)
```

- *Lazy collection fetching*: a collection is fetched when the application invokes an operation upon that collection.

[Default for collections]

```
@OneToMany(fetch=FetchType.LAZY)
```

Configurable Parent-Child operations Fetch - Cascade Example

- **public class Member{**
- `@OneToMany(mappedBy="member",fetch=FetchType.LAZY,`
  `cascade={CascadeType.PERSIST, CascadeType.MERGE })`
- `private Set<Address> addresses = new HashSet<Address>();`

- `FetchType.LAZY` means collection is NOT fetched until collection element is referenced…

***It also means that you will get a* LazyInitializationException**

If you try to reference it after the PersistenceContext is closed!!

- ***CascadeType[s]*** means collection is persisted or merged when parent is.

Main Point

1. A good ORM provides features that allow the developer to easily traverse object relationships.
2. ***Science of Consciousness:*** *When we practice the TM Technique, we tap an inner reserve of energy and intelligence that allows us to easily and flexibly manage the diverse activities of every day life*

OrganizingTest Data
Object Mother

Set of factory methods that create test data objects

- *Customer customer = CreateTestCustomer();*

Becomes

- *Customer customer = CustomerObjectMother.CreateCustomer();*

Issue

- *Customer customer = CustomerObjectMother.CreateWashingtonBasedCustomer();*

Names could become Long & Ambiguous depending on the number of variables

Versus

- *Customer customer = CustomerObjectMother.CreateCustomer();*
- *customer.setState("WA");*

More data clarity at test level: Can see test data inline

Organizing Test Data
Test Data Builder

Based on Builder Pattern

[***Solution to the telescoping constructor anti-pattern***]

[Also explicitly shows the data that affect the outcome of the test]

Reduces the number of constructors, by processing initialization parameters step by step

- *Customer customer = new CustomerBuilder()*
- *.withState("WA")*
- *.withZipCode("98765")*
- *.withFirstName("Fred")*
- *.build();*
- ….***We'll use this pattern for our Association Demos***….

Telescoping Constructors

```
public Product(Long id, String name) {
        this(id,name,"", 0.0,null);
    }

public Product(Long id, String name, String Description) {
        this(id,name,description, 0.0,null);
    }

public Product(Long id, String name, String Description,Float price) {
        this(id,name,description, price,null);
    }

 public Product(Long id, String name, String Description,Float price,Category category) {
        this.setId(id);
        this.setName(name);
            ...
    }
```

Difficult to identify parameters of the same type
Often there isn't a constructor that fits your need
So, either add a new constructor or use a null parameter.

Spring Testing Support

- Testing is an integral part of enterprise software development.
- Use of IoC makes unit and integration testing easier
- Spring provides first-class support for integration testing
- **SpringJUnit4ClassRunner** provides the functionality of the ***Spring TestContext Framework:***

  Consistent loading & caching of Spring ApplicationContexts

  Creation and roll back of transactions for each test

- **@ContextConfiguration** declares the application context resource locations[XML] or the annotated classes[JavaConfig] that will be used to load the context for integration tests
- Spring Testing
- **See OneToOneUni Junit Test**

ORM Parent-Child "Relationships"

One-to-One

One-to-Many/Many-to-One

Many-to-Many

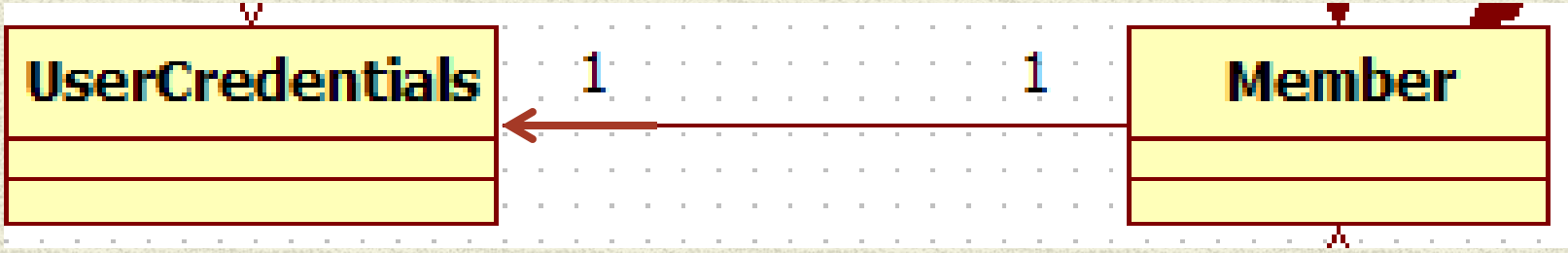
Unidirectional – Bidirectional

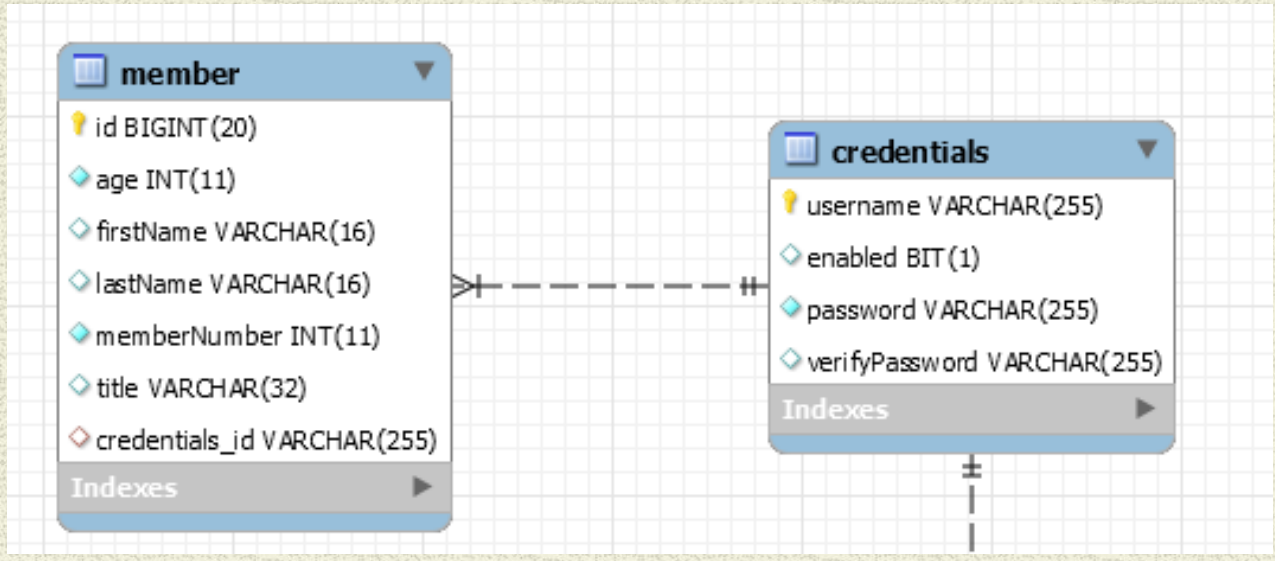
Inheritance

OneToOne Unidirectional

Member.Java

```
@OneToOne(fetch=FetchType.LAZY, cascade={CascadeType.PERSIST,CascadeType.REMOVE})
@JoinColumn(name="credentials_id")
private UserCredentials userCredentials;
```





DEMO OneToOneUni

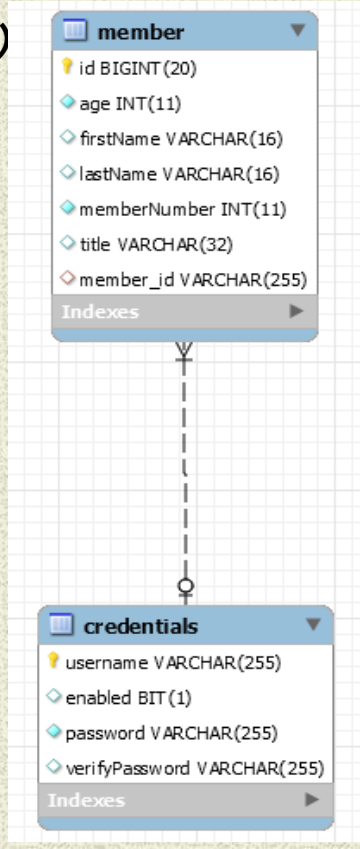
OneToOne Bi-directional

Annotate the OTHER side of the relationship ALSO…

```
@Entity
 public class UserCredentials  {
            @Id
            private long id; ...
• @OneToOne(mappedBy="userCredentials",cascade=CascadeType.PERSIST)
•   private Member member;

}
```

mappedBy – Is used in a bidirectional relationship
To identify the foreign key in the owning side of the relationship



DEMO OneToOneBI

Collections

- Mapping & performance are ORM specific
- **Hibernate Collection Categories**

Indexed

Map - unordered, no duplicates

List [indexed] – ordered, duplicates

Set

Unordered, no duplicates

Bag*

Unordered, duplicates

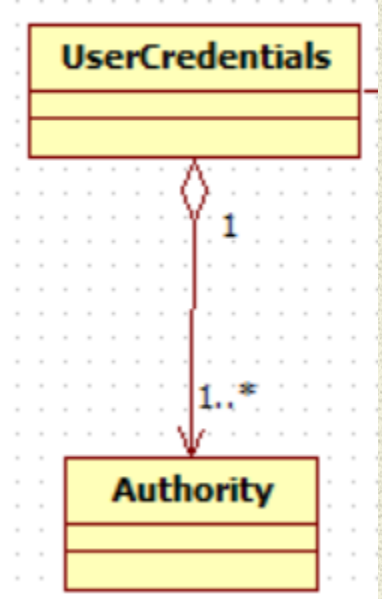
*Bag is a Hibernate artifact – it represents a NON-indexed List

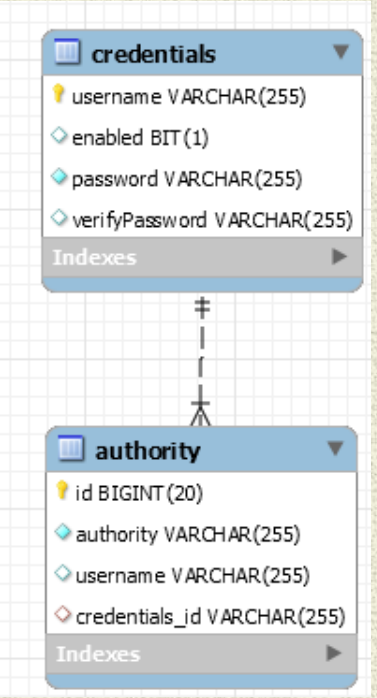
-

OneToMany Unidirectional JoinColumn

```
@Entity
public class Credentials {
    @Id
    private long id; ...

  @OneToMany
  @JoinColumn(name="credentials_id")
   private List<Authority> authorities; ...
}
```





HIBERNATE REFERENCE DOC:
A *unidirectional one-to-many association on a foreign key* is an unusual case, and is not recommended. You should instead use a join table for this kind of association.

DEMO OneToManyUniCol

Indexed List

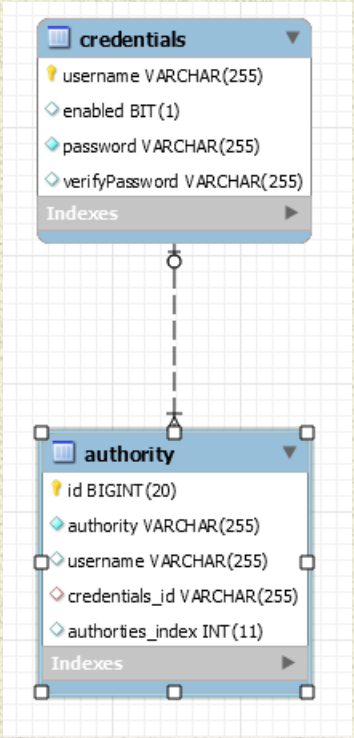
By Default LIST is NOT ordered in Database.

To create an Indexed List

@OrderColumn - put index field in target[Child/Many] table ….

```
@OneToMany(fetch = FetchType.EAGER, cascade = CascadeType.ALL)
    @JoinColumn(name="credentials_id")
    @OrderColumn(name = "authorties_index")
List<Authority> authority = new ArrayList();
```

id	authority	username	credentials_id	authorties_index
1	ROLE_USER	JohnDoe	JohnDoe	0
2	ROLE_ADMIN	JohnDoe	JohnDoe	1
3	ROLE_SUPERVISOR	JohnDoe	JohnDoe	2
NULL	NULL	NULL	NULL	NULL



DEMO OneToManyUniCol

NOTE the # of updates for Orphan Removal

Map

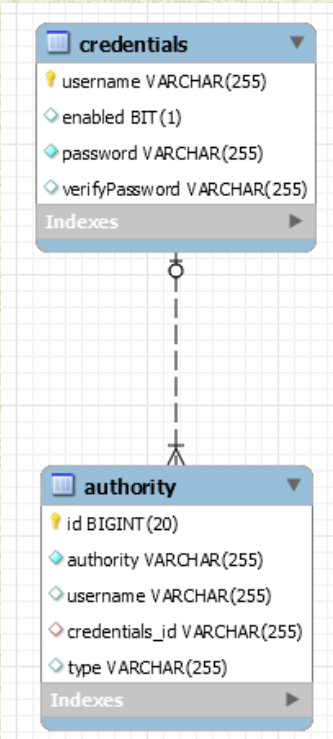
- By Default MAP is NOT ordered in Database.

To create an Map

- @MapKey – uses existing field in target[Child/Many] table ….
- @MapKeyColumn – creates column in target[Child/Many] table ….

```
@OneToMany(fetch = FetchType.EAGER, cascade = CascadeType.ALL)
    @JoinColumn(name="credentials_id")
    @MapKeyColumn(name = "type")
Map<Authority> authority = new HashMap();
```

id	authority	username	credentials_id	type
1	ROLE_ADMIN	JohnDoe	JohnDoe	admin
2	ROLE_USER	JohnDoe	JohnDoe	user
3	ROLE_SUPERVISOR	JohnDoe	JohnDoe	supervisor

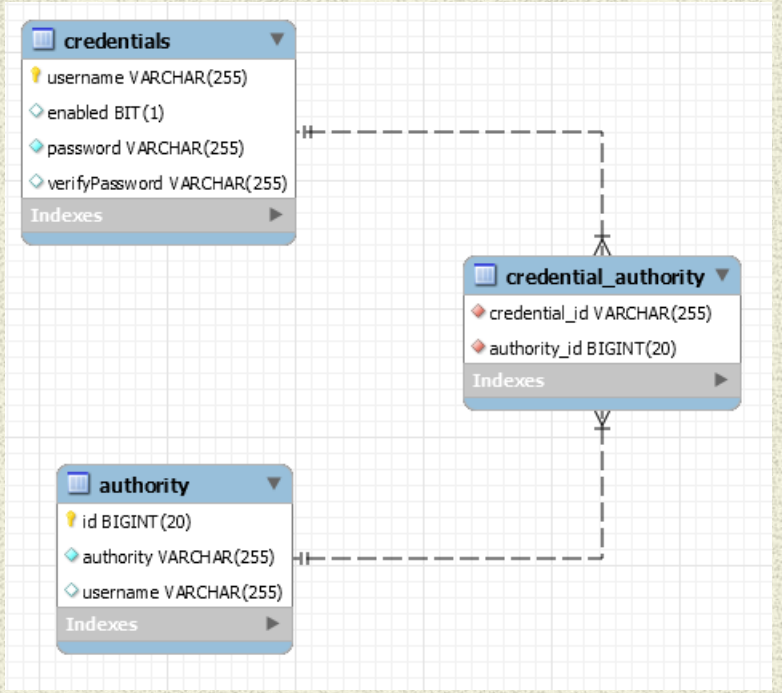


- **DEMO OneToManyUniMap**
- **See demo for query by KEY()**

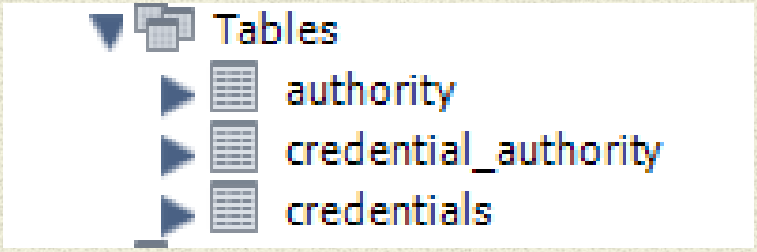
OneToMany Unidirectional JoinTable

```
@Entity public class Credentials{
   @Id
   private long id; ...

@OneToMany
private      Set<Authority> authorities;
```



DEMO OneToManyUniTable



OneToMany Bi-directional JoinColumn

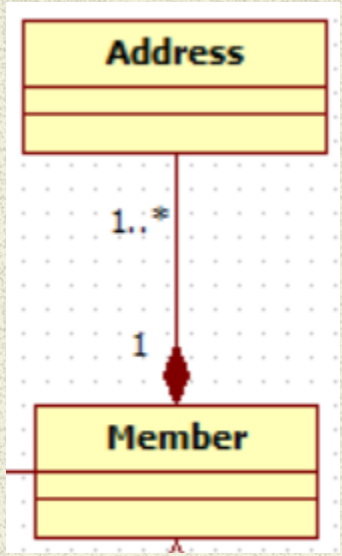
```
@Entity
public class Member {
  @Id
  @Column(name="member_id")
   private Long id; ...
  @OneToMany(mappedBy="member)
  private List<Address> addresses;

}
```

```
@Entity
public class Address {
  @Id
 private Long id;

@ManyToOne(fetch=FetchType.EAGER)
 @JoinColumn (name="member_id")
  private Member  member;}
```

Owns relationship



NOTE: JoinColumn OPTIONAL

Bidirectional DEFAULTS to Join Column

DEMO OneToManyBiCol

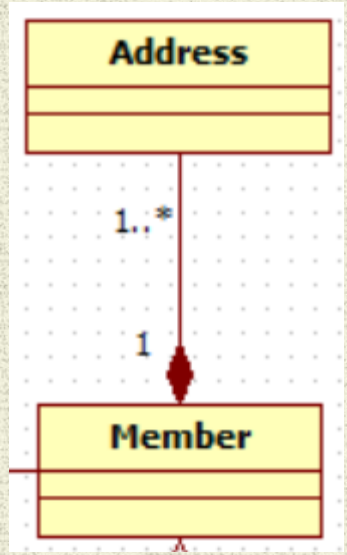
OneToMany Bidirectional JoinTable

OneToMany side same as unidirectional example only add mappedBy

```
@Entity
public class Member {
  @Id
  private long id;
  @OneToMany (mappedBy="member")
  private      List<Address> addresses;
}
```

Simply Add ManyToOne on child object PLUS Identify JoinTable by Name

```
@Entity
public class Address {
…
@ManyToOne
@JoinTable ( name="Address_Member")
private Member member;
```



DEMO OneToManyBiTable

Bidirectional JoinTable

Explicitly naming the Join Table & Columns

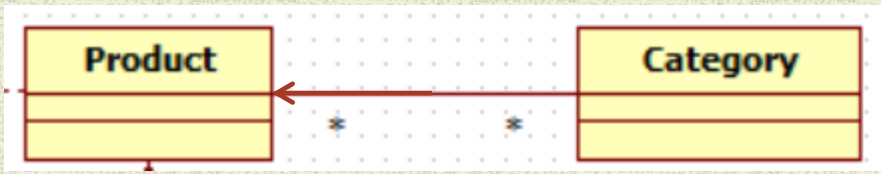
```
@Entity
public class Address{
  @Id
 private long id;

   @ManyToOne(fetch=FetchType.EAGER, cascade = {CascadeType.PERSIST})
   @JoinTable(name="AddressMember",joinColumns={@JoinColumn(name="ADDRESS")},
              inverseJoinColumns={ @JoinColumn(name="Member")} )

 private Member  member;
```

inverseJoinColumns identifies “opposite” or member side

Many-To-Many Unidirectional




```
@Entity
public class Category{
   @Id
    private long id;
@ManyToMany
   private      Set<Product> products;
```

Explicitly named JoinTable:

Similar to OneToMany – Explicitly naming the Join Table & Columns

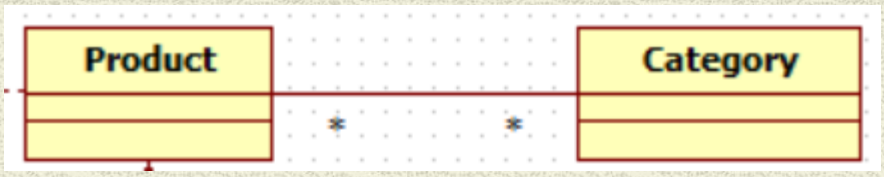
---

```
@ManyToMany(fetch = FetchType.EAGER, cascade = CascadeType.ALL)
    @JoinTable ( name="Category_Product",
                 joinColumns={@JoinColumn(name="Category_ID")},
                 inverseJoinColumns={ @JoinColumn(name="Product_ID")} )
    Set<Product> products = new HashSet<Product>();
```

- **NOTE:** If Converting from OneToMany Join Table – simply drop the unique constraint on the JoinTable created by OneToMany in the Database

DEMO ManyToManyUni

Many-To-Many Bidirectional




```
@Entity
public class Product {
   @Id
    private long id;
@ManyToMany(fetch=FetchType.EAGER,cascade={CascadeType.MERGE},mappedBy="products"
private List<Category> categories = new ArrayList<Category>();
```

Add Category reference to Product
To make it bidirectional

one direction must be defined as the *owner* and the other must use the mappedBy attribute to define its mapping.
OTHERWISE
There will be two independent relationships, duplicate rows will be inserted into the join table

DEMO ManyToManyBi

SET BOTH SIDES of a Bidirectional Relationship

- You can end up saving one side without ever setting the foreign key in the “OWNER” side [OneToMany,OneToOne]
- In a ManyToMany the side referred to by the mappedBy attribute, is the owner of the relationship. BUT there is no foreign key so what does that mean ??!!??
- **It Means:**
- **Good Practice**: Always encapsulate setting both sides….with a convenience method…
- DEMO EXAMPLES: **OneToManyBiCol [Member]**
  ManyToManyBi [Product]

Bidirectional Considerations

Domain Driven Design:

Reduce complexity by identifying a traversal direction **
(RE: avoid bidirectional associations if possible)

Removes coupling in domain model

Simplifies code in domain model

Removes circular dependencies

** Traversing the other direction is still possible by querying the underlying persistence system.

See demo OneToManyBiAsUni

Inheritance

Single Table

Contains all columns for Super Class & ALL Sub Classes. De-normalized schema. Efficient queries. Difficult to maintain as the number of columns increase.

Joined Tables [Table Per Subclass]

Normalized schema. Less efficient queries.

Effective if hierarchy isn't too deep.

Table per Class

Super Class is replicated in each subclass table.

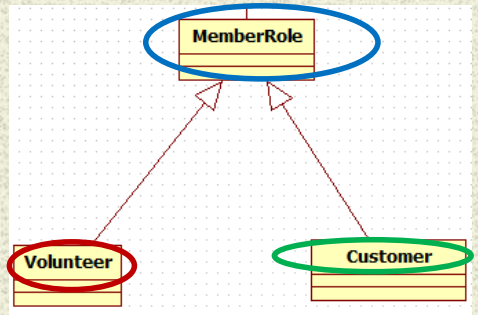
JPA - Optional

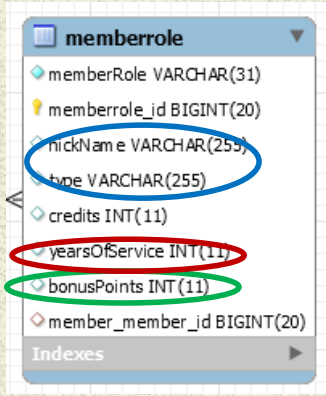
Three ways to map

Legend
Blue=MemberRole
Red=Volunteer
Green=Customer

- You can map inheritance in one of three ways:
  - Single Table [per Hierarchy]
    - De-normalized schema
    - Fast polymorphic queries

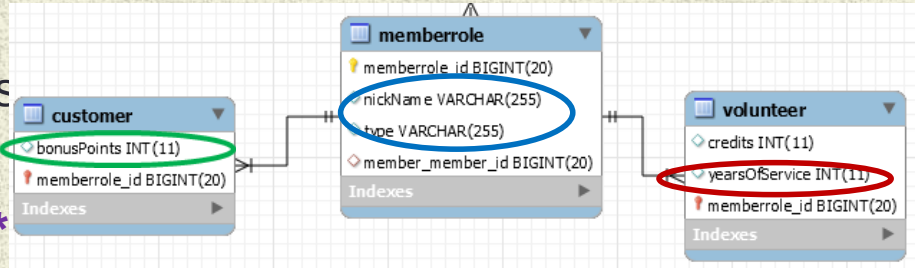
  DEMO Singletable





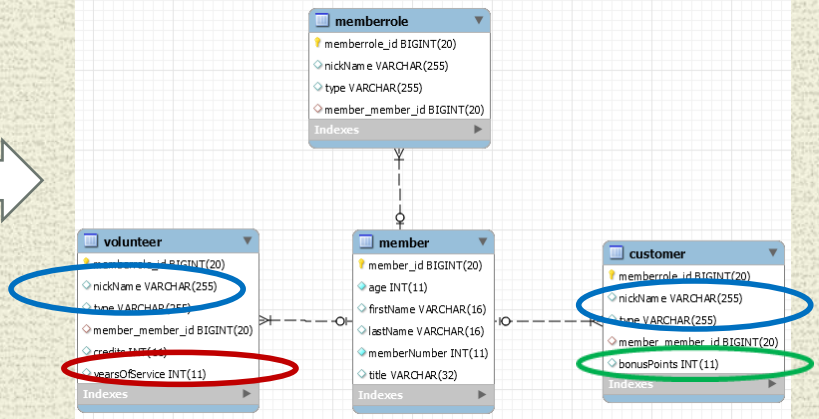
  - Joined Tables
    - Normalized & similar to OO classes
    - Slower queries

  Good if following GoF Patterns*



  DEMO TablePerSubClass

  - Table per [Concrete] Class
    - Uses UNION instead of JOIN
    - All needed columns in each table
    - **DEMO TablePerClass**

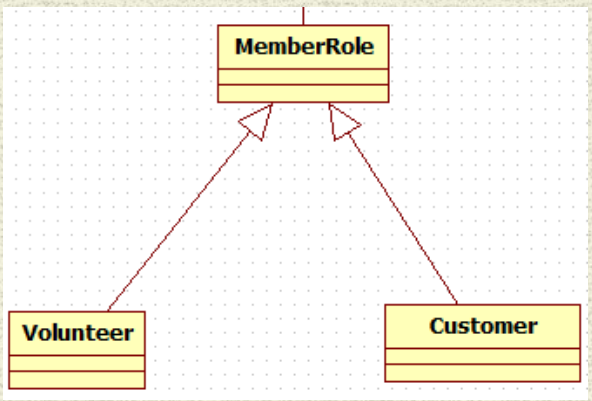


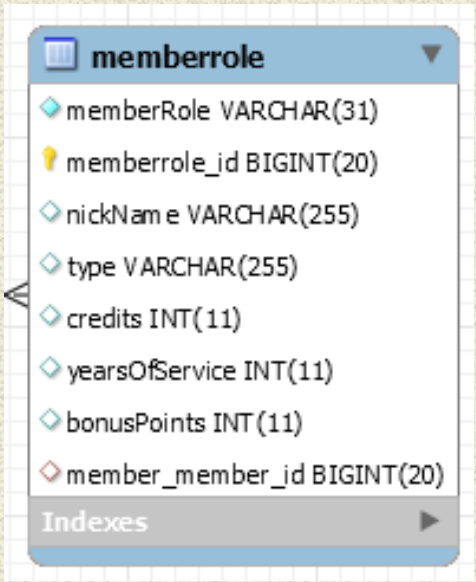
Mapping for Single Table

Single Table [per Hierarchy]

De-normalized schema

Fast polymorphic queries



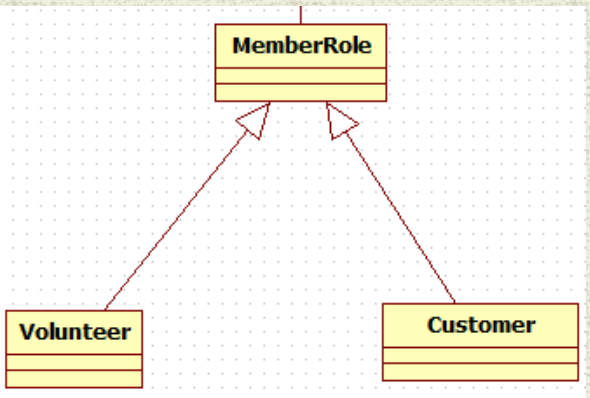


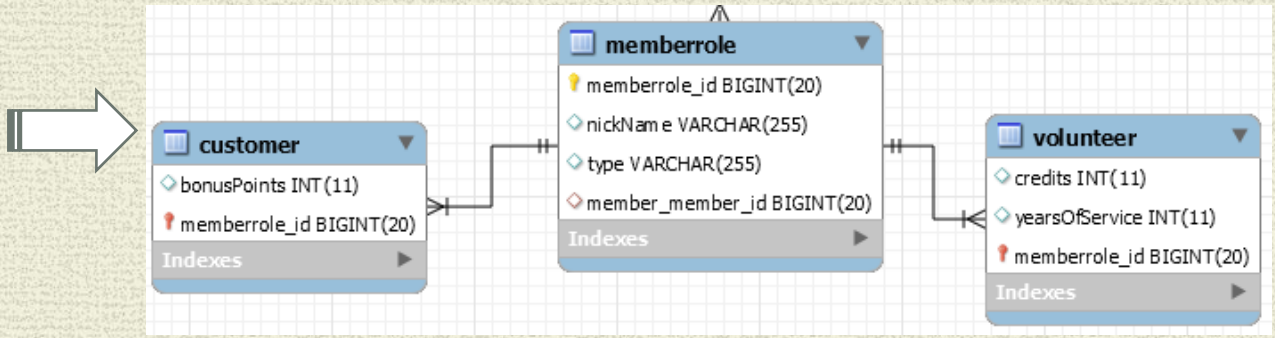
DEMO SingleTable

Mapping for Table Per SubClass

- Implemented by Joining Tables
  - Normalized & similar to OO classes
  - Slower queries

Good if following GoF Patterns*

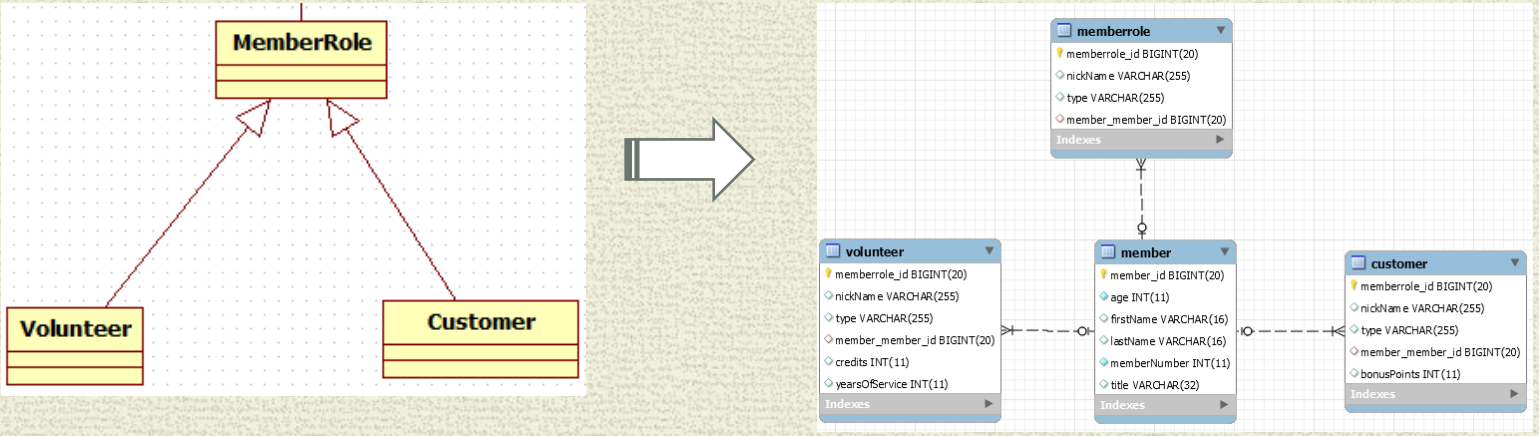




DEMO TablePerSubClass

Mapping for Table Per Class

- Table per [Concrete] Class
  - Uses UNION instead of JOIN
  - All needed columns in each table



DEMO TablePerClass

Miscellanea

- **Shared Primary Key**
- **USE CASE:** OneToOne where table is split into two tables resulting into same primary key
- **Composite key**
- **USE CASE:** Legacy database : database key is comprised of several columns
- **@SecondaryTable**
- **USE CASE:** Legacy database entity data stored in multiple tables

@Embeddable/Embedded

USE CASE: Legacy database HUGE table with “logically” separate columns [e.g. Address inside Customer]

Main Point

Entities and objects relationships can be established through the different types of associations, creating a rich foundation that can represent a real world domain.

Science of Consciousness: *Seek the highest first, start with a good foundation and build rich relationships upon it.*

Enumerations

- Using the Hibernate *@Enumerated* annotation

  Use *EnumType.ORDINAL* or *EnumType.STRING* to map the enum value to its database representation either as order number OR name.

  It Has weaknesses

- @Enumerated(EnumType.ORDINAL) store order number

  ISSUE: Removing or adding enum value in the middle or rearranging them will break existing records.

- @Enumerated(EnumType.STRING) stores enum name

  ISSUE: Renaming an enum value will break existing records

- **SOLUTION: JPA Converter**

  Provides custom conversion from Java representation to database

  Use an "Alias" representation of the enum values in the database

@Converter (*autoapply=true*) - convert all entity attributes of the given type.

```
public class ProductionStatusConverter implements
                        AttributeConverter<ProductionStatus, String> {
```

See ManyToManyUni Demo