

Add instructor notes here.

Lesson Objectives



After completing this lesson, participants will be able to understand:

- What is entity association?Different types of entity associationsWhat are class inheritance mappings?
- Implementing associations and mapping using JPA



Add instructor notes here.

What is Entity Association?



Association represents relationship between entities.

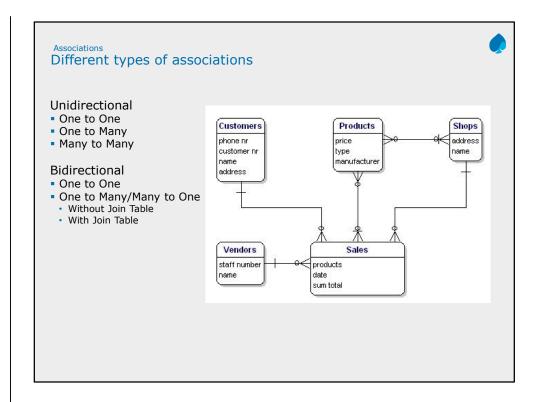
A Java class can contain an object of another class or a set of objects of another class.

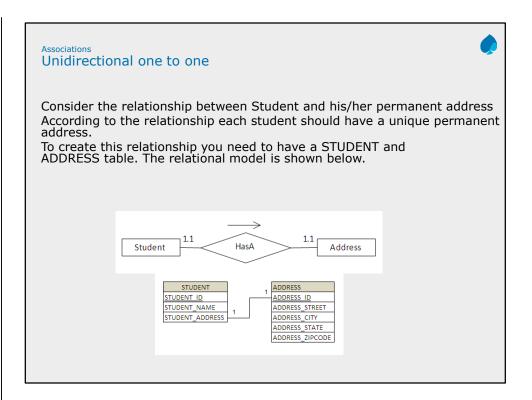
There is no directionality involved in relational world, its just a matter of writing a query. But there is notion of directionality which is possible in

Hence associations are classified as

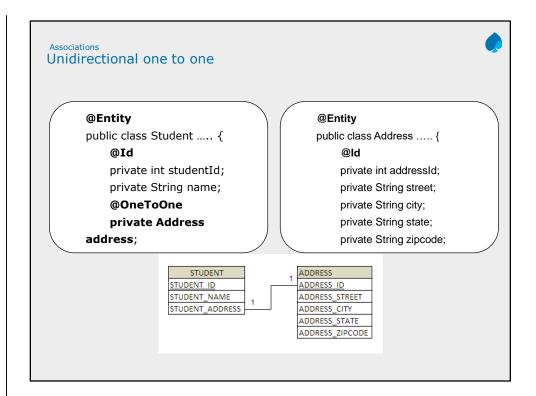
- unidirectional
- bidirectional.







Explain the need of using @JoinColumn in case of associations. Usually. while configuring JPA using annotations, most of the time JPA derives column names based on tables in association. This may conflict if you have some other column name. For example. If we want to store address reference in student table as address id, then the @JoinColumn annotation will help.



@OneToOne

Defines a single-valued association to another entity that has one-to-one multiplicity. This annotations can have following optional attributes:

- cascade (Optional): The operations that must be cascaded to the target of the association. i.e. It indicates JPA operations on associated entity along with owner of association.
- 2. fetch (Optional): Whether the association should be lazily loaded or must be eagerly fetched. i.e. When you fetch Student entity, if you want to load the associated entity (Address) immediately, then you have to mention this attribute with 'EAGER'. Default is LAZY, means the associated entity (Address) will be loaded when required.

Note: In case of relationship, most of the times associated column name (for example, STUDENT_ADDRESS in STUDENT table) is different and creates confits, to avoid this, JPA provides @JoinColumn annotation.

Add instructor notes here.

Associations

Cascading associated Entities



Cascade attribute is mandatory, whenever we apply relationship between objects, cascade attribute transfers operations done on one object onto its related child objects.

This attribute indicates JPA operations on associated entity along with owner of association. It may take one of the value represented by CascadeType enumeration.

- PERSIST
- MERGE
- REMOVE
- ALL

Cascade Types:

ALL: All cascade operations will be applied to the parent entity's related entity. All is equivalent to specifying cascade={DETACH, MERGE, PERSIST, REFRESH, REMOVE}

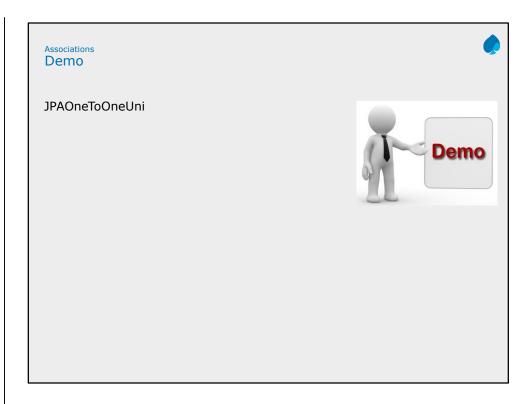
DETACH: If the parent entity is detached from the persistence context, the related entity will also be detached.

MERGE: If the parent entity is merged into the persistence context, the related entity will also be merged.

PERSIST: If the parent entity is persisted into the persistence context, the related entity will also be persisted.

REFRESH: If the parent entity is refreshed in the current persistence context, the related entity will also be refreshed.

REMOVE :If the parent entity is removed from the current persistence context, the related entity will also be removed.



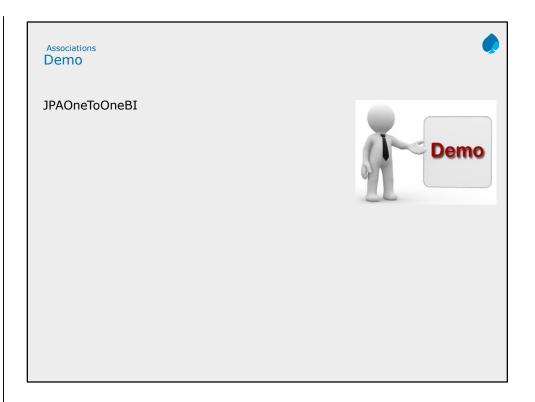
Add instructor notes here.

Associations Bidirectional one to one In this example, one employee can have one address and one address belongs to one employee only. Here, we are using bidirectional association. In such case, a foreign key is created in the primary table. Consider the following classes: @Entity @Entity public class Address { public class Student { @Id @Id private int addressId; private int studentId; private String street; private String name; private String zipcode; @OneToOne @OneToOne(mappedBy="addr private Address ess") address: private Student student;

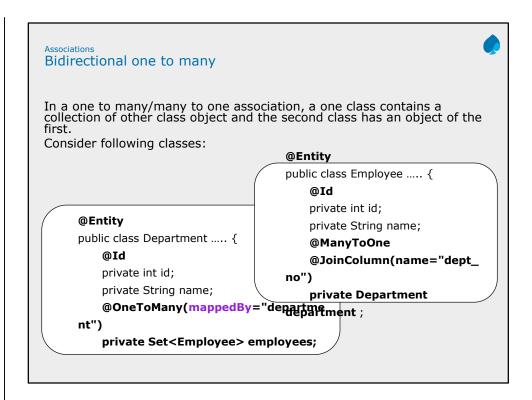
The direction of a relationship can be either bidirectional or unidirectional. A bidirectional relationship has both an **owning side** and an **inverse side**. A unidirectional relationship has only an owning side. The owning side of a relationship determines how the Persistence runtime makes updates to the relationship in the database.

In a bidirectional relationship, each entity has a relationship field or property that refers to the other entity. Through the relationship field or property, an entity class's code can access its related object. If an entity has a related field, the entity is said to "know" about its related object. Such relationship field must be marked with **mappedBy** attribute.

The inverse side of a bidirectional relationship must refer to its owning side by using the mappedBy element of the @OneToOne, @OneToMany, or @ManyToMany annotation.

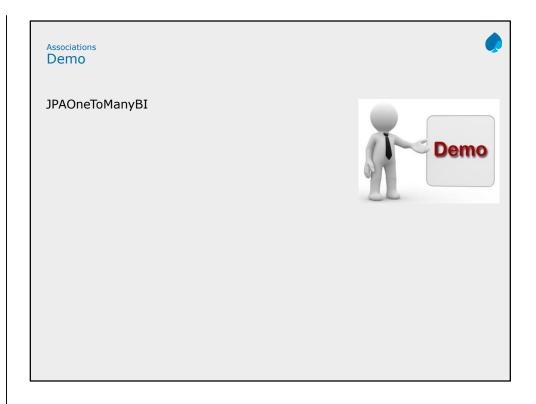


Add instructor notes here.

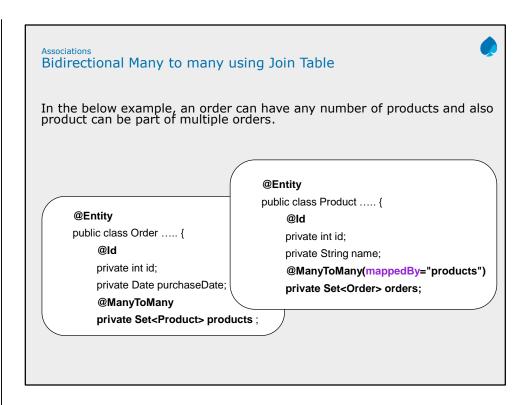


Slide example shows association between Department and its Employees. As one department can have many employees, we need to use @OneToMany annotation to represent this relationship. Therefore we need to use any type of collection as per given requirement. As department cannot have duplicate employees, so slide example uses Set<Employee> collection to store employees.

Note: The many side of many-to-one bidirectional relationships must not define the mappedBy element. The many side is always the owning side of the relationship. As shown in the slide, the Employee is owning side of the relationship.



Add instructor notes here.



For many-to-many bidirectional relationships, either side may be the owning side. For the given example, Order acts as owning side of the relationship.

Add instructor notes here.

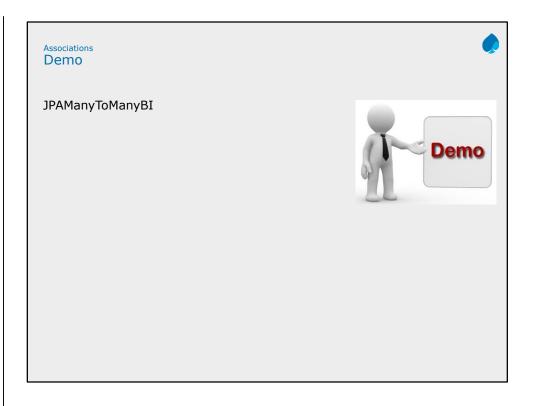


To store data of many to many relationship, join table can be used. As shown in slide, the orders stored in ORDER_MASTER table, products stored in PRODUCT_MASTER and there association is stored in PRODUCT_ORDERS.

To implement the above relationship with join table, JPA allows owning side of relationship to describe join table using @JoinTable annotation. For example,

@JoinTable: This annotation is used to describe join table properties. It has three attributes:

- 1. name: Name of the join table
- 2. joinColumns: Join column name for owning side i.e. order table
- 3. inverseColumns: Join column name for inverse side. i.e. product table.



Add instructor notes here.

Mapping Inheritance



Java classes are may related to each other in form of inheritance However, there is no inheritance between database tables JPA allows hierarchical classes to be mapped with tables with below listed strategies:

- Single Table per class hierarchy
- Table per class
- Joined subclass

Three ways of handling inheritance

- 1. Single table per class hierarchy (InheritanceType.SINGLE_TABLE)
- 2. Table per concrete entity class (InheritanceType.TABLE_PER_CLASS)
- 3. "join" strategy, where fields or properties that are specific to a subclass are mapped to a different table than the fields or properties that are common to the parent class (InheritanceType.JOINED)

Let us explore each in detail.

Add instructor notes here.

Inheritance Single Table per Class Hierarchy



In this strategy, a single table is created for all classes in the inheritance hierarchy. $\,$

It uses additional column called discriminator to distinguish the object of child classes

The value in discriminator column is used to identify rows belonging to subclasses

Single Table per Class Hierarchy:

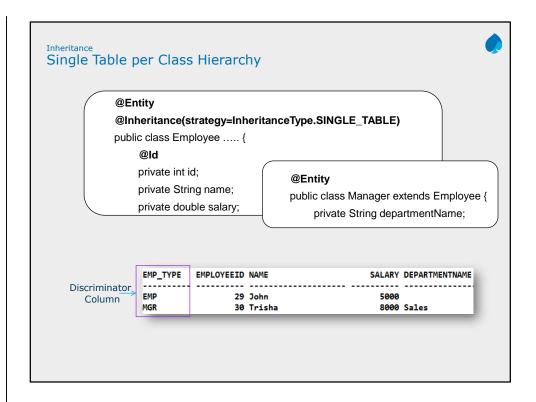
In this strategy, only one database table is created for all subclasses. It is denormalized table has columns for all attributes.

JPA Mapping Configuration:

Single annotation @Inheritance with InheritanceType strategy required only on superclass. Also use '@DiscriminatorColumn' to define discriminator column and it data type, which later will be used to differentiate parent and child rows.

Note: Each class in hierarchy can provide optional discriminator value for its own objects rows. This is done by using @DiscriminatorValue annotation.

Add instructor notes here.

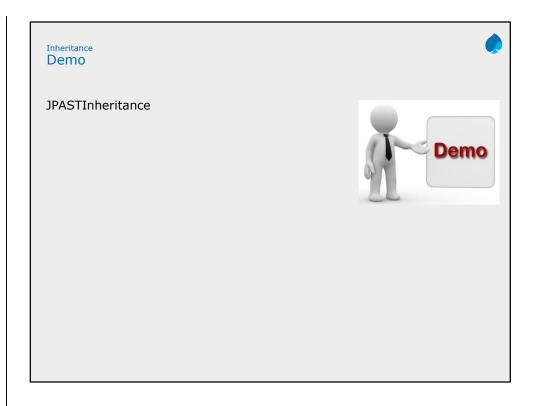


Advantages

- 1. It is the fastest of all inheritance models
- 2. Since it does not requires a join to retrieve a persistent instance from the database.
- Persisting or Updating a persistent instance requires only a single INSERT or UPDATE statement.

Disadvantages

 The larger the inheritance model gets, the "wider" the mapped table gets, in that for every field in the entire inheritance hierarchy, a column must exist in the mapped table. This may have undesirable consequence on the database size, since a wide or deep inheritance hierarchy will result in tables with many mostly-empty columns.



Add instructor notes here.

Inheritance Table per Concrete Class



In this strategy, a table is created for each class in the hierarchy

Each table will have columns for all properties in parent and child class

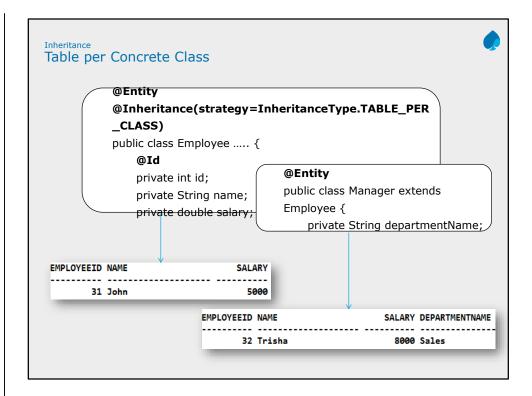
Support for this strategy is optional, and may not be supported by all Java Persistence API providers.

Table per concrete class:

In this inheritance strategy, one database table will be created for the superclass AND one per subclass.

Subclass tables have their object-specific columns along with shared columns from superclass table.

Add instructor notes here.

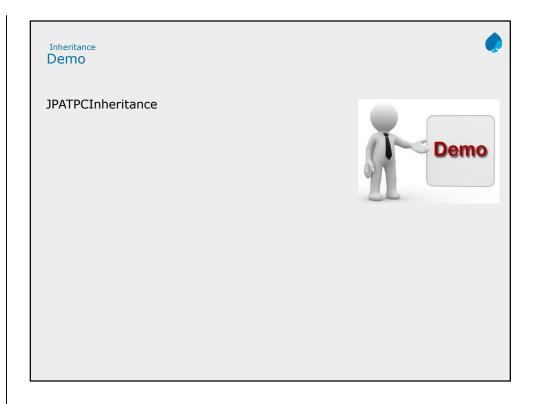


Advantages:

1. This is the easiest method of Inheritance mapping to implement.

Disadvantages:

- 1. Data that belongs to a parent class is scattered across a number of subclass tables, which represents concrete classes.
- 2. This hierarchy is not recommended for most cases.
- 3. Changes to a parent class is reflected to large number of tables
- 4. A query couched in terms of parent class is likely to cause a large number of select operations



Add instructor notes here.

Inheritance
Joined Subclass Strategy



In this strategy, a separate table is created for each class in the inheritance hierarchy.

However, columns inherited from the parent class are not repeated in subclass. $% \left(1\right) =\left(1\right) \left(1\right) \left($

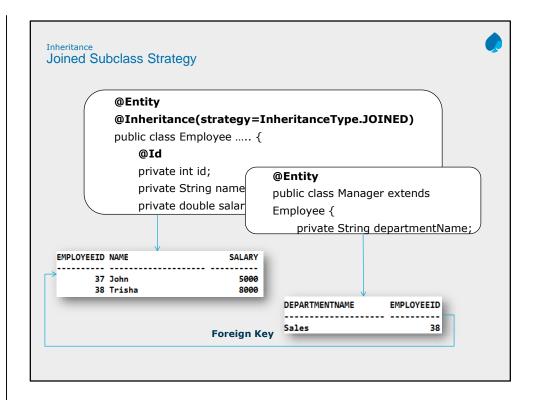
The parent table primary key is used as foreign key for child tables.

Joined Subclass Hierarchy:

In this inheritance strategy, one database table will be created for the superclass AND one per subclass.

Subclass tables have their object-specific columns along with a foreign key column referring primary key of Superclass.

Add instructor notes here.

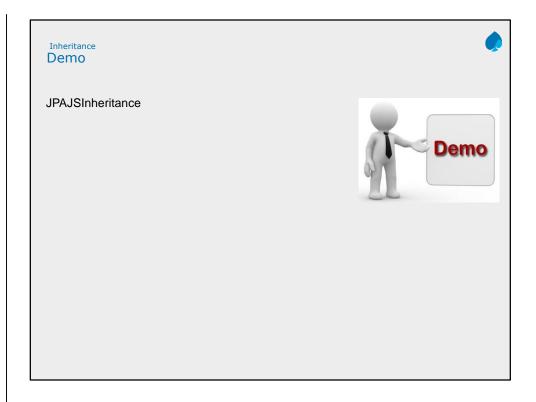


Advantages

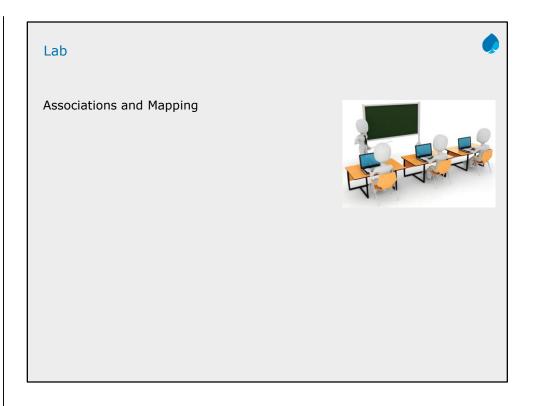
1. Using joined subclass tables results in the most normalized database schema, meaning the schema with the least spurious or redundant data.

Disadvantages

 Retrieving any subclass requires one or more database joins, and storing subclasses requires multiple INSERT or UPDATE statements.

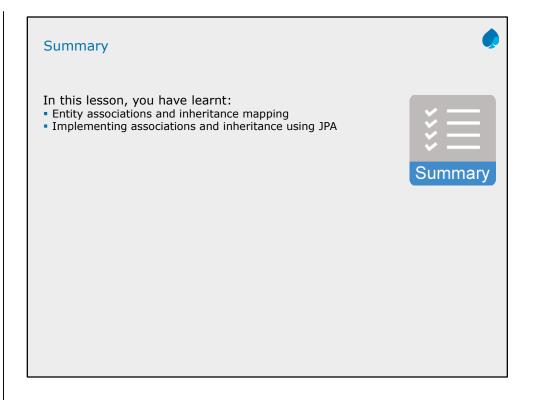


Add instructor notes here.



Add the notes here.

Add instructor notes here.



Add the notes here.

Answers:

InheritanceType.SINGLE _TABLE

2. True

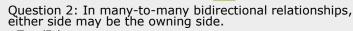
Review Question



Question 1: Which one of the following inheritance mapping type is suitable if you want to create single table for all classes in hierarchy?

- InheritanceType.TABLE_PER_CLASSInheritanceType.SINGLE_TABLE
- InheritanceType.JOINED

TKA5



True/False

