JPA Mapping I

"Code with Passion!"

Topics

- Entity Relationships
- Directionality
- Cardinality
- Inheritance (will be covered in “JPA Mapping II”)

Entity Relationships

Aspects of Entity Relationships

- Directionality
  - Uni-directional
  - Bi-directional
- Cardinality relationships
  - One to one
  - One to many
  - Many to many
- Inheritance relationship
  - Single-table
  - Joined-table

Entity Relationships: Java vs. Table

- Relationship between entities in Java code is normal object relationship
  - Inheritance
  - Association
- Relationship between tables in relational database can be represented in one of the two schemes
  - Foreign key
  - Join table
- You can control how the object relationship between Java objects should be represented in tables through JPA annotations

Directionality

Directionality & Navigation

- Directionality affects the navigation

- Uni-directional
  - *customer.getAddress()* is allowed but *address.getCustomer()* is not supported for one-to-one relationship
  - *customer.getOrders()* is allowed but *order.getCustomer()* is not supported for one-to-many relationship
  - *speaker.getEvents()* is allowed but event.*getSpeakers()* is not supported for many-to-many relationship

- Bi-directional
  - Navigations on both direction are supported

Directionality & Ownership

- In a Relationship, there is an aspect of “ownership”
  - Who owns the relationship affects how tables are created
  - One side is called “owner of the relationship” and the other side is called “inverse-owner of the relationship”
  - The side who owns the relationship (owner) has the foreign key field

- Uni-directional
  - Ownership is implied
  - Example: *customer.getAddress()* is allowed but *address.getCustomer()* is not supported for one-to-one relationship – The “customer” table has the foreign key field and is the owner

- Bi-directional
  - Ownership needs to be explicitly specified meaning which side has the ownership

Cardinality Relationships

Cardinality Entity Relationships

- Cardinality relationships
  - @*OneToOne*: A Customer has a single Address
  - @*OneToMany*, @*ManyToOne*: A Customer has many Orders
  - @*ManyToMany*: A Speaker participates in many Events , and an Event has many speakers

- One-to-Many and Many-to-Many relationships are represented in Java code through
  - Collection
  - Set
  - List
  - Map

Cardinality & Ownership

- *@OneToOne* bidirectional relationship
  - The owner is the side with the foreign key field

- *@OneToMany, @ManyToOne* bidirectional relationship
  - The owner is always the “Many” side (when foreign key scheme is used)

Cardinality Relationships: One to One

One to One Relationships

- Directionality
  - One-to-One Unidirectional
  - One-to-One Bidirectional

- Table model
  - Always Foreign key based (No join table based)

Lab:

Exercise 1: One to One relationship
4321_jpa_mapping.zip

Cardinality Relationships: One to Many

One to Many Relationships

- Directionality
  - One-to-Many Unidirectional
  - One-to-Many Bidirectional

- Table model
  - Join Table (default – when no annotation is used) or
  - Foreign key column based

OneToMany: Bi-directional, Foreign key-based

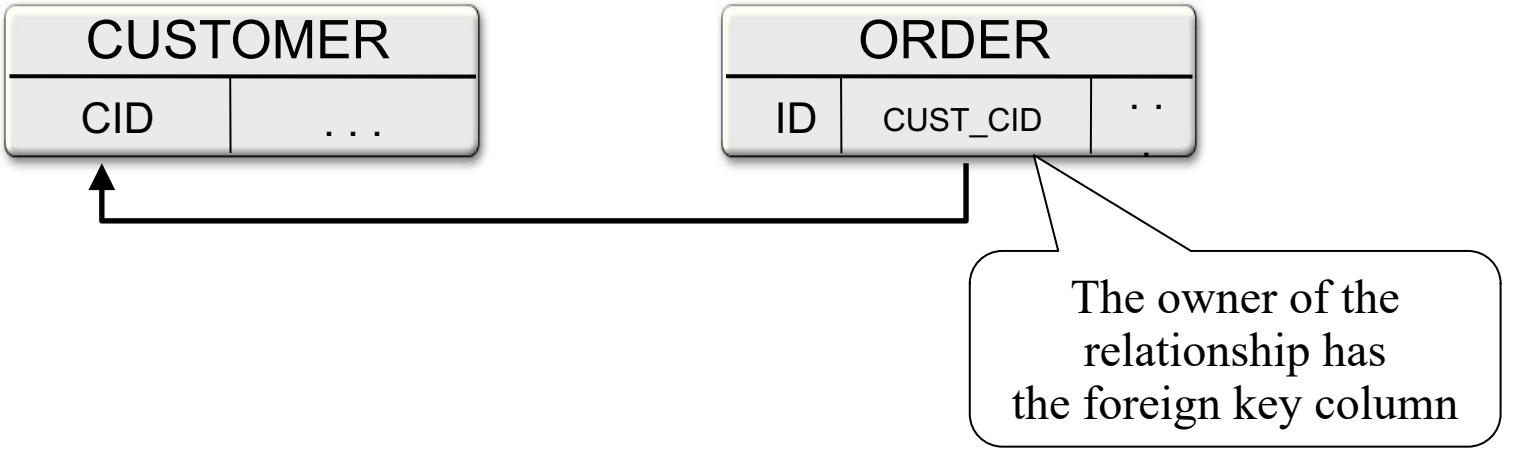
Inverse owner of relationship

```
@Entity
public class Customer {
  @Id
  int cid;
  ...

  @OneToMany(mappedBy="cust")
  List<Order> orders;
}
```

Owner of relationship (always on Many side)

```
@Entity
public class Order {
  @Id
  int id;
  ...
  @ManyToOne
  Customer cust;
}
```

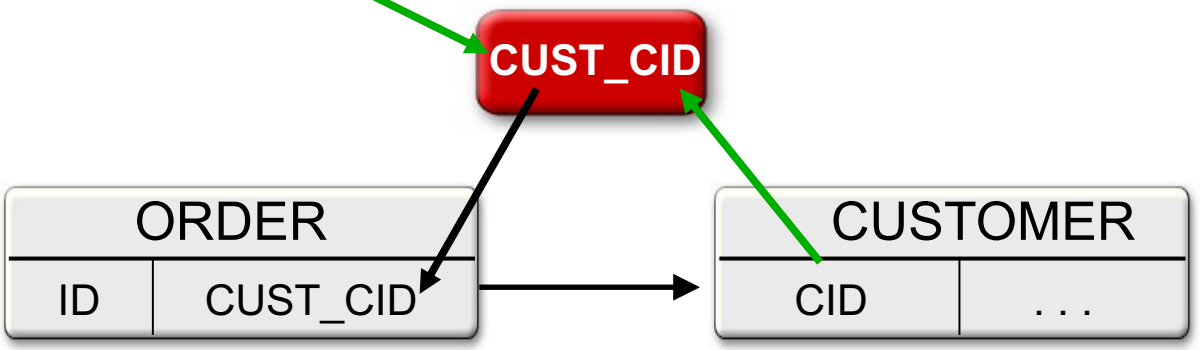


OneToMany: Foreign-key column

```
@Entity
public class Order {

    @Id
    int id;

    @ManyToOne
    Customer cust;
}
```



JPA automatically creates a “CUST_CID” foreign key column for mapping. Can be overridden via @JoinColumn (or @JoinColumns for composite foreign keys).

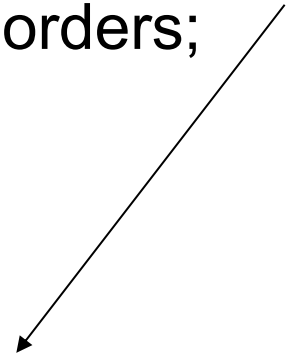
mappedby

Customer Entity:

```
@OneToMany(mappedBy="cust")
 public Set<Order> orders;
```

Order Entity:

```
@ManyToOne
public Customer cust;
```



The inverse side of a bidirectional relationship (Customer) must refer to its owning side (Order) by use of the *mappedBy* element of the OneToOne, OneToMany, or ManyToMany annotation. The *mappedBy* element designates the field in the entity ("cust") in the Owner side (Order).

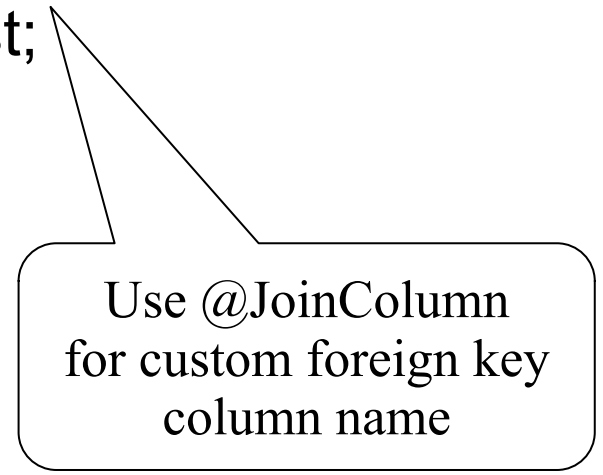
JoinColumn

Customer Entity:

```
@OneToMany(mappedBy=”cust”, cascade=ALL)
public Set<Order> orders;
```

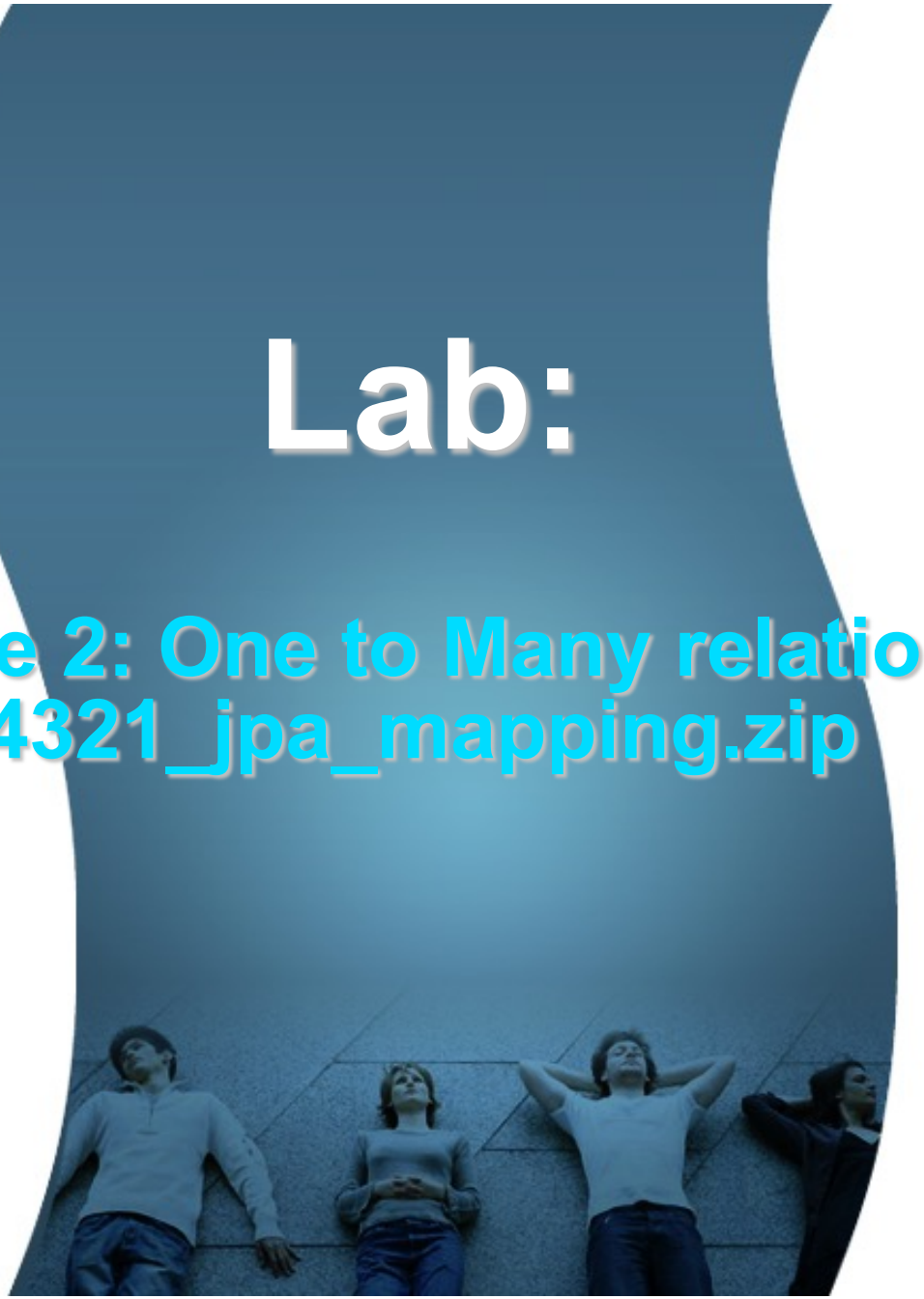
Order Entity:

```
@ManyToOne
@JoinColumn(name=”my_cust_id”)
public Customer cust;
```



Lab:

Exercise 2: One to Many relationship
4321_jpa_mapping.zip



Cardinality Relationships: Many to Many

Many to Many Relationships

- Directionality
  - Many-to-Many Unidirectional
  - Many-to-Many Bidirectional

- Table model
  - Always Join Table (no foreign-key based)

ManyToMany with Default Names

```
@Entity
public class Customer {

  @Id
  int id;

  ...

  @ManyToMany
  Collection<Phone> phones;
}
```

```
@Entity
public class Phone {

  @Id
  int id;

  ...

@ManyToMany(mappedBy="phones")
Collection<Customer> custs;
}
```

Join table name is made up of the 2 entities. Field name is the name of the field plus the name of the PK field.

ManyToMany with Custom Names

```
@Entity(access=FIELD)
public class Customer {
  ...
  @ManyToMany
  @JoinTable(name="CUST_PHONE",
    joinColumns=@JoinColumn(name="CUSTS1_ID"),
    inverseJoinColumns=@JoinColumn(name="PHONES1_ID"))
  Collection<Phone> phones;
}
```

Overriding the default OR mapping here.

Bi-directionality in Many to Many

- Although a ManyToMany relationship is always bi-directional on the database, the object model can choose if it will be mapped in both directions, and in which direction it will be mapped in

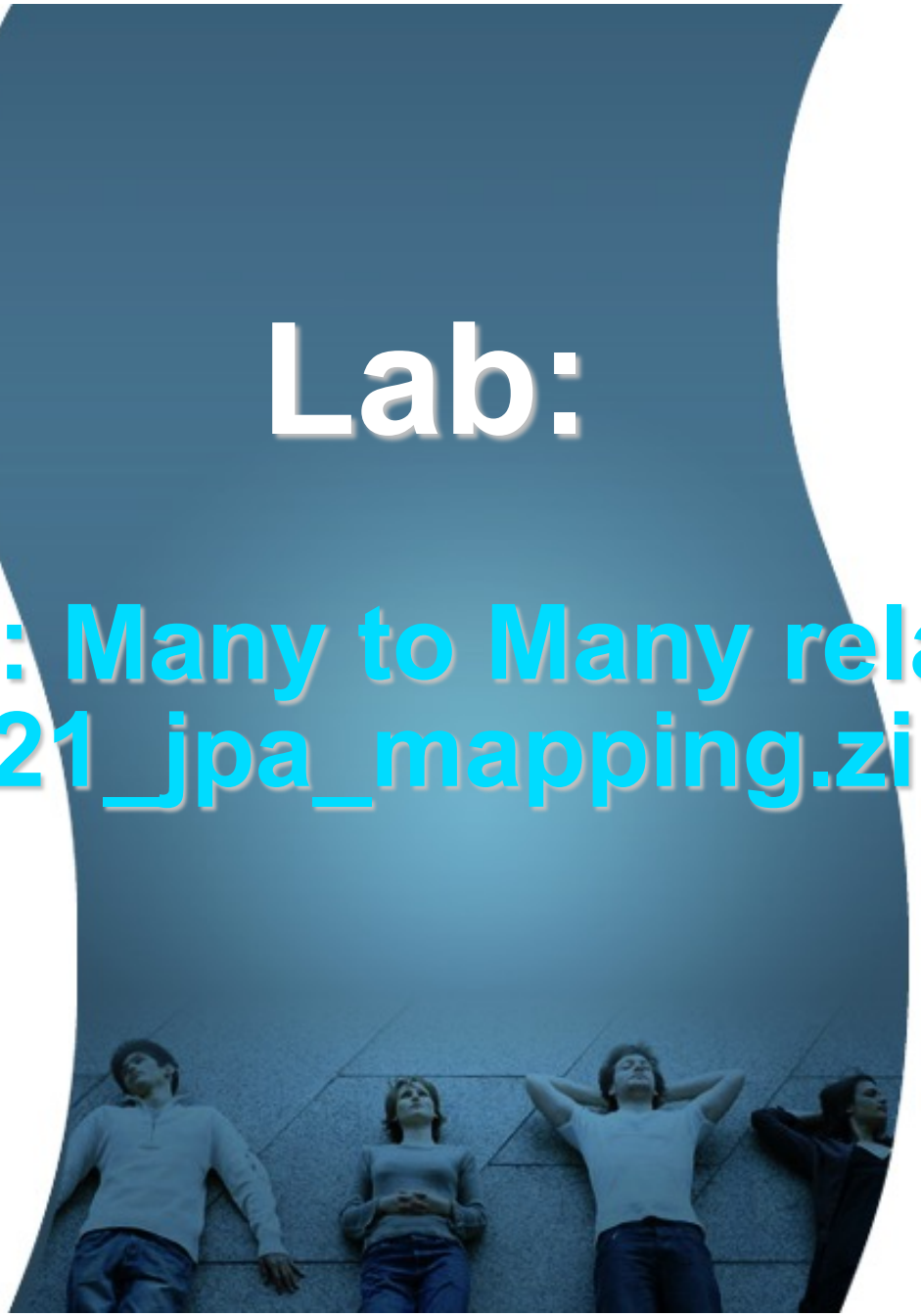
- If you choose to map the relationship in both directions, then one direction must be defined as the owner and the other must use the *mappedBy* attribute to define its mapping
  - This also avoids having to duplicate the JoinTable information in both places.

Bi-directionality in Many to Many

- If the *mappedBy* is not used, then the persistence provider will assume there are two independent relationships
  - You will end up getting duplicate rows inserted into the join table.

Lab:

Exercise 3: Many to Many relationship
4321_jpa_mapping.zip



Cascading

Cascading Behavior

- Cascading is used to propagate the effect of an operation to associated entities
- Cascading operations will work only when entities are associated to the persistence context
  - If a cascaded operation takes place on detached entity, IllegalArgumentException is thrown
- Examples
  - Cascade=PERSIST
  - Cascade=REMOVE
  - Cascade=MERGE
  - Cascade=REFRESH
  - Cascade=ALL

Code with Passion!