Introduction to Association Relationships

In enterprise applications, it is common that entity classes have association relationships. Now you will learn to explore how to implement them using JPA. But before going into implementation details of association relationships let us understand some concepts associated with them.

Directionality

Consider the following Customer and Address classes :

```java
class Customer{
   private Integer customerId;
   private String customerName;
   private Address customerAddress;
   //getter and setter methods

}

class Address{
    private Integer addressId;
    private String doorNumber;
    private String city;
    //getter and setter methods

}
```

The Customer has a reference of Address class. So you can get the address of a customer if you know customerId. But you cannot get details of a customer if you know addressId because the Address class does not have a reference of Customer class. Such type of relationships is called unidirectional relationship. In these relationships the class which has reference of other class is called as owner or source of the relationship and the class whose reference is present is called a target of the relationship. So here, the Customer is the owner and the Address is the target.

Now consider the following Customer and Address classes:

```java
class Customer{
    private Integer customerId;
    private String customerName;
    private Address customerAddress;
    //getter and setter methods

}

class Address{
    private Integer addressId;
    private String doorNumber;
    private String city;
    private Customer customer;
    //getter and setter methods

}
```

The Customer has a reference of Address class and vice versa. So you can get the address of a customer if you know customerId and also customer details if you know addressId. Such types of relationships are called a bidirectional relationship (not covered as part of this course).

Cardinality

The cardinality of a relationship defines how many entities exist on each side of the same relationship instance. In our Customer and Address example, one customer can have many addresses, so the cardinality of the Customer side is one, and the Address side is many. On the basis of cardinality, we have the following types of relationships:

- One – to – One Relationship
- One – to – Many Relationship
- Many – to – One Relationship
- Many – to – Many Relationship (not covered as part of this course)

Let's start with one – to – one relationship.

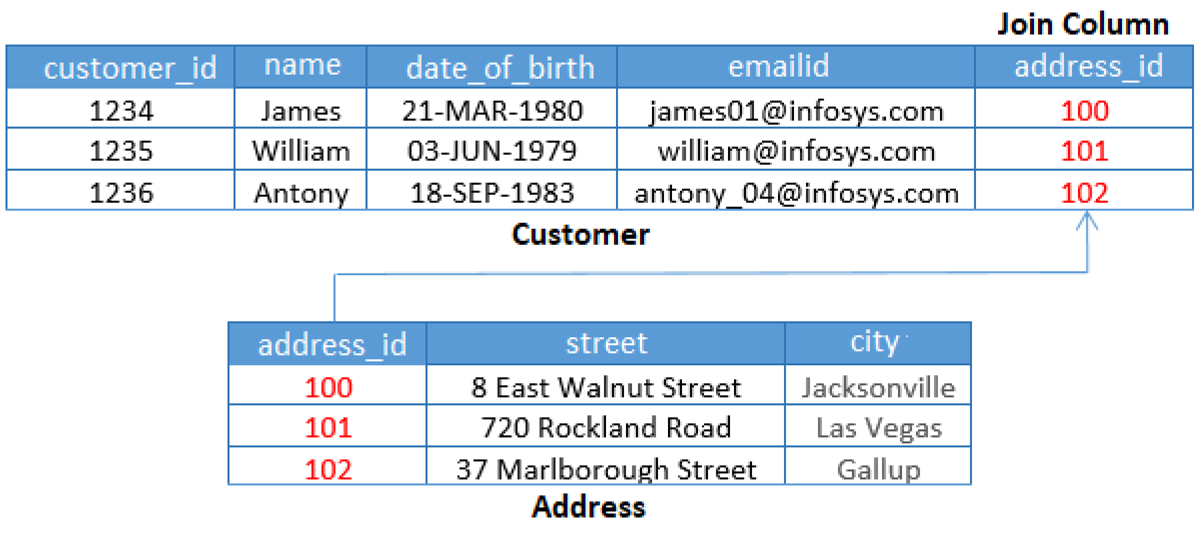
Implementing One-To-One Relationship

Consider the following requirement:

As an admin, I should be able to add, update, retrieve, and delete customer and its address details.

Now let us see how this requirement can be implemented.

A customer can have only one address so you can model this as a one-to-one relationship between Customer and Address entity classes with Customer as owner and Address as a target. To implement this, you can use two tables Customer and Address with Customer table having a foreign key column named address_id that references the Address table as shown below:



customer_id	name	date_of_birth	emailid	address_id
1234	James	21-MAR-1980	james01@infosys.com	100
1235	William	03-JUN-1979	william@infosys.com	101
1236	Antony	18-SEP-1983	antony_04@infosys.com	102

Customer

address_id	street	city
100	8 East Walnut Street	Jacksonville
101	720 Rockland Road	Las Vegas
102	37 Marlborough Street	Gallup

Address

This foreign key column is known as the join column. Now let see how we can create Customer and Address entity classes. Let us begin with Address class which is the target side of relationship and is mapped with the Address table as shown below:

```java
@Entity
public class Address {
  @Id
  private Long addressId;
  private String street;
  private String city;

}
```

The Customer is the owner of the relationship and is mapped with the Customer table and has a reference of Address as shown below :

```
@Entity
public class Customer {
  @Id
  @GeneratedValue(strategy=GenerationType.IDENTITY)
  private Integer customerId;
  private String emailId;
  private String name;
  private LocalDate dateOfBirth;
  @OneToOne(cascade = CascadeType.ALL)
  @JoinColumn(name = "address_id", unique = true)
  private Address address;
  //getter and setter

}
```

In the above code the reference of Address in annotated with @OneToOne annotation which declares that there is a one-to-one relationship between Customer and Address entity classes. The @JoinColumn annotation is used to define the name of the foreign key column in the Customer table that links the customer to the address. Now let us understand these annotations in detail:

@OneToOne(cascade = CascadeType.ALL)

- This annotation specifies that the association relationship has one-to-one multiplicity.
- The cascade attribute specifies operations performed on the owner entity that must be transferred or cascaded to the target entity. It takes values of type CascadeType enumeration. The values of this enumeration are PERSIST, REFRESH, REMOVE, MERGE, DETACH, and ALL. The value ALL specifies that all operations performed on Customer will be cascaded to Address. By default, none of the operations will be cascaded.

@JoinColumn(name = "address_id", unique = true)

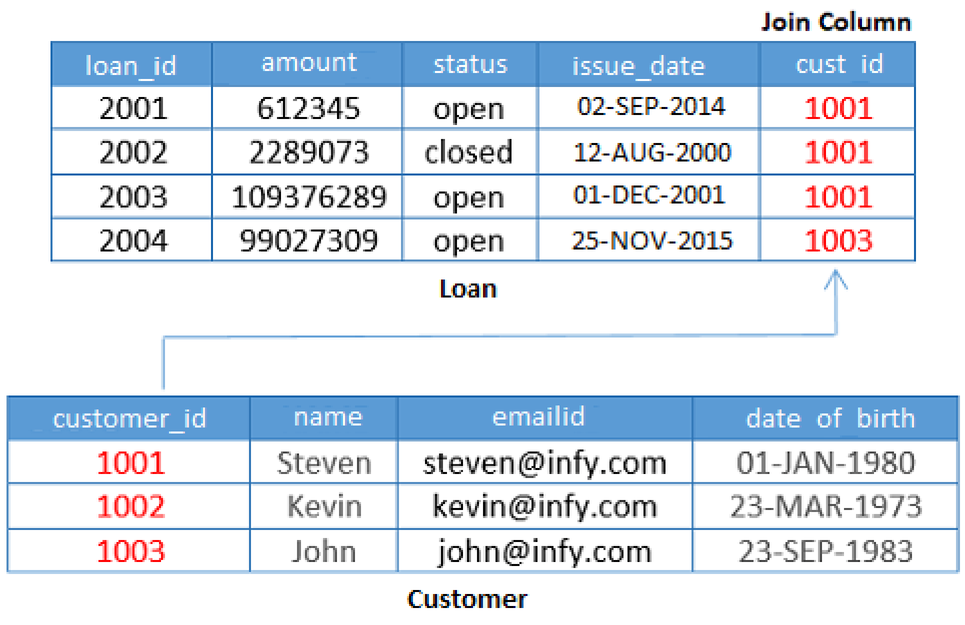
- This annotation is used to specify the foreign key column that joins the owner and target entity.
- The name attribute specifies the name of the foreign key column in the table mapped to the owner entity.
- The unique = true assures that unique values will be stored in the join column.

Implementing Many-To-One Relationship

Consider the following requirement:

As an admin I should be able to sanction multiple loans to a customer.

Now let us see how this requirement can be implemented. A customer can have multiple loans such as car loan, home loan etc. so you can model this as a **many-to-one** relationship between Customer and Loan entity classes with Loan as an owner of the relationship. To implement, you can use two tables Customer and Loan. The Loan table has a foreign key column named cust_id that references the Customer table as shown below:


Join Column

loan_id	amount	status	issue_date	cust_id
2001	612345	open	02-SEP-2014	1001
2002	2289073	closed	12-AUG-2000	1001
2003	109376289	open	01-DEC-2001	1001
2004	99027309	open	25-NOV-2015	1003

Loan

customer_id	name	emailid	date of birth
1001	Steven	steven@infy.com	01-JAN-1980
1002	Kevin	kevin@infy.com	23-MAR-1973
1003	John	john@infy.com	23-SEP-1983

Customer

This foreign key column is known as the join column and it can have duplicate values. Now let us see how to implement Customer and Loan entity classes. Let's begin with Customer class which is target side of relationship and is mapped with Customer table as shown below:

```java
@Entity
public class Customer {
  @Id
  private Integer customerId;
  @Column(name="emailid")
  private String emailId;
  private String name;
  private LocalDate dateOfBirth;
     //getter and setter methods

}
```

The Loan entity class is the owner's side of relationship and is mapped with Loan table. This has a reference of Customer entity class as shown below :

```java
@Entity
public class Loan{
  @Id
  @GeneratedValue(strategy=GenerationType.IDENTITY)
  private Integer loanId;
  private Double amount;
  private LocalDate issueDate;
  private String status;
  @ManyToOne(cascade=CascadeType.ALL)
  @JoinColumn(name="cust_id")
  private Customer customer;

     //getter and setter methods

}
```

In the above code the reference of Customer entity class in annotated with @ManyToOne annotation which declares that there is many-to-one relationship between Customer and Loan. The @JoinColumn annotation is used to define the name of the foreign key column in the Customer table that links the customer to the card. Now let us understand these annotations in detail:

@ManyToOne(cascade = CascadeType.ALL)

- This annotation indicates that the relationship has many-to-one cardinality.
- The cascade attribute tells which operation (such as insert, update, delete) performed on source entity can be transferred or cascaded to target entity. By default, none of the operations will be cascaded. It takes values of type CascadeType enumeration. The value ALL means all operations will be cascaded from source to target. Other values of this enumeration are PERSIST, REFRESH, REMOVE, MERGE, and DETACH.

@JoinColumn(name = "cust_id")

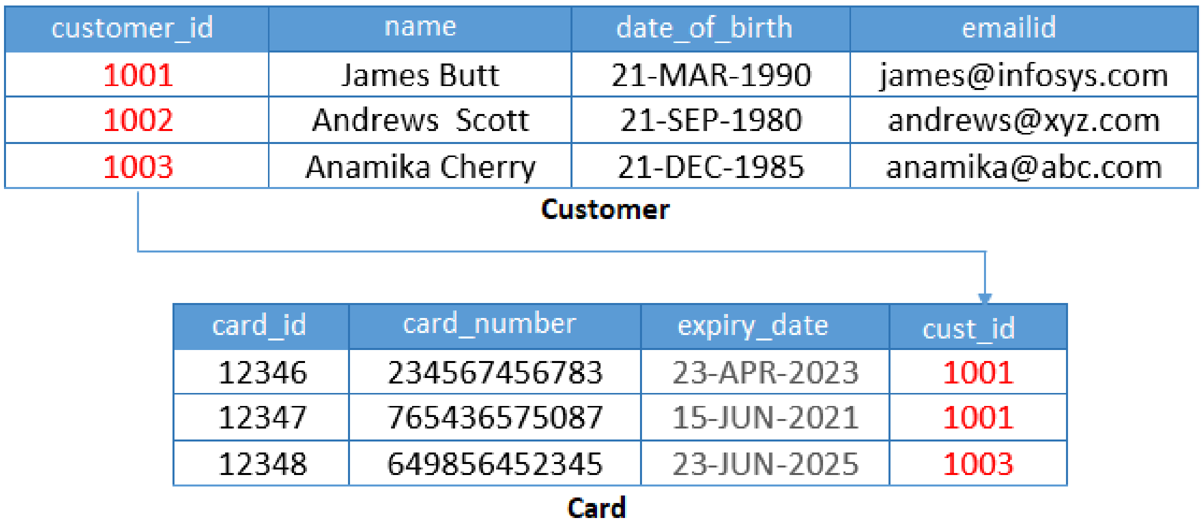
- This annotation is used to define the name of the foreign key column that joins the owner and target entity.
- The name attribute specifies the name of the foreign key column in the table mapped to the source entity.

Implementing One-To-Many Relationship

Consider the following requirement:

As an admin, I should be able to issue multiple cards to the customer.

Now let us see how this requirement can be implemented. A customer can have multiple cards so you can model this as one-to-many relationship between Customer and Card entity classes with Customer entity as owner. To implement this user story, you can use two tables Customer and Card. The Card table has a foreign key column named cust_id which references the Customer table as shown below:



customer_id	name	date_of_birth	emailid
1001	James Butt	21-MAR-1990	james@infosys.com
1002	Andrews Scott	21-SEP-1980	andrews@xyz.com
1003	Anamika Cherry	21-DEC-1985	anamika@abc.com

Customer

card_id	card_number	expiry_date	cust_id
12346	234567456783	23-APR-2023	1001
12347	765436575087	15-JUN-2021	1001
12348	649856452345	23-JUN-2025	1003

Card

Now let us see how you can create entity classes that are mapped with these tables. Let's begin with Card entity class which is the target side of the relationship and is mapped with the Card table as shown below:

```java
@Entity
public class Card {
  @Id
  private Integer cardId;
  private String cardNumber;
  private LocalDate expiryDate;
     // getter and setter methods
}
```

The Customer is the owner's side of the relationship and is mapped with the Customer table. Since one customer can have many cards so Customer entity class has a reference of List<Card> to store information about multiple cards as shown below:

```java
@Entity
public class Customer{
  @Id
  @GeneratedValue(strategy=GenerationType.IDENTITY)
  private Integer customerId;
  private String emailId;
  private String name;
  private LocalDate dateOfBirth;
  @OneToMany(cascade=CascadeType.ALL)
  @JoinColumn(name="cust_id")
  private List<Card> cards;
    //getter and setter methods

}
```

In Customer entity class reference of List<Card> in annotated with @OneToMany annotation which declares that there exists a one-to-many relationship between Customer and Card entity classes. The @JoinColumn annotation is used to give the name of a foreign key column in Card table which links the customer to the card. You can also use Set<Card> instead of List<Card>. Now let us understand these annotations in detail:

@OneToMany(cascade = CascadeType.ALL)

- This annotation indicates that the relationship has one-to-many cardinality.
- The cascade attribute tells which operation (such as insert, update, delete) performed on the source entity can be transferred or cascaded to the target entity. By default, none of the operations will be cascaded. It takes values of type CascadeType enumeration. The value ALL means all operations will be cascaded from source to target. Other values of this enumeration are PERSIST, REFRESH, REMOVE, MERGE, and DETACH.

@JoinColumn(name = "cust_id")

- This annotation is used to specify the join column using its name attribute. The target entity is mapped to the Customer table which has cust_id as the foreign key column, so the name is cust_id.