Mindtree

Welcome to possible

Hibernate annotation and entity manager

© Mindtree limited 2012

Objectives

- How to use annotations in mapping?
- Define EntityManager
- Life Cycle methods in hibernate framework
- Differentiate between XML based and annotations based mapping

Mapping with JPA (Java Persistence Annotations)

- JPA entities are plain POJOs. Actually, they are Hibernate persistent entities. Their mappings are defined through JDK 5.0 annotations instead of hbm.xml files.
- Annotations can be split in two categories,
  - The logical mapping annotations (describing the object model, the association between two entities etc.)
  - The physical mapping annotations (describing the physical schema, tables, columns, indexes, etc).
- JPA annotations are in the javax.persistence.* package.

Marking a POJO as persistent entity

```java
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;

@Entity
@Table(name = "EMP")
public class Employee {
    @Id
    @Column(name = "EMP_ID")
    private int empId;
    @Column(name = "EMP_NAME", length = 50, nullable = false)
    private String name;
    @Column(name = "HIRE_DATE")
    @Temporal(TemporalType.TIMESTAMP)
    private Date hireDate;
    @Column(name = "SAL", precision = 2)
    private double salary;
```

@Entity
Every persistent POJO class is an entity and is declared using the @Entity annotation (at the class level)

@Table is set at the class level; it allows you to define the table, catalog, and schema names for your entity mapping. If no @Table is defined the default values are used: the unqualified class name of the entity.

Marking a POJO as persistent entity

```java
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;
import javax.persistence.Temporal;
import javax.persistence.TemporalType;

@Entity
@Table(name = "EMP")
public class Employee {
    @Id
    @Column(name = "EMP_ID")
    private int empId;
    @Column(name = "EMP_NAME", length = 50, nullable = false)
    private String name;
    @Column(name = "HIRE_DATE")
    @Temporal(TemporalType.TIMESTAMP)
    private Date hireDate;
    @Column(name = "SAL", precision = 2)
    private double salary;
```

@Id declares the identifier property of this entity

Marking a POJO as persistent entity

- **Generating the identifier property**
- JPA defines five types of identifier generation strategies:
  - AUTO - either identity column, sequence or table depending on the underlying DB
  - TABLE - table holding the id
  - IDENTITY - identity column
  - SEQUENCE - sequence
  - identity copy - the identity is copied from another entity

Usage example:

```
@Id @GeneratedValue(strategy=GenerationType.IDENTITY)
private int empId;
```

Marking a POJO as persistent entity

- **Declaring column attributes:** @Column annotation. Use it to override default values
  - name (optional): the column name (default to the property name)
  - unique (optional): set a unique constraint on this column or not (default false)
  - nullable (optional): set the column as nullable (default true).
  - length (optional): column length (default 255)
  - precision (optional): column decimal precision (default 0)
  - scale (optional): column decimal scale if useful (default 0)

JPA Persistence Unit

- A JPA Persistence Unit is a logical grouping of user defined persistable classes (entity classes, embeddable classes and mapped superclasses) with related settings

  - **Persistence Unit** - is a named configuration of entity classes.
  - **Persistence Context** - is a managed set of entity instances. The entities classes are part of the Persistence Unit configurations
  - **Managed Entities** - an entity instance is managed if it is part of a persistence context and that Entity Manager can act upon it

The persistence.xml

- The previous tutorials used the Hibernate-specific hibernate.cfg.xml configuration file. JPA, however, defines a different bootstrap process that uses its own configuration file named persistence.xml.
- Create a file called persistence.xml in the src/META-INF directory

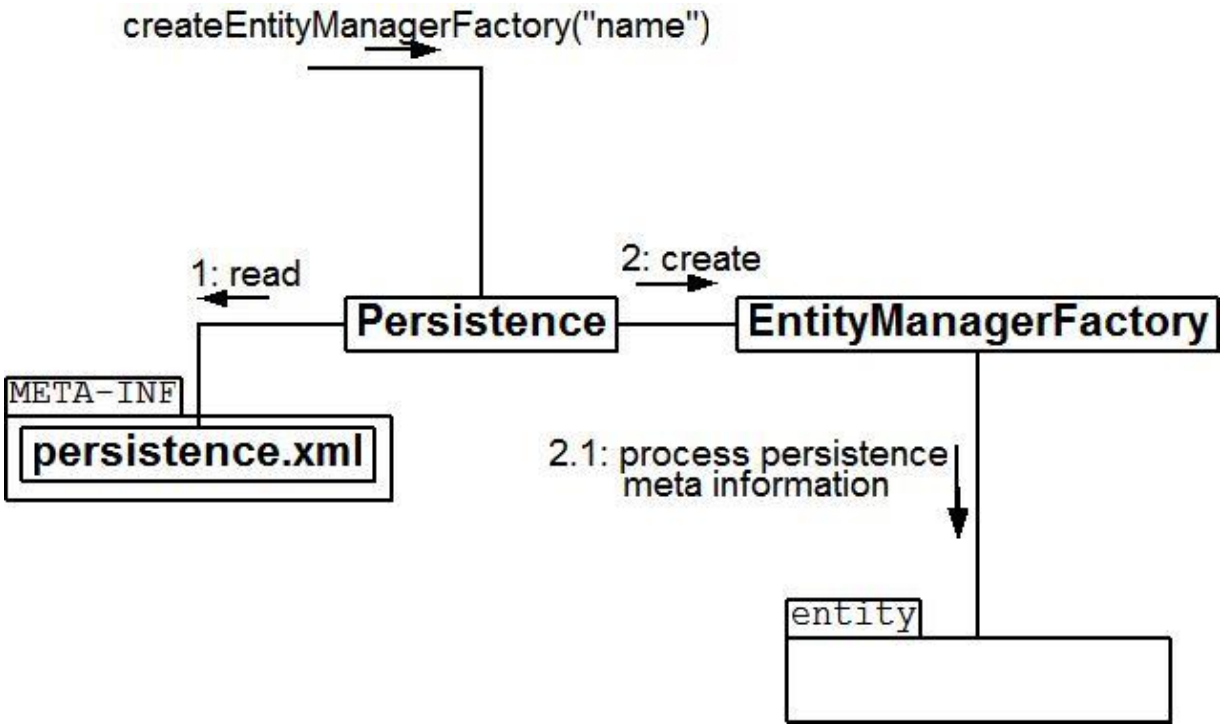
```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
           http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
        version="2.0">
    <persistence-unit name=“JPAService">
        ...
    </persistence-unit>
</persistence>
```

The persistence.xml listing

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://
  <persistence-unit name="JPAService" transaction-type="RESOURCE_LOCAL">
        <provider>org.hibernate.ejb.HibernatePersistence</provider>
        <properties>
            <property name="hibernate.connection.driver_class"
                value="com.mysql.jdbc.Driver"/>
            <property name="hibernate.connection.password"
                value="Welcome123"/>
            <property name="hibernate.connection.url"
                value="jdbc:mysql://localhost/happytrip"/>
            <property name="hibernate.connection.username" value="root"/>
            <property name="hibernate.dialect"
                    value="org.hibernate.dialect.MySQLDialect"/>
                <property name="hibernate.hbm2ddl.auto" value="update" />
            <property name="hibernate.archive.autodetection" value="class"/>
            <property name="hibernate.show_sql" value="true"/>
        </properties>
    </persistence-unit>
</persistence>
```

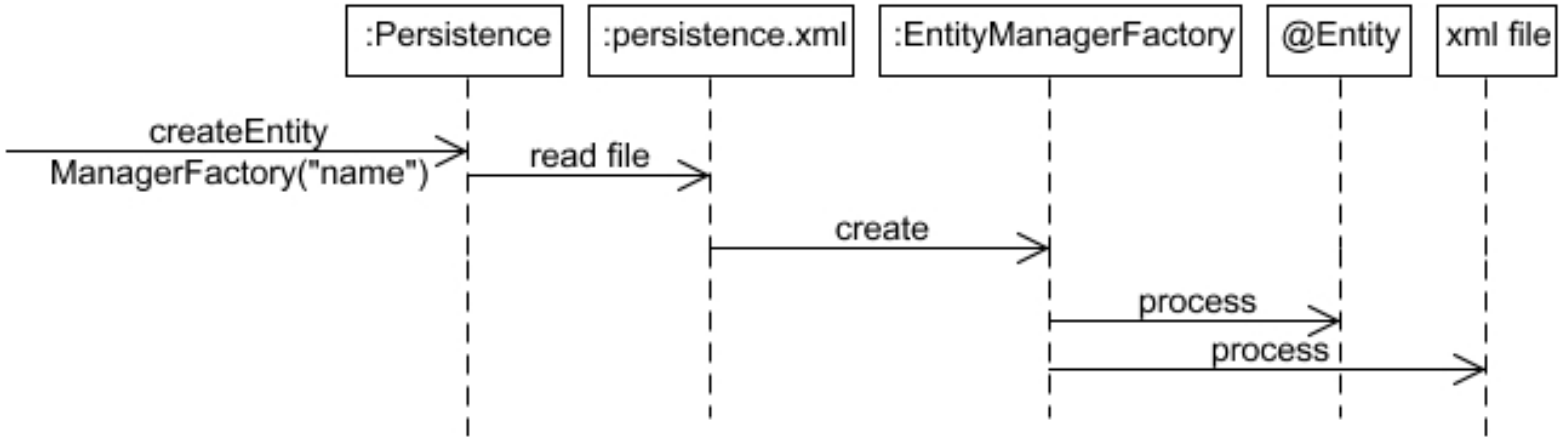
EntityManagerFactory

- An instance of this class provides a way to create entity managers.
- The Entity Manager Factory is the in-memory representation of a Persistence Unit



EntityManagerFactory

- Sequence

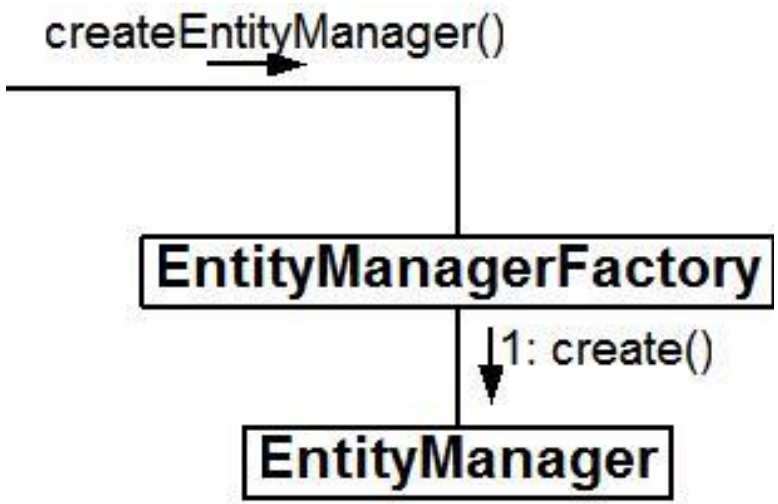


EntityManager in JPA

- **EntityManager is a** class that manages the persistent state(or lifecycle) of an entity.
  - There are three main types of Entity Managers defined in JPA.
    - Container Managed Entity Managers
      - When a container of the application(be it a Java EE container or any other custom container like Spring) manages the lifecycle of the Entity Manager, the Entity Manager is said to be Container Managed.
    - Application Managed Entity Managers
      - An Entity Manager that is created not by the container, but actually by the application itself is an application scoped Entity Manager

EntityManager in JPA

- It provides methods for persisting, merging, removing, retrieving and querying objects. It is **not** thread safe so we need one per thread.
- The Entity Manager also serves as a first level cache.
- It maintains changes and then attempts to optimize changes to the database by batching them up when the transaction completes.



EntityManager in JPA

Controls life-cycle of entities

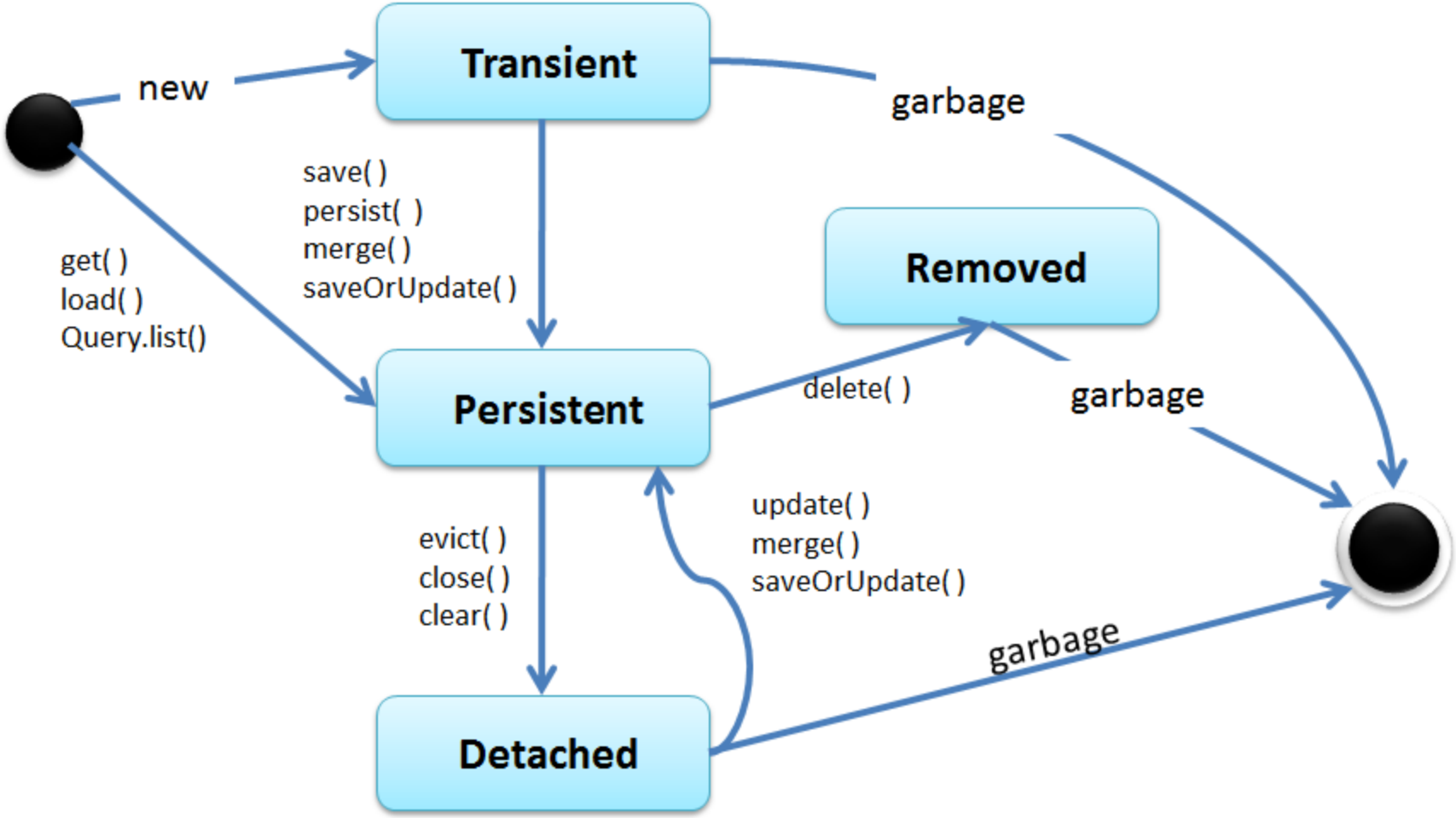
- persist() - insert an entity into the DB
- remove() - remove an entity from the DB
- merge() - synchronize the state of detached entities
- refresh() - reloads state from the database

Using EnitityManager

```
public static void main(String[] args) {
    /* There needs to be a persistence.xml file in the
     * META-INF folder of the application with
     * persistence-unit name="JPAService"
     */
    EntityManagerFactory emf =
            Persistence.createEntityManagerFactory("JPAService");
        EntityManager em = null;

        try {
            em = emf.createEntityManager();
            em.getTransaction().begin();
                Employee employee =
                        new Employee(100,"Smith",new Date(),65666.66);
                em.persist(employee);
            em.getTransaction().commit();
        } finally{
            em.close();
        }
}
```

Hibernate Object States



Mindtree

Welcome to possible

Hibernate association mapping

© Mindtree limited 2012

Objectives

- Understand Hibernate association mapping

Association Mapping

- JPA supports all standard relationships
  - One-To-One
  - One-To-Many
  - Many-To-One
  - Many-To-Many

- Supports unidirectional and bidirectional relationships

One-to-One Mapping

- Consider a relationship between Employee and Address.
- Employee has relationship with "Address" table.
- Employee has a Address and its unique and that address can't be assigned to other employees. This relationship is called "one-to-one".
- We can achieve one-to-one mapping by:
  - Sharing of Primary Key: by creating a primary key in each of the relations or tables, where the associated table will contain the primary key value of the associating table, which is called as foreign key.
  - Making Foreign key unique

@OneToOne using @JoinColumn

```
@Entity
public class Employee {
        @Id
        @GeneratedValue
        private int empid;
        private String name;

        @OneToOne
         @JoinColumn(name="ADD_ID")
        //@PrimaryKeyJoinColumn
        Address homeAddress;

@Entity
public class Address {
    @Id
    @GeneratedValue
    private int id;
    private String street;
    @OneToOne(mappedBy="homeAddress")
    Employee employee;
```

EMPID	NAME	ADD_ID
1	Ganesh	1

ID	STREET
1	M.G.Road

@OneToOne using @PrimaryKeyJoinColumn

```
@Entity
public class Employee {
        @Id
        @GeneratedValue
        private int empid;
        private String name;

        @OneToOne
        //@JoinColumn(name="ADD_ID")
        @PrimaryKeyJoinColumn
        Address homeAddress;
```

EMPID	NAME
1	Ganesh

ID	STREET
1	M.G.Road

```
@Entity
public class Address {
    @Id
    @GeneratedValue
    private int id;
    private String street;
    @OneToOne(mappedBy="homeAddress")
    Employee employee;
```

CASCADE types

- PERSIST: When the owning entity is persisted, all its related data is also persisted.
- MERGE: When a detached entity is merged back to an active persistence context, all its related data is also merged.
- REMOVE: When an entity is removed, all its related data is also removed.
- ALL: All the above applies.

@OneToMany

- @OneToMany defines the *one* side of a one-to-many relationship
- The *mappedBy* element of the annotation defines the object reference used by the *child* entity
- @OrderBy defines an collection ordering required when relationship is retrieved
- The child (many) side will be represented using an implementation of the java.util.Collection interface

One-to-Many unidirectional

```java
@Entity
public class Trainer {
    @Id
    @GeneratedValue
    @Column
    private Integer id;
    @Column
    private String name;

    @OneToMany
    @JoinColumn(name = "trainer_id")
    private Set<Course> courses;
```

```java
@Entity
public class Course {
    @Id
    @GeneratedValue
    private Integer id;
    @Column
    private String name;
```

Table: TRAINER

ID	NAME
1	Banu Prakash

Table: COURSE

ID	NAME	TRAINER_ID
2	Java	1
3	Hibernate	1
4	Spring	1

One-to-Many unidirectional

```
EntityManager em =  emf.createEntityManager();
Trainer trainer = new Trainer();
trainer.setName("Banu Prakash");
em.persist(trainer);
em.getTransaction().begin();
Course c1 = new Course();
c1.setName("Java");
Course c2 = new Course();
c2.setName("Hibernate");
Course c3 = new Course();
c3.setName("Spring");
trainer.setCourses(new HashSet<Course>());
trainer.getCourses().add(c1);
trainer.getCourses().add(c2);
trainer.getCourses().add(c3);
em.persist(c1);
em.persist(c2);
em.persist(c3);
em.getTransaction().commit();
```

Table: TRAINER

ID	NAME
1	Banu Prakash

Table: COURSE

ID	NAME	TRAINER_ID
2	Java	1
3	Hibernate	1
4	Spring	1

One-to-Many unidirectional Without Join Column

- Without @JoinColumn a link table “TRAINER_COURSE” is created.

```java
@Entity
public class Trainer {
    @Id
    @GeneratedValue
    @Column
    private Integer id;
    @Column
    private String name;

    @OneToMany
    //@JoinColumn(name = "trainer_id")
    private Set<Course> courses;
```

Table: TRAINER

ID	NAME
1	Banu Prakash

Table: COURSE

ID	NAME
2	Java
3	Hibernate
4	Spring

Table: TRAINER_COURSE

TRAINER_ID	COURSES_ID
1	4
1	2
1	3

One-to-many and Many-to-one [Bidirectional]

```
@Entity public class Customer {
   @Id String name;

   @OneToMany(mappedBy = "customer", cascade = CascadeType.ALL)
   Set<Order> orders = new HashSet<Order>();
```

Table: CUSTOMER

NAME
Banu Prakash
Ajay

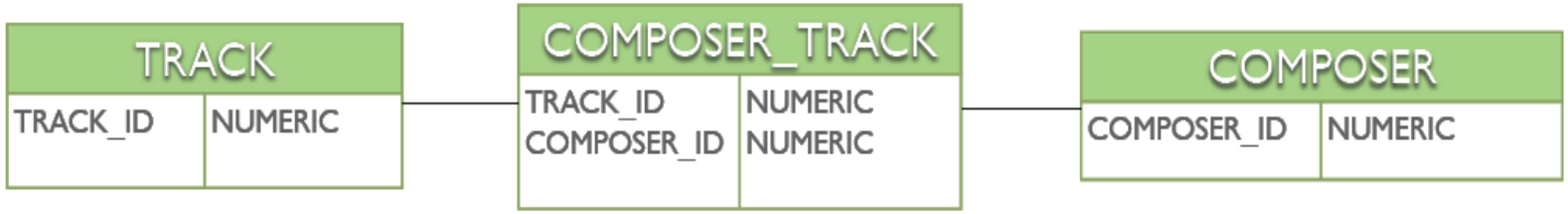
```
@Entity
@Table(name ="OrderTable")
public class Order {
   @Id
   @GeneratedValue(strategy=GenerationType.SEQUENCE)
   int orderId;

@ManyToOne
@JoinColumn(name = "customer_fk"
Customer customer;
```

Table: ORDERTABLE

ORDERID	AMOUNT	ORDERDATE	CUSTOMER_FK
10	1234	2008-02-11 11:56:13.0	Banu Prakash
11	42234	2008-02-11 11:56:13.0	Banu Prakash
12	61223	2008-02-11 11:56:13.0	Ajay
13	8234	2008-02-11 11:56:13.0	Ajay

@ManyToMany




```java
@Entity
public class Track {

   @Id
   @Column(name = "TRACK_ID")
   private Long id;

   @ManyToMany(mappedBy="compositions")
   private Set<Composer> composers
               = new HashSet<Composer>();
}
```

```java
@Entity
public class Composer {
   @Id
   @Column(name = "COMPOSER_ID")
   private Long id;

   @ManyToMany
   @JoinTable(name="COMPOSER_TRACK",
   joinColumns = { @JoinColumn(name = "COMPOSER_ID") },
   inverseJoinColumns = { @JoinColumn(name = "TRACK_ID")
   private Set<Track> compositions;
}
```

FETCH types

Eager fetching	**Also know as eager loading, this is the default option, the entity-manager will attempt to retrieve all of the entity field data when the find method is invoked.** **When eagerly fetching the EntityManager will use a JOIN in the SELECT statement to retrieve the entity.**
Lazy fetching	Lazy fetching supports more than one mechanism @Basic - specifiying a column with this annotation means that the data will only be load the first time it is accessed When lasy fetching the EntityManager would use separate SELECT statements and thus is less efficient than eager loading, when using entity relationships.

Welcome to possible

India | USA | UK | Germany | Sweden | Belgium | France | Switzerland | UAE | Singapore | Australia | Japan | China