JAVA PERSISTENCE API (JPA)

Tutorial

Simply Easy Learning

www.tutorialspoint.com

SIMPLY EASY LEARNING

About the Tutorial

This tutorial provides a basic understanding of how to store a copy of database objects into temporary memory using JAVA Persistence API (JPA).

Audience

This tutorial is designed for readers intend to do Java programing with Database connectivity, using Persistence API.

Prerequisites

Awareness of Java programming with JDK 1.6 or later is a prerequisite to understand this tutorial. In addition, we assume the readers are acquainted with the concepts of JDBC in Java.

Copyright & Disclaimer

© Copyright 2014 by Tutorials Point (I) Pvt. Ltd.

All the content and graphics published in this e-book are the property of Tutorials Point (I) Pvt. Ltd. The user of this e-book is prohibited to reuse, retain, copy, distribute or republish any contents or a part of contents of this e-book in any manner without written consent of the publisher.

We strive to update the contents of our website and tutorials as timely and as precisely as possible, however, the contents may contain inaccuracies or errors. Tutorials Point (I) Pvt. Ltd. provides no guarantee regarding the accuracy, timeliness or completeness of our website or its contents including this tutorial. If you discover any errors on our website or in this tutorial, please notify us at contact@tutorialspoint.com .

tutorialspoint
SIMPLYEASYLEARNING

Table of Contents

About the Tutorial 3

Audience 3

Prerequisites 3

Copyright & Disclaimer 3

Table of Contents 4

1. JPA – INTRODUCTION 7

Mismatch between Relational and Object Models 7

What is JPA? 7

Where to use JPA? 7

JPA History 8

JPA Providers 8

2. JPA – ARCHITECTURE 9

Class Level Architecture 9

JPA Class Relationships 10

3. JPA – ORM COMPONENTS 12

Object Relational Mapping 12

Advanced Features 12

ORM Architecture 12

Mapping.xml 14

Annotations 17

Java Bean Standard 19

Bean Conventions 19

4. JPA – INSTALLATION 21

tutorialspoint
SIMPLYEASYLEARNING

Step 1 : Verify your Java Installation **21**

Step 2 : Set your Java Environment **22**

Step 3 : Installing JPA **22**

Adding MySQL connector to the Project **27**

5. JPA – ENTITY MANAGERS 29

Creating Entities **30**

Persistence Operations **34**

Create Employee **34**

Update Employee **35**

Deleting Employee **38**

6. JPA – JPQL 40

Java Persistence Query language **40**

Query Structure **40**

Scalar and Aggregate Functions **41**

Between, And, Like Keywords **43**

Ordering **45**

Named Queries **46**

Eager and Lazy Fetching **50**

7. JPA – ADVANCED MAPPINGS 51

Inheritance Strategies **51**

Single Table strategy **51**

Joined Table Strategy **58**

Table per Class Strategy **66**

8. JPA – ENTITY RELATIONSHIPS 74

@ManyToOne Relation **74**

tutorialspoint
SIMPLYEASYLEARNING

@OneToMany Relation **81**

@OneToOne Relation **89**

@ManyToMany Relation **96**

9. JPA – CRITERIA API 106

History of Criteria **106**

Criteria Query Structure **106**

Example of Criteria API **107**

tutorialspoint
SIMPLYEASYLEARNING

1. JPA – INTRODUCTION

Any enterprise application performs database operations by storing and retrieving vast amounts of data. Despite all the available technologies for storage management, application developers normally struggle to perform database operations efficiently.

Generally, Java developers use lots of code, or use the proprietary framework to interact with the database, whereas using JPA, the burden of interacting with the database reduces significantly. It forms a bridge between object models (Java program) and relational models (database program).

Mismatch between Relational and Object Models

Relational objects are represented in a tabular format, while object models are represented in an interconnected graph of object format. While storing and retrieving an object model from a relational database, some mismatch occurs due to the following reasons:

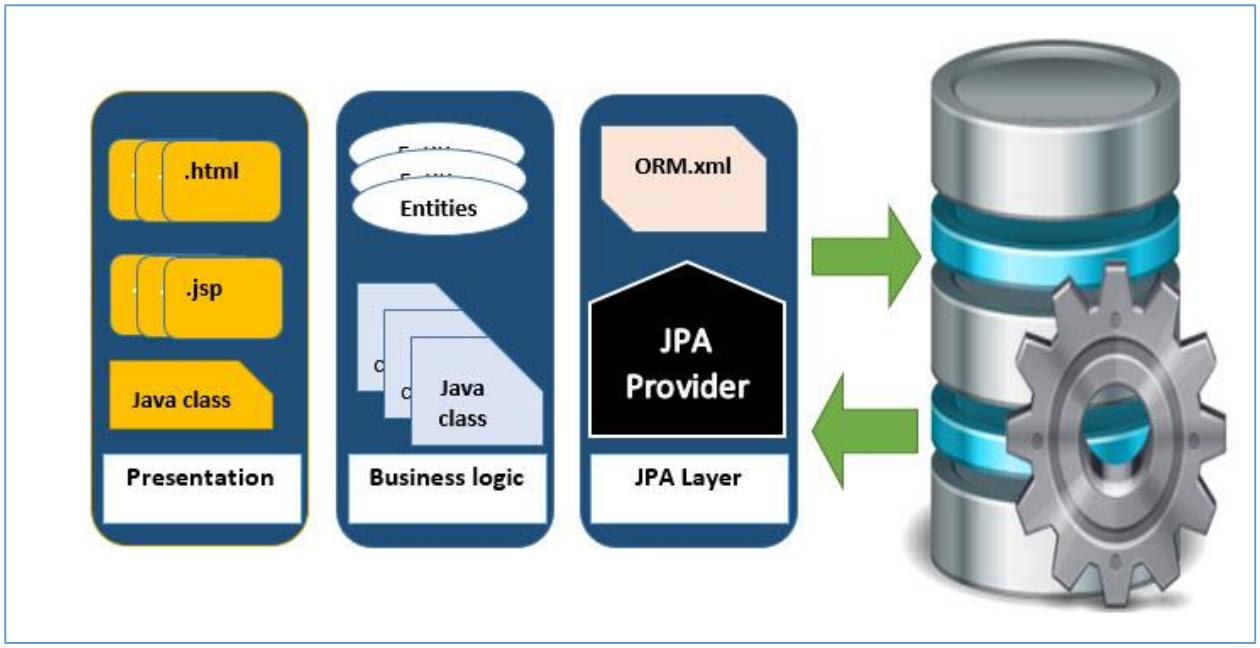
- **Granularity**: Object model has more granularity than relational model.
- **Subtypes**: Subtypes (means inheritance) are not supported by all types of relational databases.
- **Identity**: Like object model, relational model does not expose identity while writing equality.
- **Associations**: Relational models cannot determine multiple relationships while looking into an object domain model.
- **Data navigation**: Data navigation between objects in an object network is different in both models.

What is JPA?

Java Persistence API is a collection of classes and methods to persistently store the vast amounts of data into a database which is provided by the Oracle Corporation.

Where to use JPA?

To reduce the burden of writing codes for relational object management, a programmer follows the 'JPA Provider' framework, which allows easy interaction with database instance. Here the required framework is taken over by JPA.



JPA History

Earlier versions of EJB, defined the persistence layer combined with the business logic layer using **javax.ejb.EntityBean** Interface.

- While introducing EJB 3.0, the persistence layer was separated and specified as JPA 1.0 (Java Persistence API). The specifications of this API were released along with the specifications of JAVA EE5 on May 11, 2006 using JSR 220.
- JPA 2.0 was released with the specifications of JAVA EE6 on December 10, 2009 as a part of Java Community Process JSR 317.
- JPA 2.1 was released with the specification of JAVA EE7 on April 22, 2013 using JSR 338.

JPA Providers

JPA is an open source API, therefore various enterprise vendors such as Oracle, Redhat, Eclipse, etc. provide new products by adding the JPA persistence flavor in them. Some of these products include:

- Hibernate
- Eclipselink
- Toplink
- Spring Data JPA

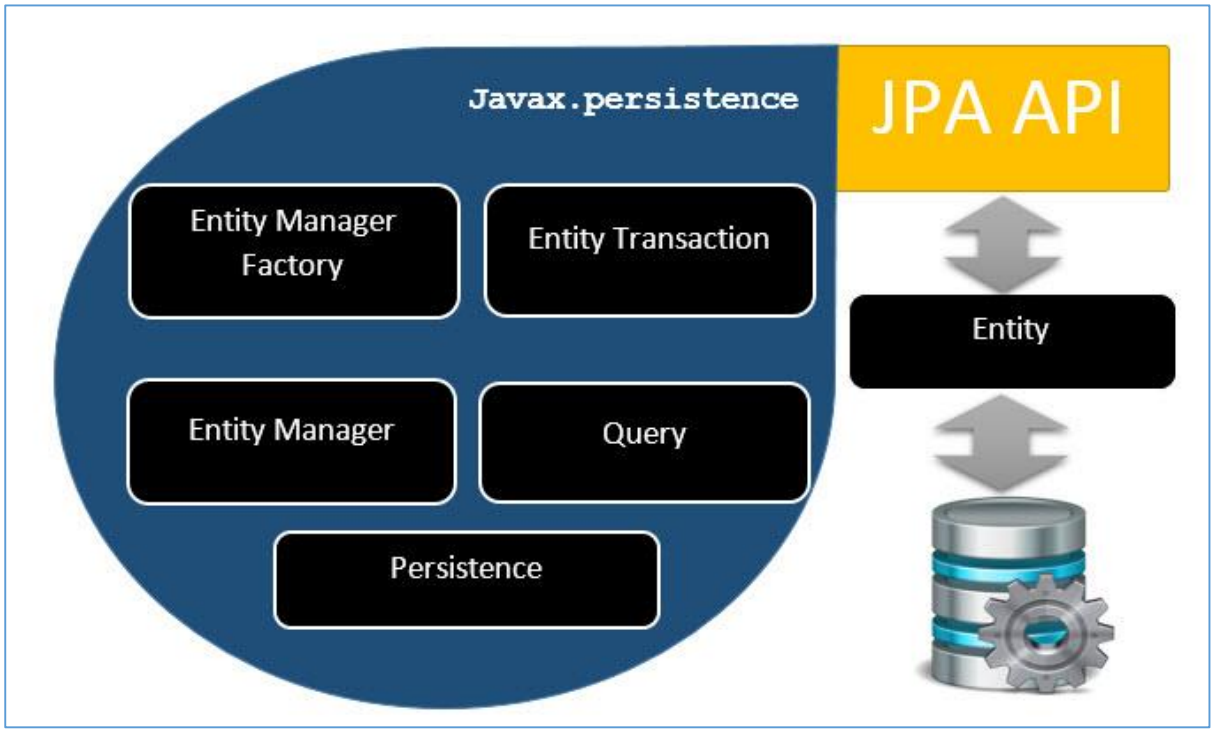
tutorialspoint
SIMPLYEASYLEARNING

2. JPA – ARCHITECTURE

Java Persistence API is a source to store business entities as relational entities. It shows how to define a Plain Oriented Java Object (POJO) as an entity and how to manage entities with relations.

Class Level Architecture

The following image shows the class level architecture of JPA. It shows the core classes and the interfaces of JPA.



The following table describes each of the units shown in the above architecture.

Units	Description
EntityManagerFactory	This is a factory class of EntityManager. It creates and manages multiple EntityManager instances.
EntityManager	It is an Interface. It manages the persistence operations on objects. It works like a factory for Query instance.

tutorialspoint
SIMPLYEASYLEARNING

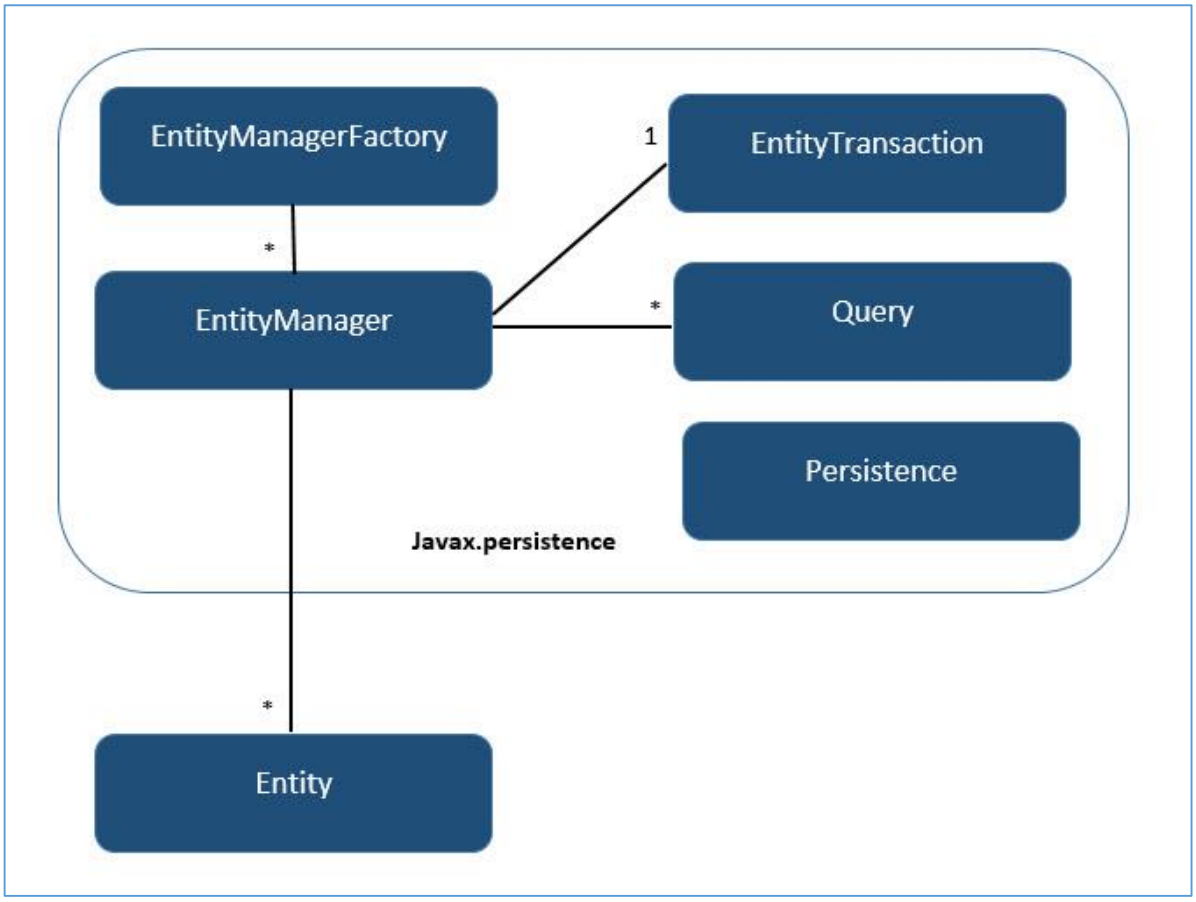
Entity	Entities are the persistence objects, stored as records in the database.
EntityTransaction	It has one-to-one relationship with the EntityManager. For each EntityManager, operations are maintained by the EntityTransaction class.
Persistence	This class contains static methods to obtain the EntityManagerFactory instance.
Query	This interface is implemented by each JPA vendor to obtain the relational objects that meet the criteria.

The above classes and interfaces are used for storing entities into a database as a record. They help programmers by reducing their efforts to write codes for storing data into a database so that they can concentrate on more important activities such as writing codes for mapping the classes with database tables.

JPA Class Relationships

In the above architecture, the relations between the classes and interfaces belong to the **javax.persistence** package. The following diagram shows the relationship between them.

tutorialspoint
SIMPLYEASYLEARNING



- The relationship between EntityManagerFactory and EntityManager is one-to-many. It is a factory class to EntityManager instances.
- The relationship between EntityManager and EntityTransaction is one-to-one. For each EntityManager operation, there is an EntityTransaction instance.
- The relationship between EntityManager and Query is one-to-many. A number of queries can execute using one EntityManager instance.
- The relationship between EntityManager and Entity is one-to-many. One EntityManager instance can manage multiple Entities.

tutorialspoint
SIMPLYEASYLEARNING

3. JPA – ORM COMPONENTS

Most contemporary applications use relational database to store data. Recently, many vendors switched to object database to reduce their burden on data maintenance. It means object database or object relational technologies are taking care of storing, retrieving, updating, and maintaining data. The core part of this object relational technology is mapping **orm.xml** files. As xml does not require compilation, we can easily make changes to multiple data sources with less administration.

Object Relational Mapping

Object Relational Mapping (ORM) briefly tells you about what is ORM and how it works. ORM is a programming ability to covert data from object type to relational type and vice versa.

The main feature of ORM is mapping or binding an object to its data in the database. While mapping, we have to consider the data, the type of data, and its relations with self-entity or entities in any other table.

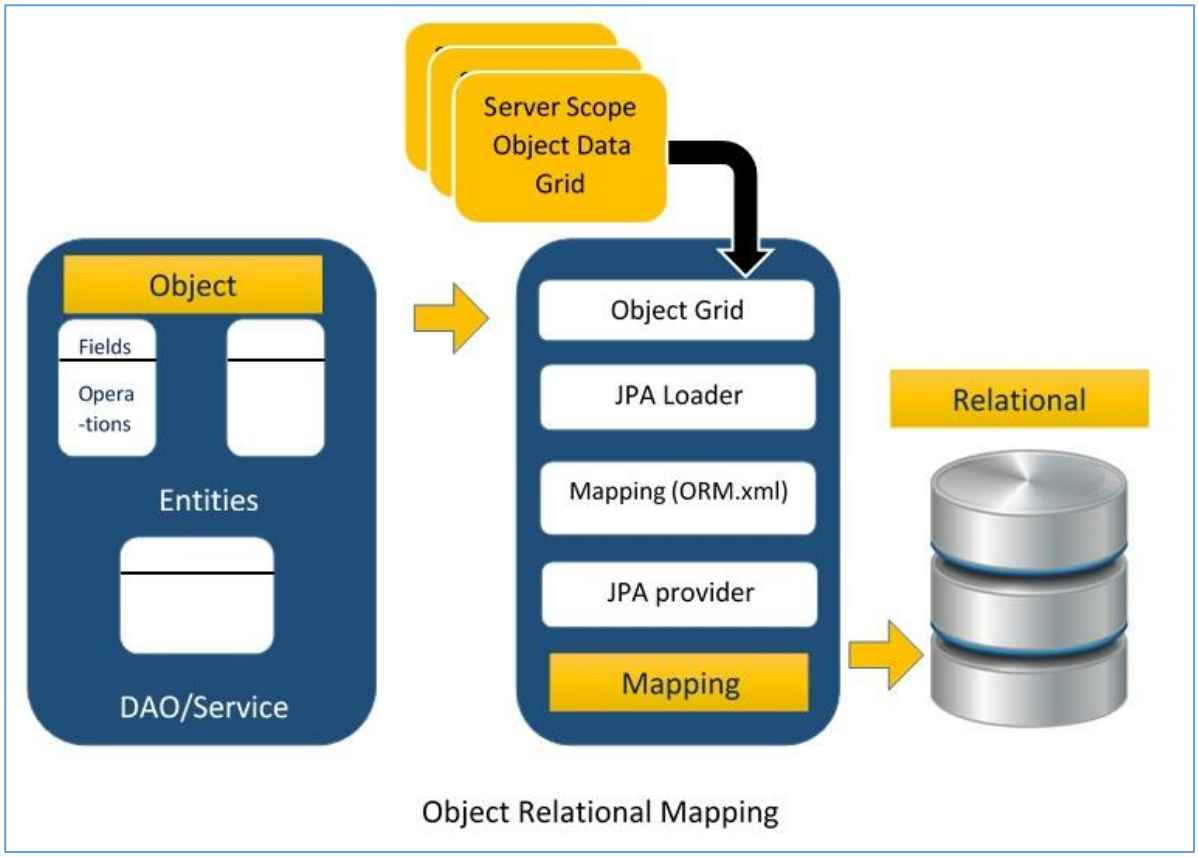
Advanced Features

- Idiomatic persistence: It enables you to write persistence classes using object oriented classes.
- High Performance: It has many fetching techniques and helpful locking techniques.
- Reliable: It is highly stable and used by many professional programmers.

ORM Architecture

The ORM architecture looks as follows.

tutorialspoint
SIMPLYEASYLEARNING



The above architecture explains how object data is stored into a relational database in three phases.

Phase 1

The first phase, named as the **object data phase**, contains POJO classes, service interfaces, and classes. It is the main business component layer, which has business logic operations and attributes.

For example, let us take an employee database as a schema.

- Employee POJO class contains attributes such as ID, name, salary, and designation. It also contains methods like setter and getter of those attributes.
- Employee DAO/Service classes contain service methods such as create employee, find employee, and delete employee.

Phase 2

The second phase, named as **mapping** or **persistence phase**, contains JPA provider, mapping file (ORM.xml), JPA Loader, and Object Grid.

- **JPA Provider**: It is the vendor product that contains the JPA flavor (javax.persistence). For example Eclipselink, Toplink, Hibernate, etc.

tutorialspoint
SIMPLYEASYLEARNING

- **Mapping file**: The mapping file (ORM.xml) contains mapping configuration between the data in a POJO class and data in a relational database.

- **JPA Loader**: The JPA loader works like a cache memory. It can load the relational grid data. It works like a copy of database to interact with service classes for POJO data (attributes of POJO class).

- **Object Grid**: It is a temporary location that can store a copy of relational data, like a cache memory. All queries against the database is first effected on the data in the object grid. Only after it is committed, it affects the main database.

Phase 3

The third phase is the **relational data phase**. It contains the relational data that is logically connected to the business component. As discussed above, only when the business component commits the data, it is stored into the database physically. Until then, the modified data is stored in a cache memory as a grid format. The process of the obtaining the data is identical to that of storing the data.

The mechanism of the programmatic interaction of the above three phases is called as **object relational mapping**.

Mapping.xml

The mapping.xml file instructs the JPA vendor to map the entity classes with the database tables.

Let us take an example of Employee entity that contains four attributes. The POJO class of Employee entity named **Employee.java** is as follows:

```
public class Employee
{
      private int eid;
      private String ename;
      private double salary;
      private String deg;
      public Employee(int eid, String ename, double salary, String deg)
      {
            super( );
            this.eid = eid;
            this.ename = ename;
            this.salary = salary;
            this.deg = deg;
```


tutorialspoint
SIMPLYEASYLEARNING

```
}

public Employee( )
{
	super();
}

public int getEid( )
{
	return eid;
}
public void setEid(int eid)
{
	this.eid = eid;
}

public String getEname( )
{
	return ename;
}
public void setEname(String ename)
{
	this.ename = ename;
}

public double getSalary( )
{
	return salary;
}
public void setSalary(double salary)
{
	this.salary = salary;
}

public String getDeg( )
```


tutorialspoint
SIMPLYEASYLEARNING

```
    {
        return deg;
    }
    public void setDeg(String deg)
    {
        this.deg = deg;
    }
}
```

The above code is the Employee entity POJO class. It contain four attributes **eid**, **ename**, **salary**, and **deg**. Consider these attributes as the table fields in a table and **eid** as the primary key of this table. Now we have to design the hibernate mapping file for it. The mapping file named **mapping.xml** is as follows:

```
<? xml version="1.0" encoding="UTF-8" ?>
<entity-mappings xmlns="http://java.sun.com/xml/ns/persistence/orm"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence/orm
                        http://java.sun.com/xml/ns/persistence/orm_1_0.xsd"
                        version="1.0">
    <description> XML Mapping file</description>
    <entity class="Employee">
        <table name="EMPLOYEETABLE"/>
        <attributes>
            <id name="eid">
                <generated-value strategy="TABLE"/>
            </id>
            <basic name="ename">
                <column name="EMP_NAME" length="100"/>
            </basic>
            <basic name="salary">
            </basic>
            <basic name="deg">
            </basic>
        </attributes>
    </entity>
</entity-mappings>
```


tutorialspoint
SIMPLYEASYLEARNING

The above script is used for mapping the entity class with the database table. In this file,

- <entity-mappings> tag defines the schema definition to allow entity tags into the xml file.
- <description> tag provides a description of the application.
- <entity> tag defines the entity class which you want to convert into a table in a database. Attribute class defines the POJO entity class name.
- <table> tag defines the table name. If you want to have identical names for both the class as well as the table, then this tag is not necessary.
- <attributes> tag defines the attributes (fields in a table).
- <id> tag defines the primary key of the table. The <generated-value> tag defines how to assign the primary key value such as Automatic, Manual, or Taken from Sequence.
- <basic> tag is used for defining the remaining attributes of the table.
- <column-name> tag is used to set user-defined field names in the table.

Annotations

Generally xml files are used to configure specific components, or mapping two different specifications of components. In our case, we have to maintain xml files separately in a framework. That means while writing a mapping xml file, we need to compare the POJO class attributes with entity tags in the mapping.xml file.

Here is the solution. In the class definition, we can write the configuration part using annotations. Annotations are used for classes, properties, and methods. Annotations start with '**@**' symbol. Annotations are declared prior to a class, property, or method. All annotations of JPA are defined in the **javax.persistence** package.

The list of annotations used in our examples are given below.

Annotation	Description
@Entity	Declares the class as an entity or a table.
@Table	Declares table name.
@Basic	Specifies non-constraint fields explicitly.
@Embedded	Specifies the properties of class or an entity whose value is an instance of an embeddable class.

tutorialspoint
SIMPLYEASYLEARNING

@Id	Specifies the property, use for identity (primary key of a table) of the class.
@GeneratedValue	Specifies how the identity attribute can be initialized such as automatic, manual, or value taken from a sequence table.
@Transient	Specifies the property that is not persistent, i.e., the value is never stored in the database.
@Column	Specifies the column attribute for the persistence property.
@SequenceGenerator	Specifies the value for the property that is specified in the @GeneratedValue annotation. It creates a sequence.
@TableGenerator	Specifies the value generator for the property specified in the @GeneratedValue annotation. It creates a table for value generation.
@AccessType	This type of annotation is used to set the access type. If you set @AccessType(FIELD), then access occurs Field wise. If you set @AccessType(PROPERTY), then access occurs Property wise.
@JoinColumn	Specifies an entity association or entity collection. This is used in many- to-one and one-to-many associations.
@UniqueConstraint	Specifies the fields and the unique constraints for the primary or the secondary table.
@ColumnResult	References the name of a column in the SQL query using select clause.
@ManyToMany	Defines a many-to-many relationship between the join Tables.
@ManyToOne	Defines a many-to-one relationship between the join

tutorialspoint
SIMPLYEASYLEARNING

	Tables.
@OneToMany	Defines a one-to-many relationship between the join Tables.
@OneToOne	Defines a one-to-one relationship between the join Tables.
@NamedQueries	Specifies a list of named queries.
@NamedQuery	Specifies a Query using static name.

Java Bean Standard

The Java class encapsulates the instance values and their behaviors into a single unit called object. Java Bean is a temporary storage and reusable component or an object. It is a serializable class which has a default constructor and getter and setter methods to initialize the instance attributes individually.

Bean Conventions

- Bean contains its default constructor or a file that contains a serialized instance. Therefore, a bean can instantiate another bean.
- The properties of a bean can be segregated into Boolean properties or non-Boolean properties.
- Non-Boolean property contains **getter** and **setter** methods.
- Boolean property contains **setter** and **is** method.
- **Getter** method of any property should start with small lettered **get** (Java method convention) and continued with a field name that starts with a capital letter. For example, the field name is **salary**, therefore the getter method of this field is **getSalary ()**.
- **Setter** method of any property should start with small lettered **set** (Java method convention), continued with a field name that starts with a capital letter and the **argument value** to set to field. For example, the field name is **salary**, therefore the setter method of this field is **setSalary (double sal)**.

tutorialspoint
SIMPLYEASYLEARNING

- For Boolean property, the **is** method is used to check if it is true or false. For example, for the Boolean property **empty**, the **is** method of this field is **isEmpty ()**.

tutorialspoint
SIMPLYEASYLEARNING

4. JPA – INSTALLATION

This chapter takes you through the process of setting up JPA on Windows and Linux based systems. JPA can be easily installed and integrated with your current Java environment following a few simple steps without any complex setup procedures. User administration is required while installation.

System Requirements

JDK	Java SE 2 JDK 1.5 or above
Memory	1 GB RAM (recommended)
Disk Space	No minimum requirement
Operating System Version	Windows XP or above, Linux

Let us now proceed with the steps to install JPA.

Step 1 : Verify your Java Installation

First of all, you need to have Java Software Development Kit (SDK) installed on your system. To verify this, execute any of the following two commands depending on the platform you are working on.

If the Java installation has been done properly, then it will display the current version and specification of your Java installation. A sample output is given in the following table.

Platform	Command	Sample Output
Windows	Open command console and type: **\>java –version**	Java version "1.7.0_60" Java (TM) SE Run Time Environment (build 1.7.0_60-b19) Java Hotspot (TM) 64-bit Server VM (build 24.60-b09,mixed mode)
Linux	Open command terminal and	java version "1.7.0_25"

tutorialspoint
SIMPLYEASYLEARNING

	type: **$java –version**	Open JDK Runtime Environment (rhel-2.3.10.4.el6_4-x86_64) Open JDK 64-Bit Server VM (build 23.7-b01, mixed mode)

- We assume the readers of this tutorial have Java SDK version 1.7.0_60 installed on their system.

- In case you do not have Java SDK, download its current version from http://www.oracle.com/technetwork/java/javase/downloads/index.html and have it installed.

Step 2 : Set your Java Environment

Set the environment variable JAVA_HOME to point to the base directory location where Java is installed on your machine. For example,

Platform	Description
Windows	Set JAVA_HOME to C:\ProgramFiles\java\jdk1.7.0_60
Linux	Export JAVA_HOME=/usr/local/java-current

Append the full path of the Java compiler location to the System Path.

Platform	Description
Windows	Append the String "C:\Program Files\Java\jdk1.7.0_60\bin" to the end of the system variable PATH.
Linux	Export PATH=$PATH:$JAVA_HOME/bin/

Execute the command **java -version** from the command prompt as explained above.

Step 3 : Installing JPA

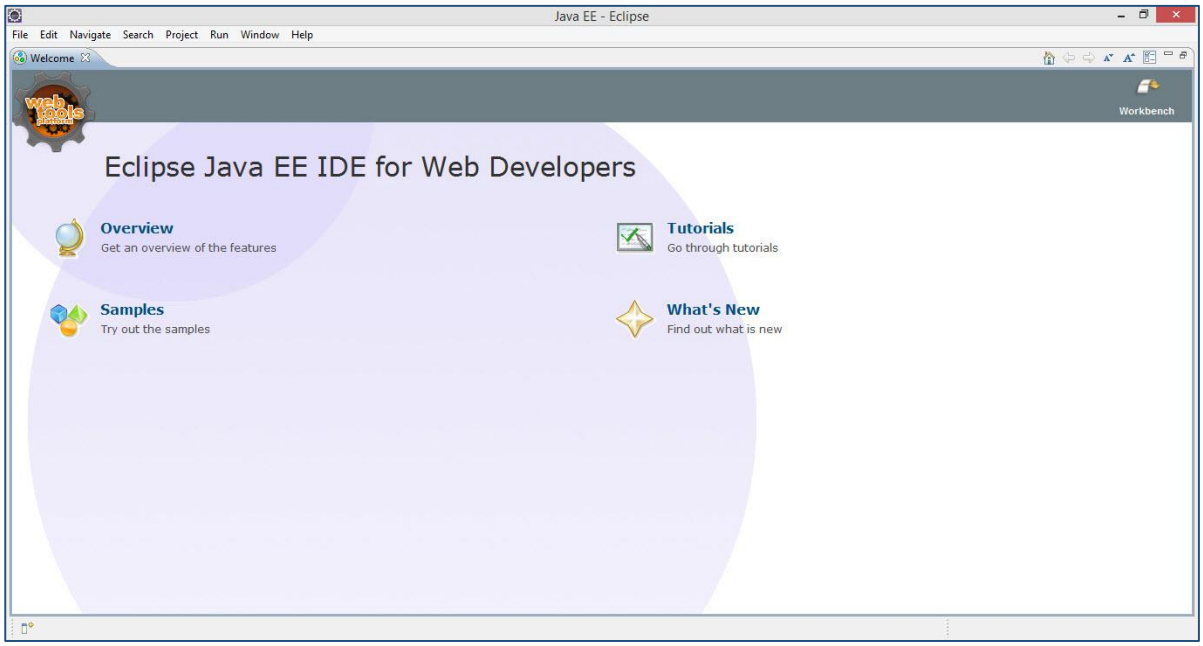
You can go through the JPA installation by using any of the JPA Providers from this tutorial, e.g., Eclipselink, Hibernate. Let us follow the JPA installation using Eclipselink.

tutorialspoint
SIMPLYEASYLEARNING

For JPA programming, we require to follow the specific folder framework, therefore it is better to use IDE.

Download Eclipse IDE form following link https://www.eclipse.org/downloads/. Choose the EclipseIDE for JavaEE developer, that is, Eclipse indigo.

Unzip the Eclipse zip file on to your C drive. Open Eclipse IDE.

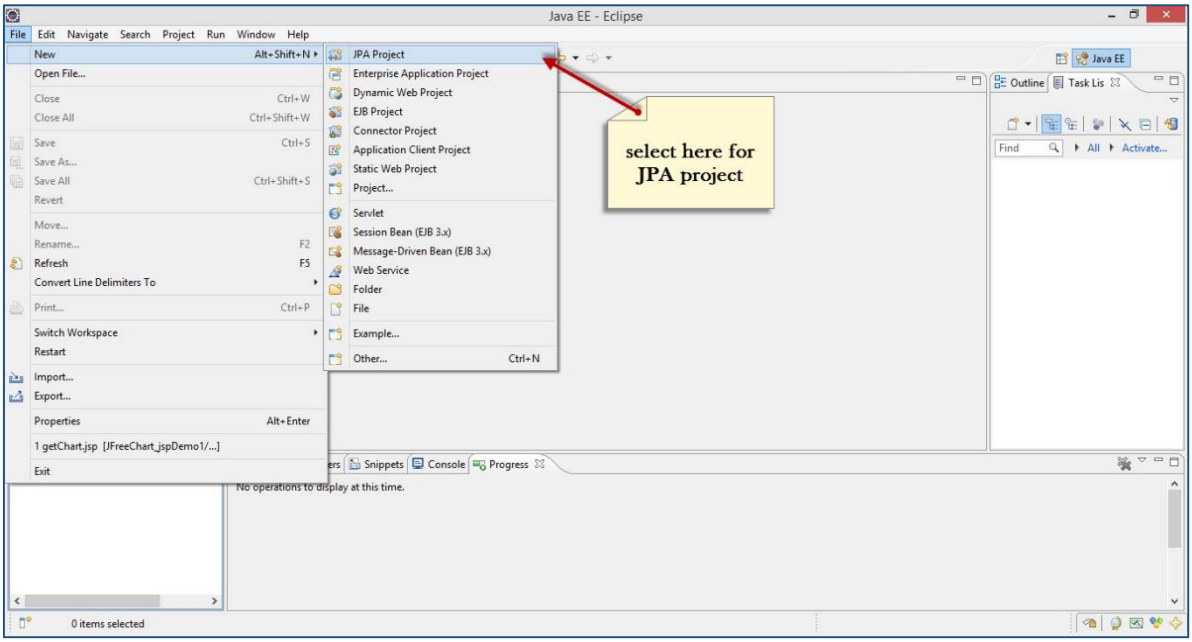


Installing JPA using Eclipselink

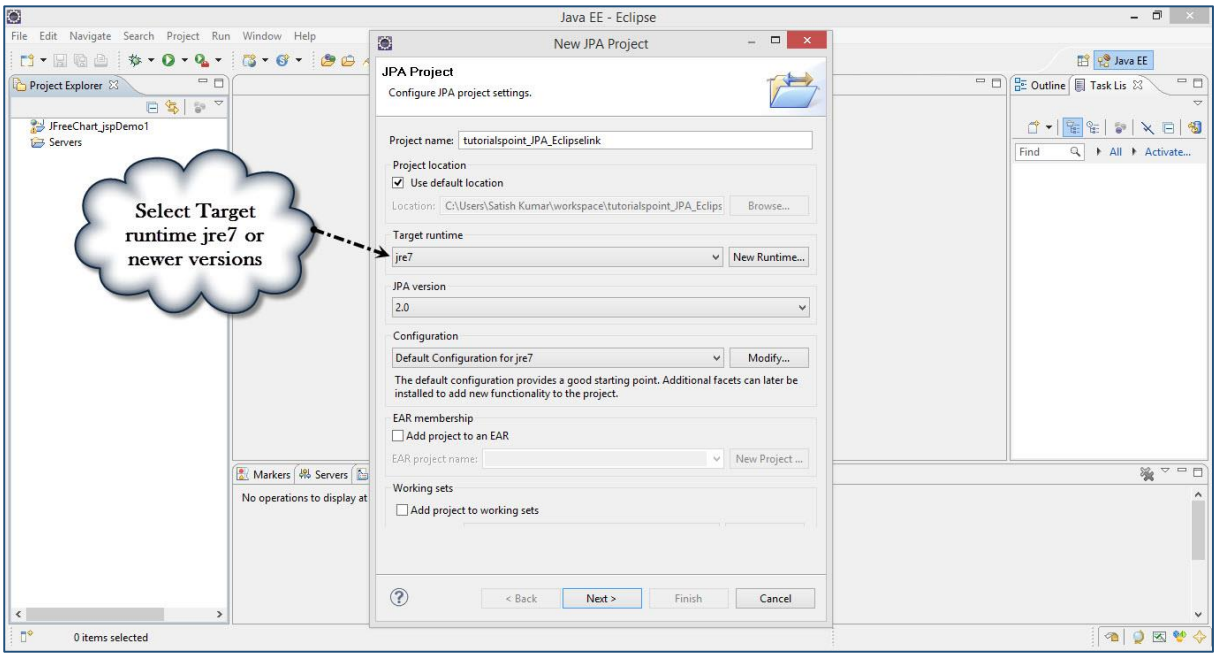
Eclipselink is a library, therefore we cannot add it directly to Eclipse IDE. For installing JPA using Eclipselink, you need to follow the steps given below.

1. Create a new JPA project by selecting **File->New->JPA Project** in the Eclipse IDE as follows:

tutorialspoint
SIMPLYEASYLEARNING

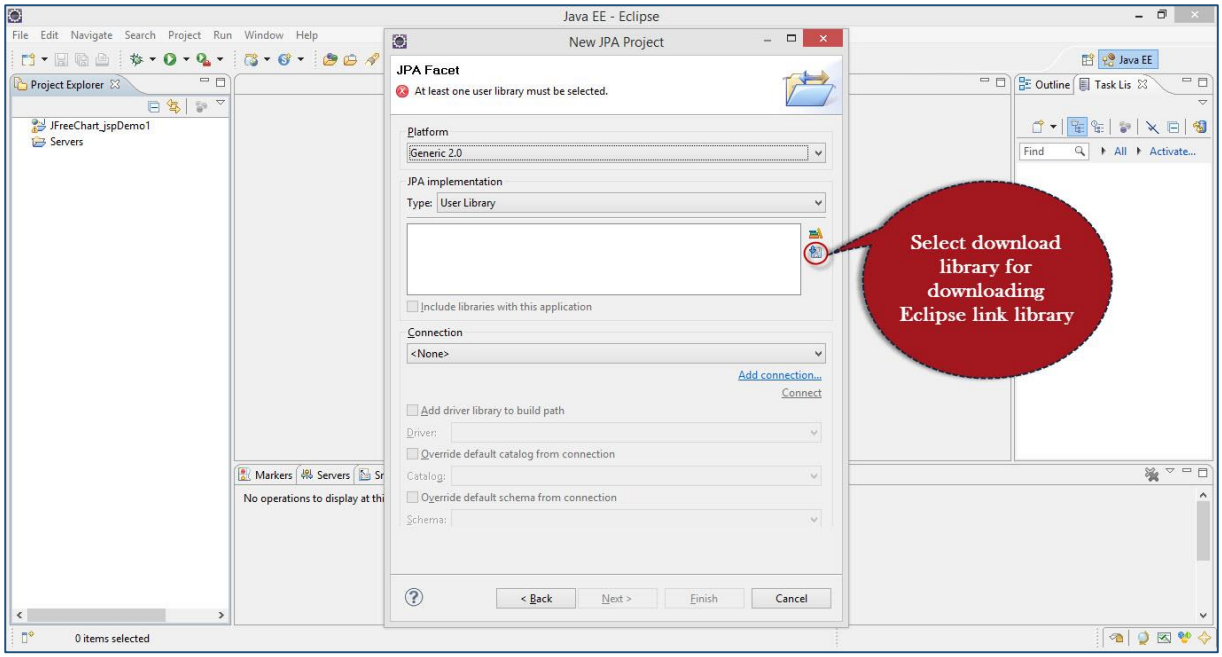


2. You will get a dialog box named **New JPA Project**. Enter the project name as **tutorialspoint_JPA_Eclipselink**, check the **jre** version and click next.

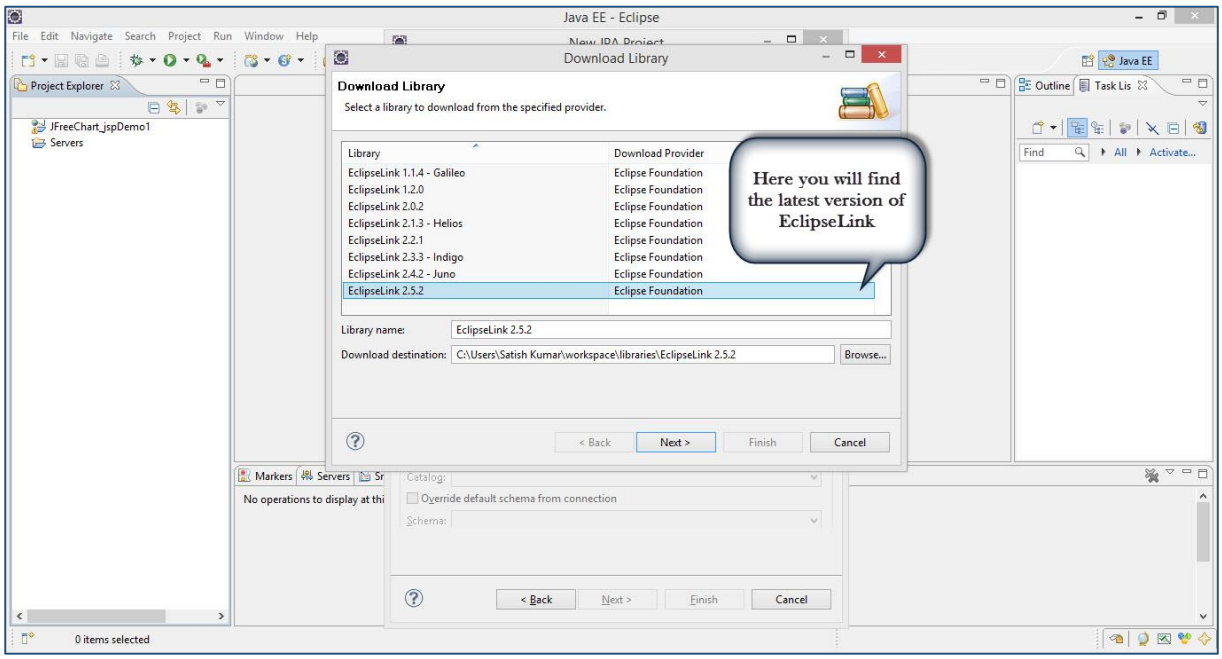


3. Click on download library (if you do not have the library) in the user library section.

tutorialspoint
SIMPLYEASYLEARNING

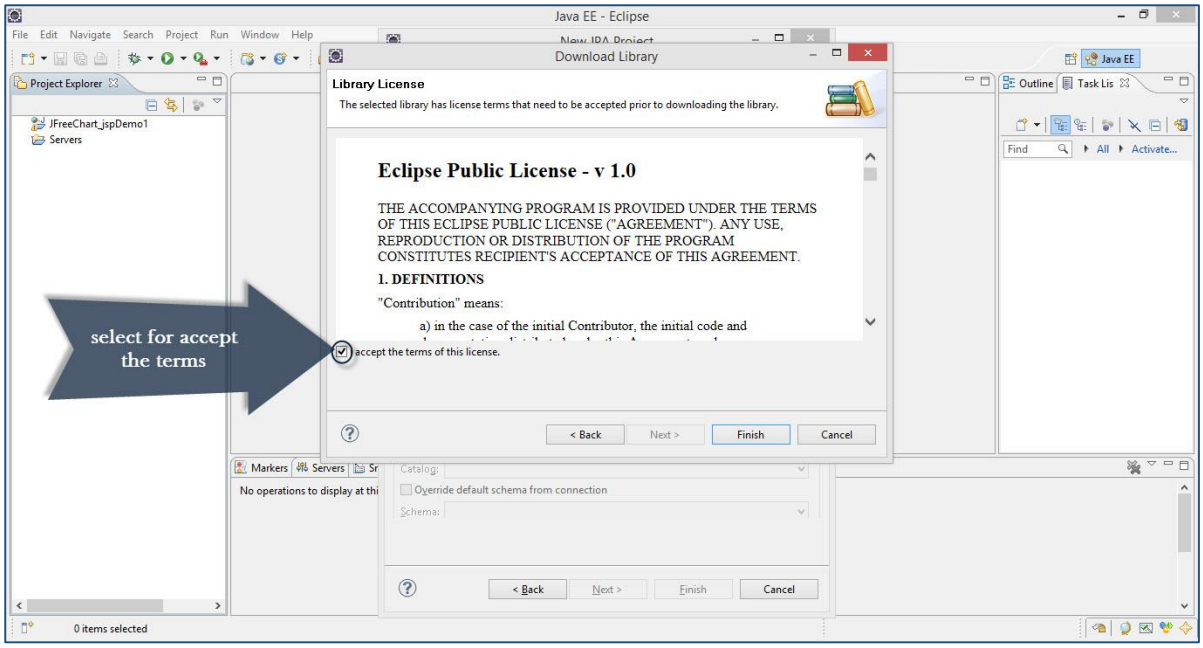


4. Select the latest version of Eclipselink library in the Download library dialog box and click next.

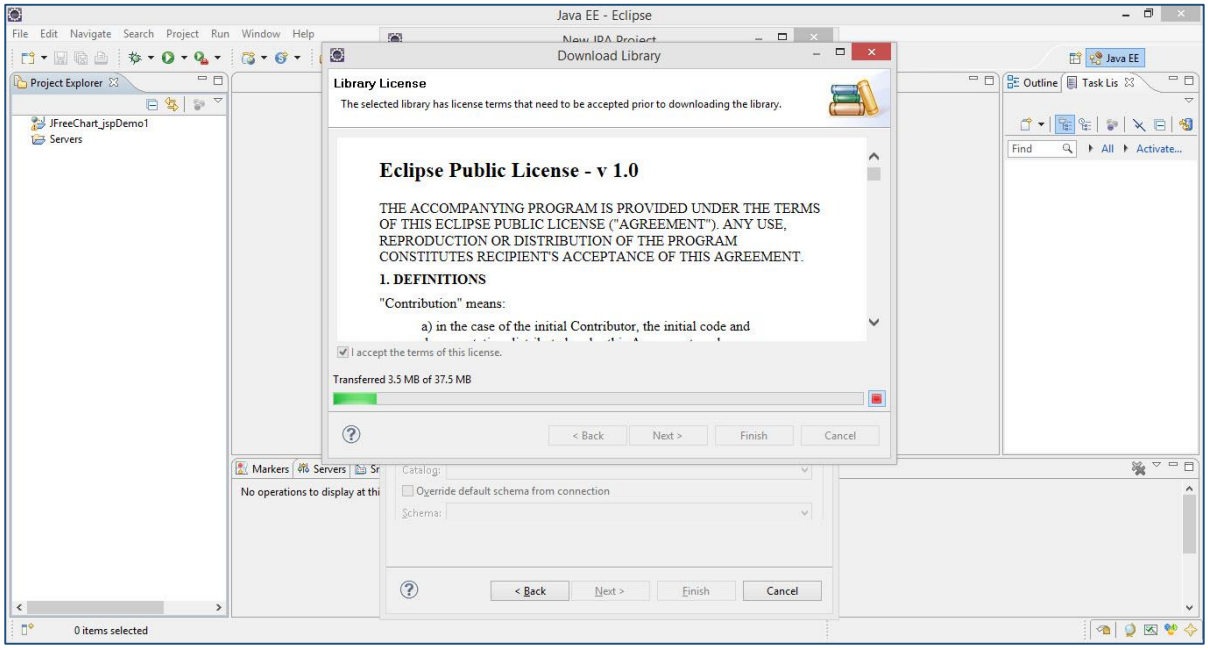


5. Accept the terms of license and click finish for download.

tutorialspoint
SIMPLYEASYLEARNING

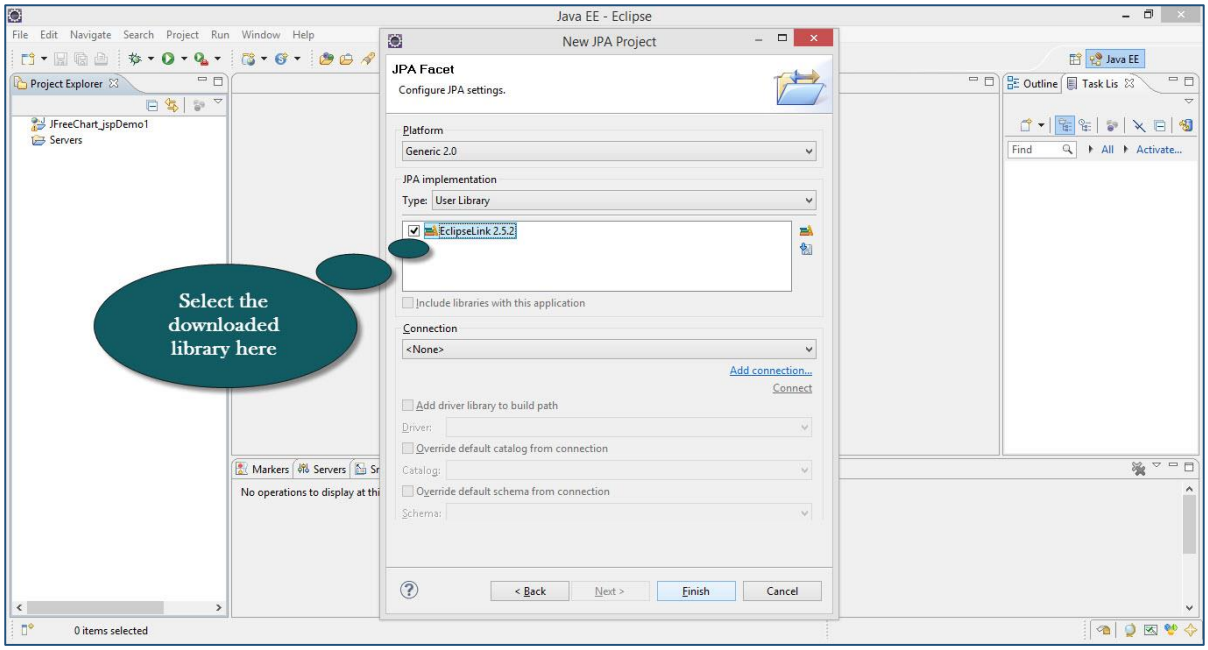


6. Downloading starts as is shown in the following screenshot.

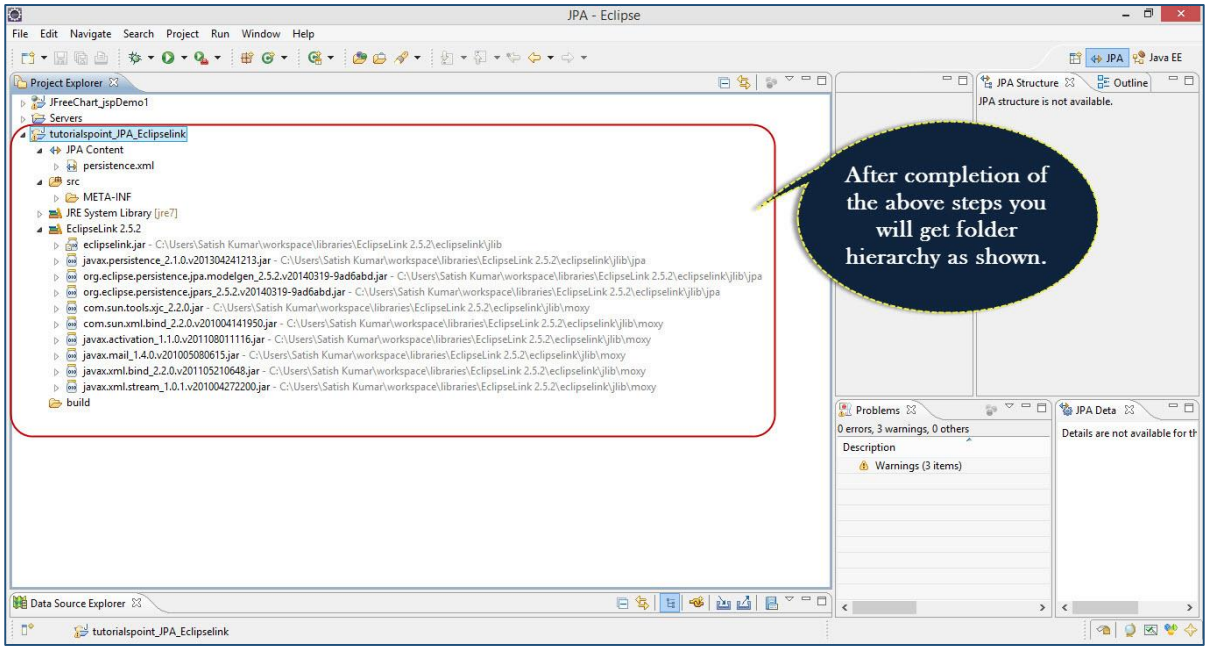


7. After downloading, select the downloaded library in the user library section and click finish.

tutorialspoint
SIMPLYEASYLEARNING



8. Finally you get the project file in the **Package Explorer** in the Eclipse IDE. Extract all the files, you will get the folder and file hierarchy as follows:



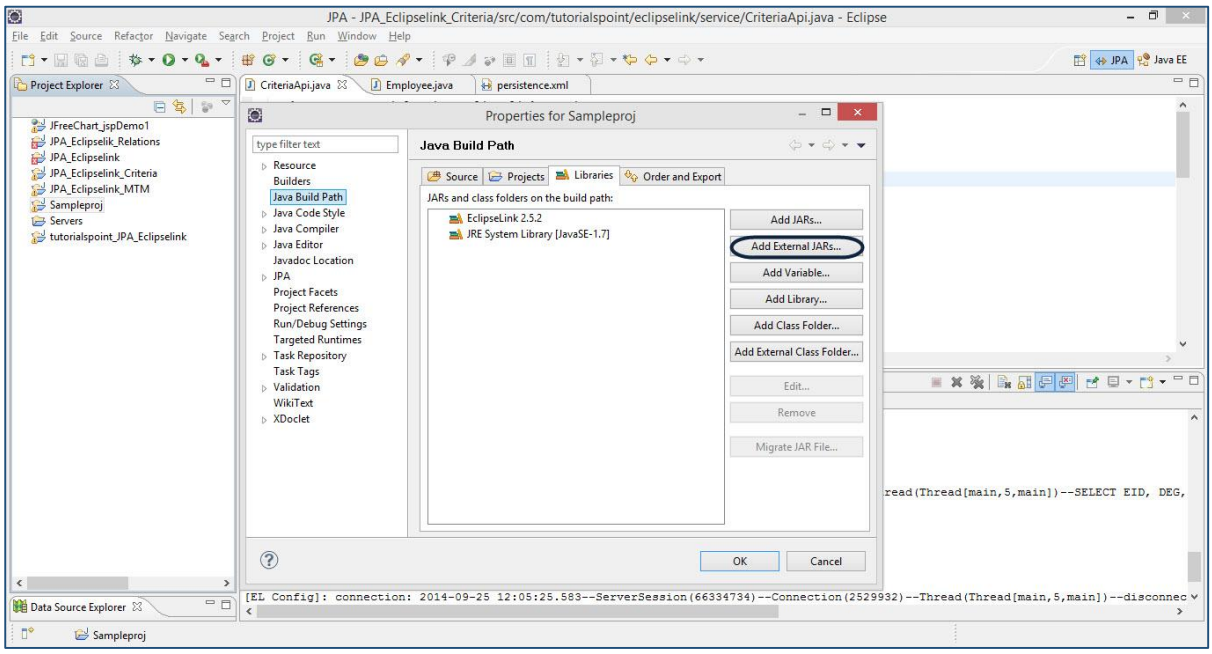
Adding MySQL connector to the Project

Any example that we discuss here requires database connectivity. Let us consider MySQL database for database operations. It requires **mysql-connector** jar to interact with a Java program.

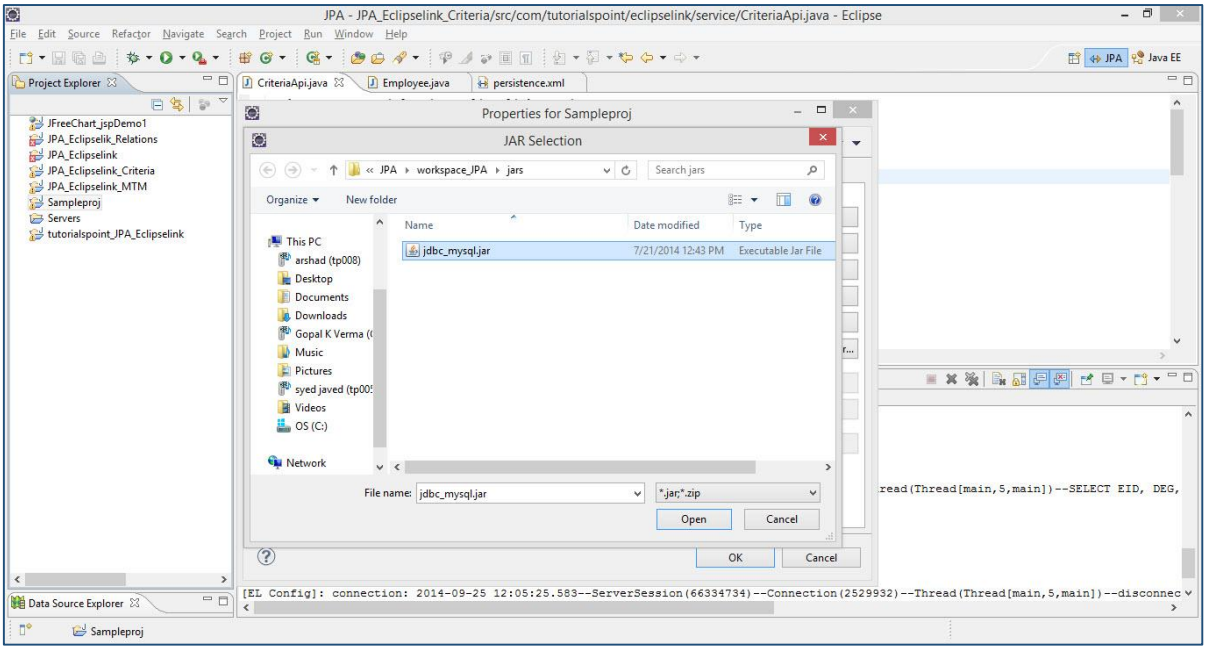
Follow the given steps to configure the database jar in your project.

tutorialspoint
SIMPLYEASYLEARNING

1. Go to Project properties -> Java Build Path by right clicking on it. You will get a dialog box as shown in the following screenshot. Click Add External Jars.



2. Go to the jar location in your system memory, select the file and click open.



3. Click ok on properties dialog. You will get the MySQL-connector jar into your project. Now you can perform database operations using MySQL.

tutorialspoint
SIMPLYEASYLEARNING

5. JPA – ENTITY MANAGERS

This chapter uses a simple example to demonstrate how JPA works. Let us consider Employee Management as an example. Suppose the Employee Management creates, updates, finds, and deletes the records of an employee. As mentioned, we are using MySQL database for database operations.

The main modules for this example are as follows:

1. **Model or POJO**

   Employee.java

2. **Persistence**

   Persistence.xml

3. **Service**

   CreatingEmployee.java

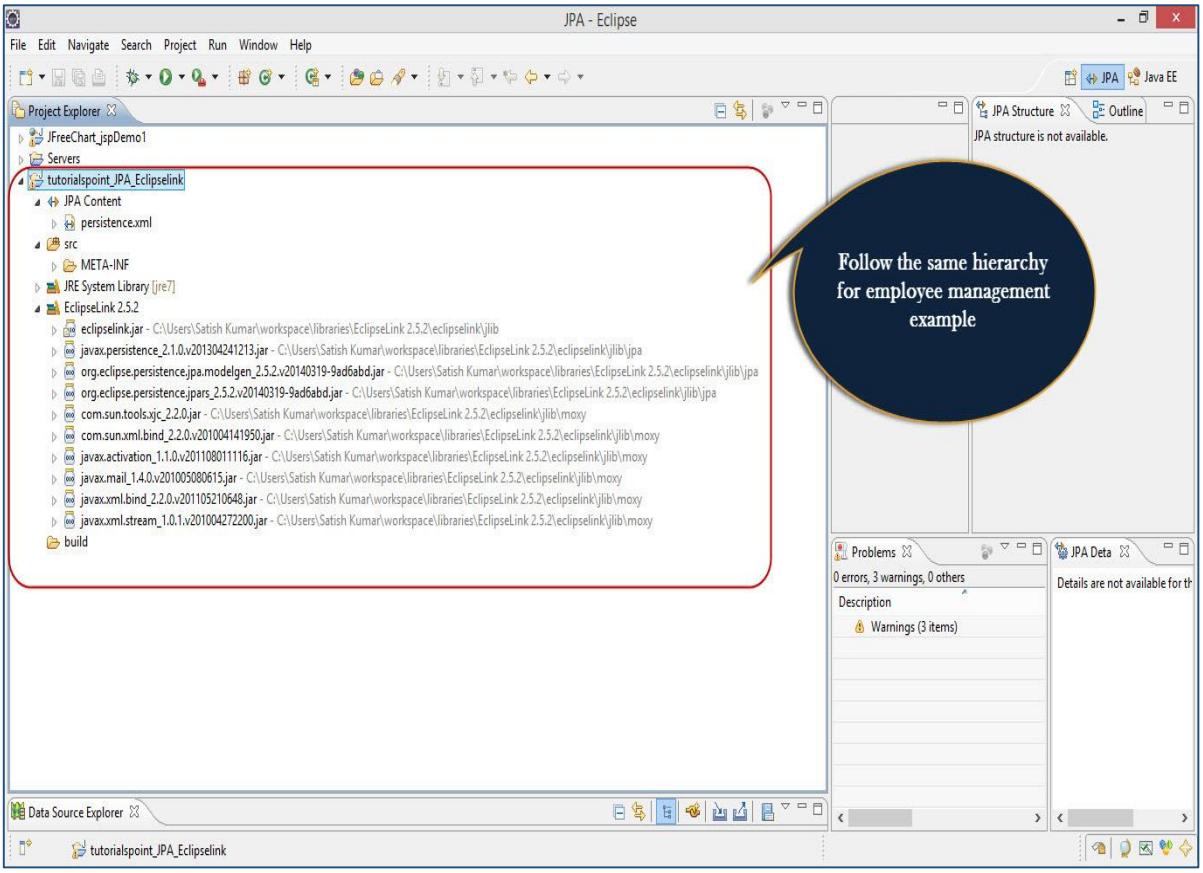
   UpdatingEmployee.java

   FindingEmployee.java

   DeletingEmployee.java

Let us take the package hierarchy which we have used in the JPA installation with Eclipselink. For this example, follow the hierarchy as shown below.

tutorialspoint
SIMPLYEASYLEARNING



Creating Entities

Entities are nothing but beans or models. In this example, we will use **Employee** as an entity. **eid, ename, salary,** and **deg** are the attributes of this entity. It contains a default constructor as well as the setter and getter methods of those attributes.

In the above shown hierarchy, create a package named **com.tutorialspoint.eclipselink.entity**, under **src** (Source) package. Create a class named **Employee.java** under the given package as follows:

```
package com.tutorialspoint.eclipselink.entity;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;
```


tutorialspoint
SIMPLYEASYLEARNING

```
@Entity
@Table
public class Employee
{
      @Id
      @GeneratedValue(strategy= GenerationType.AUTO)
      private int eid;
      private String ename;
      private double salary;
      private String deg;
      public Employee(int eid, String ename, double salary, String deg)
      {
            super( );
            this.eid = eid;
            this.ename = ename;
            this.salary = salary;
            this.deg = deg;
      }

      public Employee( )
      {
            super();
      }

      public int getEid( )
      {
            return eid;
      }
      public void setEid(int eid)
      {
            this.eid = eid;
      }

      public String getEname( )
      {
```


tutorialspoint
SIMPLYEASYLEARNING

```
            return ename;
      }
      public void setEname(String ename)
      {
            this.ename = ename;
      }

      public double getSalary( )
      {
            return salary;
      }
      public void setSalary(double salary)
      {
            this.salary = salary;
      }

      public String getDeg( )
      {
            return deg;
      }
      public void setDeg(String deg)
      {
            this.deg = deg;
      }
      @Override
      public String toString() {
            return "Employee [eid=" + eid + ", ename=" + ename + ", salary="
                        + salary + ", deg=" + deg + "]";
      }
}
```

In the above code, we have used @Entity annotation to make this POJO class an entity.

Before going to the next module, we need to create a database for relational entity, which will register the database in the **persistence.xml** file. Open MySQL workbench and type the following query.

tutorialspoint
SIMPLYEASYLEARNING

```
create database jpadb
use jpadb
```

Persistence.xml

This module plays a crucial role in the concept of JPA. In this xml, file we will register the database and specify the entity class.

In the above shown package hierarchy, persistence.xml under JPA Content package is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0" 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">
     <persistence-unit name="Eclipselink_JPA"
                       transaction-type="RESOURCE_LOCAL">

     <class>com.tutorialspoint.eclipselink.entity.Employee</class>
     <properties>
        <property name="javax.persistence.jdbc.url"
                  value="jdbc:mysql://localhost:3306/jpadb"/>
        <property name="javax.persistence.jdbc.user" value="root"/>
        <property name="javax.persistence.jdbc.password" value="root"/>
        <property name="javax.persistence.jdbc.driver"
                  value="com.mysql.jdbc.Driver"/>
        <property name="eclipselink.logging.level" value="FINE"/>
        <property name="eclipselink.ddl-generation" value="create-tables"/>
     </properties>
     </persistence-unit>
</persistence>
```

In the above xml, the <persistence-unit> tag is defined with a specific name for JPA persistence. The <class> tag defines the entity class with the package name. The <properties> tag defines all the properties, and the <property> tag defines each property such as database registration, URL specification, username, and password. These are the Eclipselink properties. This file will configure the database.

tutorialspoint
SIMPLYEASYLEARNING

Persistence Operations

Persistence operations are used for interacting with a database and they are **load** and **store** operations. In a business component, all the persistence operations fall under service classes.

In the above shown package hierarchy, create a package named **com.tutorialspoint.eclipselink.service** under the **src** (source) package. All the service classes are named as CreateEmloyee.java, UpdateEmployee.java, FindEmployee.java, and DeleteEmployee.java.

Create Employee

The following code segment shows how to create an Employee class named **CreateEmployee.java**.

```
package com.tutorialspoint.eclipselink.service;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import com.tutorialspoint.eclipselink.entity.Employee;

public class CreateEmployee
{
      public static void main( String[ ] args )
      {
            EntityManagerFactory emfactory = Persistence.
                        createEntityManagerFactory( "Eclipselink_JPA" );
            EntityManager entitymanager = emfactory.
                        createEntityManager( );
            entitymanager.getTransaction( ).begin( );

            Employee employee = new Employee( );
            employee.setEid( 1201 );
            employee.setEname( "Gopal" );
            employee.setSalary( 40000 );
            employee.setDeg( "Technical Manager" );
            entitymanager.persist( employee );
```


tutorialspoint
SIMPLYEASYLEARNING

```
            entitymanager.getTransaction( ).commit( );

            entitymanager.close( );
            emfactory.close( );
      }
}
```

In the above code, **createEntityManagerFactory()** creates a persistence unit by providing the same unique name which we provided for the persistence-unit in the persistent.xml file. The **entitymanagerfactory** object will create the **entitymanger** instance by using the **createEntityManager()** method. The **entitymanager** object creates the **entitytransaction** instance for transaction management. By using the **entitymanager** object, we can persist entities into the database.

After compiling and executing the above program, you will get notifications from the eclipselink library on the console panel of eclipse IDE. For result, open the MySQL workbench and type the following queries.

```
use jpadb
select * from employee
```

The effected database table named **employee** will be shown in a tabular format as follows:

Eid	Ename	Salary	Deg
1201	Gopal	40000	Technical Manager

Update Employee

To update the records of an employee, we need to retrieve the existing records form the database, make changes, and finally commit it to the database. The class named **UpdateEmployee.java** is shown as follows:

```
package com.tutorialspoint.eclipselink.service;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import com.tutorialspoint.eclipselink.entity.Employee;
```


tutorialspoint
SIMPLYEASYLEARNING

```
public class UpdateEmployee
{
      public static void main( String[ ] args )
      {
            EntityManagerFactory emfactory = Persistence.
                        createEntityManagerFactory( "Eclipselink_JPA" );
            EntityManager entitymanager = emfactory.
                        createEntityManager( );
            entitymanager.getTransaction( ).begin( );
            Employee employee=entitymanager.
                        find( Employee.class, 1201 );
            //before update
            System.out.println( employee );
            employee.setSalary( 46000 );
            entitymanager.getTransaction( ).commit( );
            //after update
            System.out.println( employee );
            entitymanager.close();
            emfactory.close();
      }
}
```

After compiling and executing the above program, you will get notifications from Eclipselink library on the console panel of eclipse IDE.

For result, open the MySQL workbench and type the following queries.

```
use jpadb
select * from employee
```

The effected database table named **employee** will be shown in a tabular format as follows:

Eid	Ename	Salary	Deg
1201	Gopal	46000	Technical Manager

tutorialspoint
SIMPLYEASYLEARNING

The salary of employee 1201 is updated to 46000.

Find Employee

To find the records of an employee, we will have to retrieve the existing data from the database and display it. In this operation, EntityTransaction is not applied while retrieving a record.

The class named **FindEmployee.java** is as follows.

```
package com.tutorialspoint.eclipselink.service;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import com.tutorialspoint.eclipselink.entity.Employee;

public class FindEmployee
{
      public static void main( String[ ] args )
      {
            EntityManagerFactory emfactory = Persistence
                        .createEntityManagerFactory( "Eclipselink_JPA" );
            EntityManager entitymanager = emfactory.
                        createEntityManager();
            Employee employee = entitymanager.
                        find( Employee.class, 1201 );

            System.out.println("employee ID = "+employee.getEid( ));
            System.out.println("employee NAME = "+employee.getEname( ));
            System.out.println("employee SALARY = "+employee.getSalary( ));
            System.out.println("employee DESIGNATION = "+employee.getDeg( ));
      }
}
```


tutorialspoint
SIMPLYEASYLEARNING

After compiling and executing the above program, you will get the following output from the Eclipselink library on the console panel of eclipse IDE.

```
employee ID = 1201
employee NAME = Gopal
employee SALARY = 46000.0
employee DESIGNATION = Technical Manager
```

Deleting Employee

To delete the records of an employee, first we will find the existing records and then delete it. Here EntityTransaction plays an important role.

The class named **DeleteEmployee.java** is as follows:

```
package com.tutorialspoint.eclipselink.service;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import com.tutorialspoint.eclipselink.entity.Employee;

public class DeleteEmployee
{
      public static void main( String[ ] args )
      {
            EntityManagerFactory emfactory = Persistence.
                        createEntityManagerFactory( "Eclipselink_JPA" );
            EntityManager entitymanager = emfactory.
                        createEntityManager( );
            entitymanager.getTransaction( ).begin( );
            Employee employee=entitymanager.
                        find( Employee.class, 1201 );
            entitymanager.remove( employee );
            entitymanager.getTransaction( ).commit( );
            entitymanager.close( );
            emfactory.close( );
```


tutorialspoint
SIMPLYEASYLEARNING

```
    }
}
```

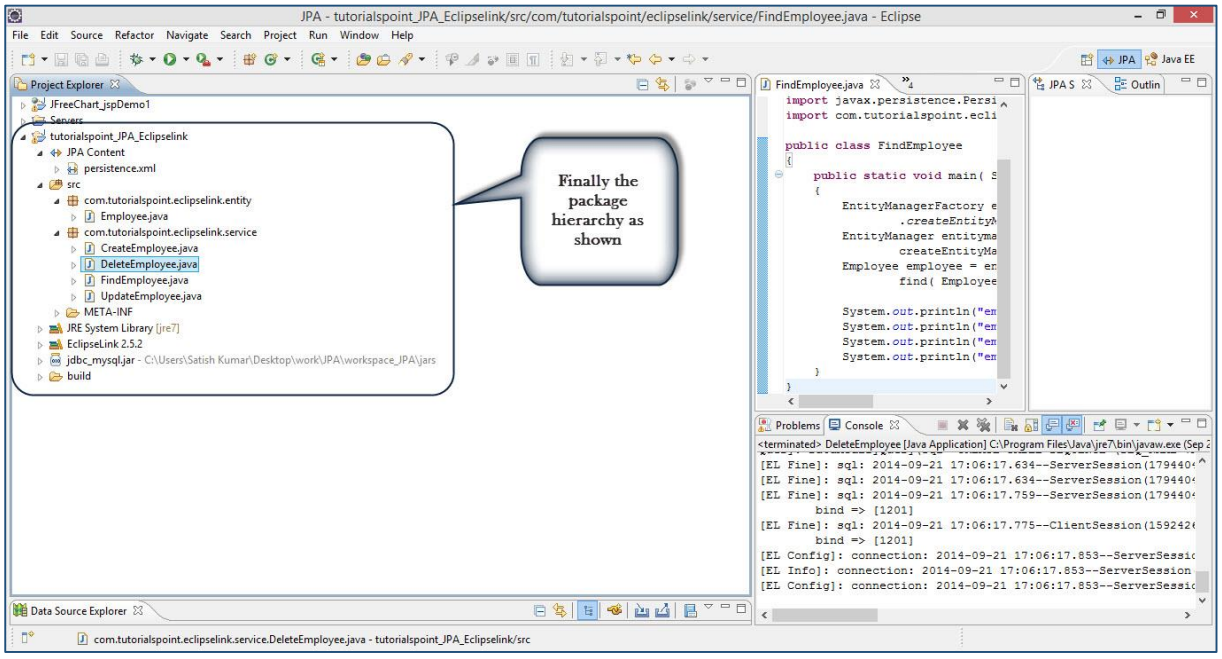
After compiling and executing the above program, you will get notifications from the Eclipselink library on the console panel of eclipse IDE.

For result, open the MySQL workbench and type the following queries.

```
use jpadb
select * from employee
```

The effected database named **employee** will have null records.

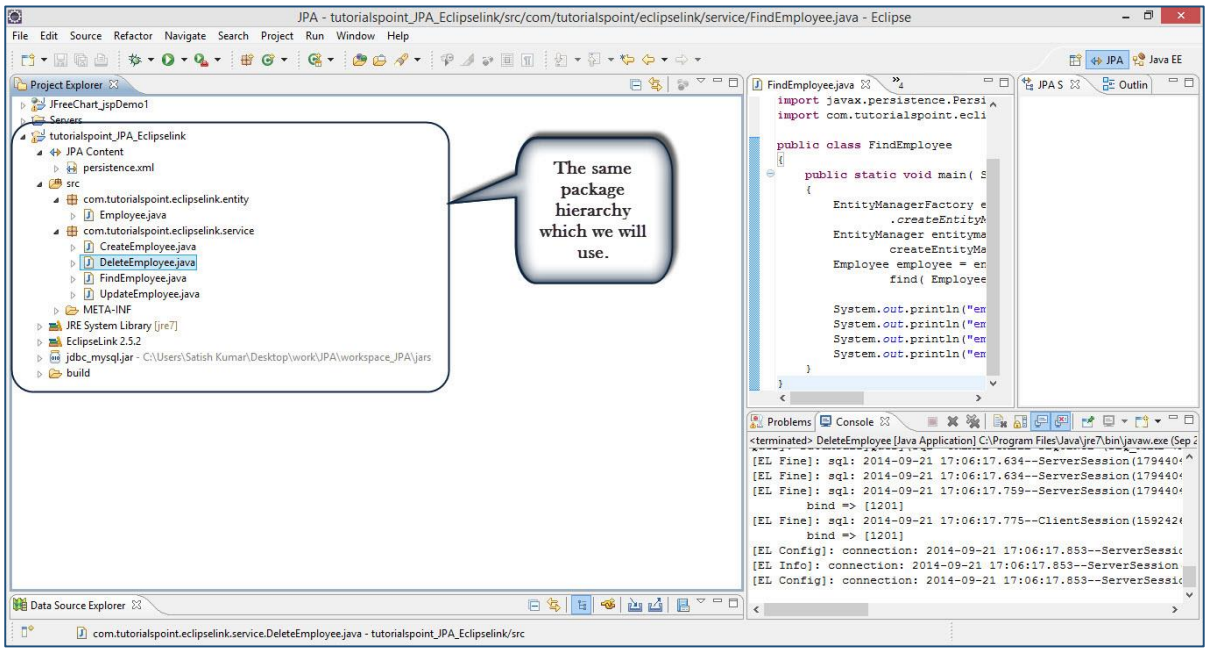
After completion of all the modules in this example, the package and file hierarchy looks as follows:



tutorialspoint
SIMPLYEASYLEARNING

6. JPA – JPQL

This chapter describes about JPQL and how it works with persistence units. In this chapter, the given examples follow the same package hierarchy, which we used in the previous chapter.



Java Persistence Query language

JPQL stands for Java Persistence Query Language. It is used to create queries against entities to store in a relational database. JPQL is developed based on SQL syntax. But it won't affect the database directly.

JPQL can retrieve data using SELECT clause, can do bulk updates using UPDATE clause and DELETE clause.

Query Structure

JPQL syntax is very similar to the syntax of SQL. Having SQL like syntax is an advantage because SQL is simple and being widely used. SQL works directly against relational database tables, records, and fields, whereas JPQL works with Java classes and instances.

For example, a JPQL query can retrieve an entity object rather than a field result set from a database, as with SQL. The JPQL query structure is as follows.

```
SELECT ... FROM ...
```


tutorialspoint
SIMPLYEASYLEARNING

```
[WHERE ...]
[GROUP BY ... [HAVING ...]]
[ORDER BY ...]
```

The structures of JPQL DELETE and UPDATE queries are as follows.

```
DELETE FROM ... [WHERE ...]

UPDATE ... SET ... [WHERE ...]
```

Scalar and Aggregate Functions

Scalar functions return resultant values based on input values. Aggregate functions return the resultant values by calculating the input values.

We will use the same example Employee Management as in the previous chapter. Here we will go through the service classes using scalar and aggregate functions of JPQL.

Let us assume the **jpadb.employee** table contains the following records.

Eid	Ename	Salary	Deg
1201	Gopal	40000	Technical Manager
1202	Manisha	40000	Proof reader
1203	Masthanvali	40000	Technical Writer
1204	Satish	30000	Technical writer
1205	Krishna	30000	Technical Writer
1206	Kiran	35000	Proof reader

Create a class named **ScalarandAggregateFunctions.java** under the **com.tutorialspoint.eclipselink.service** package as follows.

```
package com.tutorialspoint.eclipselink.service;

import java.util.List;
import javax.persistence.EntityManager;
```


tutorialspoint
SIMPLYEASYLEARNING

```
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import javax.persistence.Query;

public class ScalarandAggregateFunctions
{
      public static void main( String[ ] args )
      {
            EntityManagerFactory emfactory = Persistence.
                        createEntityManagerFactory( "Eclipselink_JPA" );
            EntityManager entitymanager = emfactory.
                        createEntityManager();
            //Scalar function
            Query query = entitymanager.
                  createQuery("Select UPPER(e.ename) from Employee e");
            List<String> list=query.getResultList();

            for(String e:list)
            {
                  System.out.println("Employee NAME :"+e);
            }
            //Aggregate function
            Query query1 = entitymanager.
                  createQuery("Select MAX(e.salary) from Employee e");
            Double result=(Double) query1.getSingleResult();
            System.out.println("Max Employee Salary :"+result);
      }
}
```

After compiling and executing the above program, you will get the following output on the console panel of Eclipse IDE.

```
Employee NAME :GOPAL
Employee NAME :MANISHA
Employee NAME :MASTHANVALI
```


tutorialspoint
SIMPLYEASYLEARNING

```
Employee NAME :SATISH
Employee NAME :KRISHNA
Employee NAME :KIRAN
Max Employee Salary :40000.0
```

Between, And, Like Keywords

Between, **And**, and **Like** are the main keywords of JPQL. These keywords are used after the **Where clause** in a query.

Create a class named **BetweenAndLikeFunctions.java** under the **com.tutorialspoint.eclipselink.service** package as follows:

```
package com.tutorialspoint.eclipselink.service;

import java.util.List;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import javax.persistence.Query;
import com.tutorialspoint.eclipselink.entity.Employee;

public class BetweenAndLikeFunctions
{
     public static void main( String[ ] args )
     {
            EntityManagerFactory emfactory = Persistence.
                        createEntityManagerFactory( "Eclipselink_JPA" );
            EntityManager entitymanager = emfactory.
                        createEntityManager();
            //Between
            Query query = entitymanager.
                        createQuery( "Select e " +
                                          "from Employee e " +
                                          "where e.salary " +
                                          "Between 30000 and 40000" );
```


tutorialspoint
SIMPLYEASYLEARNING

```
            List<Employee> list=(List<Employee>)query.getResultList( );


            for( Employee e:list )
            {
                  System.out.print("Employee ID :"+e.getEid( ));
                  System.out.println("\t Employee salar :"+e.getSalary( ));
            }

            //Like
            Query query1 = entitymanager.
                        createQuery("Select e " +
                                          "from Employee e " +
                                          "where e.ename LIKE 'M%'");
            List<Employee> list1=(List<Employee>)query1.getResultList( );
            for( Employee e:list1 )
            {
                  System.out.print("Employee ID :"+e.getEid( ));
                  System.out.println("\t Employee name :"+e.getEname( ));
            }
      }
}
```

After compiling and executing the above program, you will get the following output on the console panel of Eclipse IDE.

```
Employee ID :1201   Employee salary :40000.0
Employee ID :1202   Employee salary :40000.0
Employee ID :1203   Employee salary :40000.0
Employee ID :1204   Employee salary :30000.0
Employee ID :1205   Employee salary :30000.0
Employee ID :1206   Employee salary :35000.0
Employee ID :1202   Employee name :Manisha
Employee ID :1203   Employee name :Masthanvali
```


tutorialspoint
SIMPLYEASYLEARNING

Ordering

To order the records in JPQL, we use the **ORDER BY** clause. The usage of this clause is same as in SQL, but it deals with entities. The following example shows how to use the ORDER BY clause. Create a class **Ordering.java** under the **com.tutorialspoint.eclipselink.service** package as follows:

```
package com.tutorialspoint.eclipselink.service;

import java.util.List;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import javax.persistence.Query;
import com.tutorialspoint.eclipselink.entity.Employee;

public class Ordering
{
     public static void main( String[ ] args )
     {
            EntityManagerFactory emfactory = Persistence.
                        createEntityManagerFactory( "Eclipselink_JPA" );
            EntityManager entitymanager = emfactory.
                        createEntityManager();
            //Between
            Query query = entitymanager.
                        createQuery( "Select e " +
                                          "from Employee e " +
                                          "ORDER BY e.ename ASC" );
            List<Employee> list=(List<Employee>)query.getResultList( );

            for( Employee e:list )
            {
                  System.out.print("Employee ID :"+e.getEid( ));
                  System.out.println("\t Employee Name :"+e.getEname( ));
            }
     }
```


tutorialspoint
SIMPLYEASYLEARNING

```
}
```

Compiling and executing the above program will produce the following output on the console panel of Eclipse IDE.

```
Employee ID :1201   Employee Name :Gopal
Employee ID :1206   Employee Name :Kiran
Employee ID :1205   Employee Name :Krishna
Employee ID :1202   Employee Name :Manisha
Employee ID :1203   Employee Name :Masthanvali
Employee ID :1204   Employee Name :Satish
```

Named Queries

A @NamedQuery annotation is defined as a query with a predefined query string that is unchangeable. In contrast to dynamic queries, named queries may improve code organization by separating the JPQL query strings from POJO. It also passes the query parameters rather than embedding the literals dynamically into the query string and therefore produces more efficient queries.

First of all, add @NamedQuery annotation to the Employee entity class named **Employee.java** under the **com.tutorialspoint.eclipselink.entity** package as follows:

```
package com.tutorialspoint.eclipselink.entity;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.NamedQuery;
import javax.persistence.Table;

@Entity
@Table
@NamedQuery(
            query = "Select e from Employee e where e.eid = :id",
            name = "find employee by id")
```


tutorialspoint
SIMPLYEASYLEARNING

```
public class Employee
{
      @Id
      @GeneratedValue(strategy= GenerationType.AUTO)
      private int eid;
      private String ename;
      private double salary;
      private String deg;
      public Employee(int eid, String ename, double salary, String deg)
      {
            super( );
            this.eid = eid;
            this.ename = ename;
            this.salary = salary;
            this.deg = deg;
      }

      public Employee( )
      {
            super();
      }

      public int getEid( )
      {
            return eid;
      }
      public void setEid(int eid)
      {
            this.eid = eid;
      }

      public String getEname( )
      {
            return ename;
      }
```


tutorialspoint
SIMPLYEASYLEARNING

```
	public void setEname(String ename)
	{
		this.ename = ename;
	}

	public double getSalary( )
	{
		return salary;
	}
	public void setSalary(double salary)
	{
		this.salary = salary;
	}

	public String getDeg( )
	{
		return deg;
	}
	public void setDeg(String deg)
	{
		this.deg = deg;
	}
	@Override
	public String toString() {
		return "Employee [eid=" + eid + ", ename=" + ename + ", salary="
				+ salary + ", deg=" + deg + "]";
	}
}
```

Create a class named **NamedQueries.java** under the **com.tutorialspoint.eclipselink.service** package as follows:

```
package com.tutorialspoint.eclipselink.service;
import java.util.List;
import javax.persistence.EntityManager;
```


tutorialspoint
SIMPLYEASYLEARNING

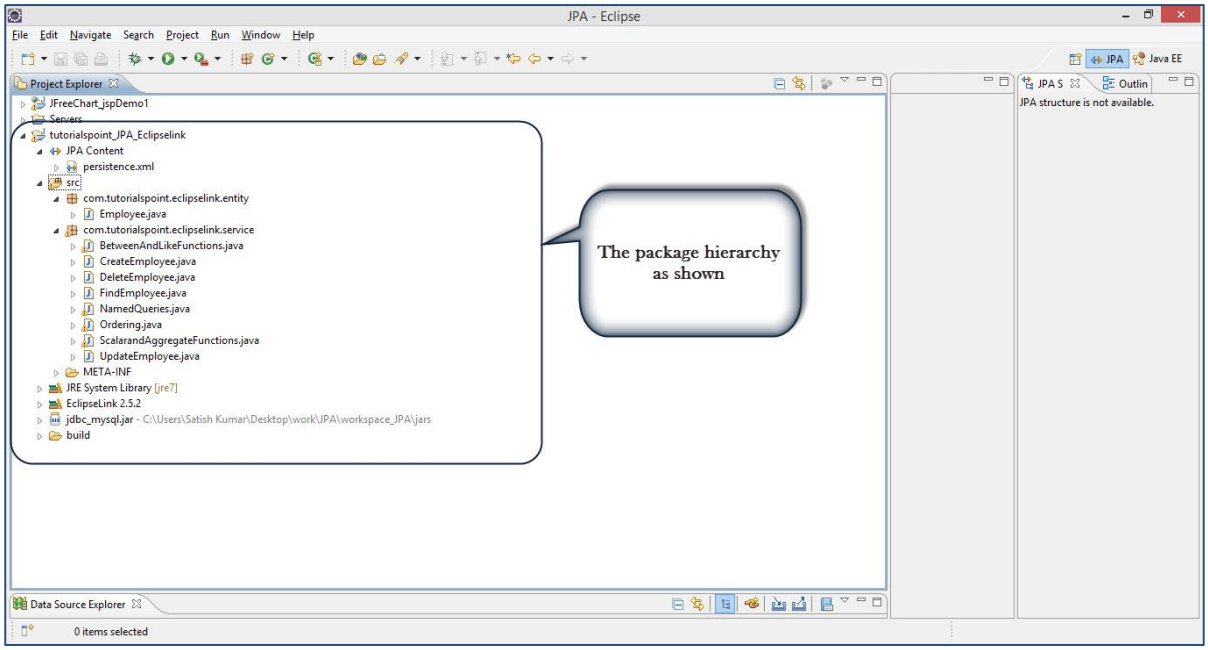
```
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import javax.persistence.Query;
import com.tutorialspoint.eclipselink.entity.Employee;

public class NamedQueries
{
      public static void main( String[ ] args )
      {
            EntityManagerFactory emfactory = Persistence.
                        createEntityManagerFactory( "Eclipselink_JPA" );
            EntityManager entitymanager = emfactory.
                        createEntityManager();
            Query query = entitymanager.createNamedQuery(
                        "find employee by id");
            query.setParameter("id", 1204);
            List<Employee> list = query.getResultList( );
            for( Employee e:list )
            {
                  System.out.print("Employee ID :"+e.getEid( ));
                  System.out.println("\t Employee Name :"+e.getEname( ));
            }
      }
}
```

After compiling and executing the above program, you will get the following output on the console panel of Eclipse IDE.

```
Employee ID :1204   Employee Name :Satish
```


tutorialspoint
SIMPLYEASYLEARNING

After adding all the above classes, the package hierarchy looks as follows.



Eager and Lazy Fetching

The most important concept of JPA is to make a duplicate copy of the database in the cache memory. While transacting with a database, the JPA first creates a duplicate set of data and only when it is committed using an entity manager, the changes are effected into the database.

There are two ways of fetching records from a database:

- Eager fetch**:** In eager fetching, related child objects are uploaded automatically while fetching a particular record.
- Lazy fetch**:** In lazy fetching, related objects are not uploaded automatically unless you specifically request for them. First of all, it checks the availability of related objects and notifies. Later, if you call any of the *getter* method of that entity, then it fetches all the records.

Lazy fetch is possible when you try to fetch the records for the first time. That way, a copy of the whole record is already stored in the cache memory. Performance-wise, lazy fetch is preferable.

tutorialspoint
SIMPLYEASYLEARNING

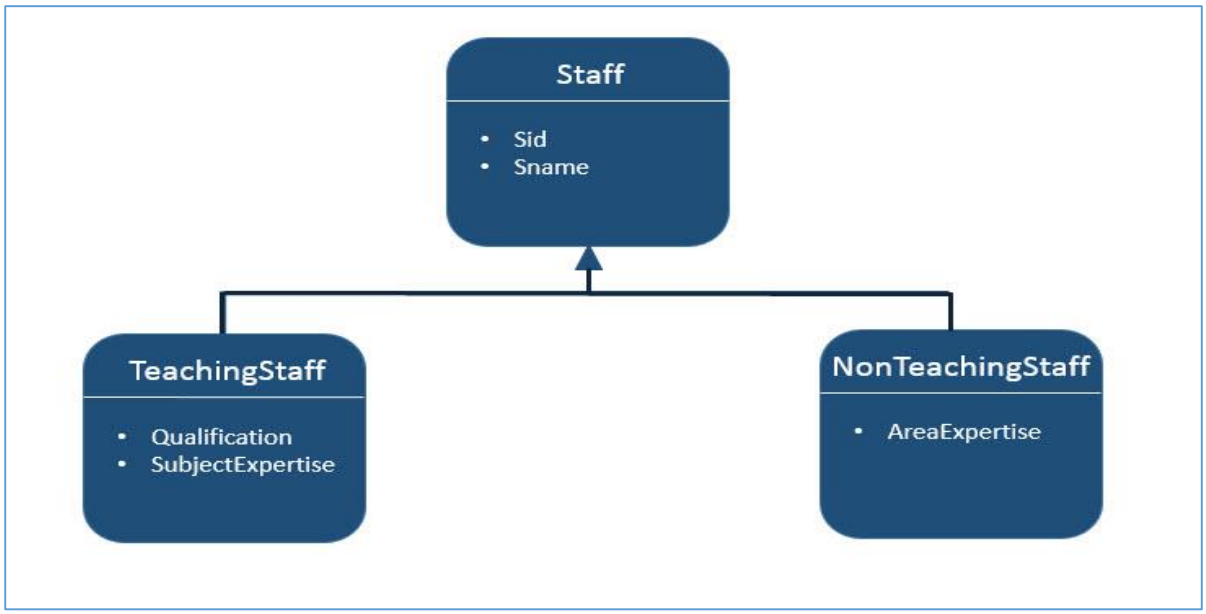
7. JPA – ADVANCED MAPPINGS

JPA is a library which is released with Java specifications. Therefore, it supports all the object-oriented concepts for entity persistence. Till now, we are done with the basics of object relational mapping. This chapter takes you through the advanced mappings between objects and relational entities.

Inheritance Strategies

Inheritance is the core concept of any object-oriented language, therefore we can use inheritance relationships or strategies between entities. JPA support three types of inheritance strategies: SINGLE_TABLE, JOINED_TABLE, and TABLE_PER_CONCRETE_CLASS.

Let us consider an example. The following diagram shows three classes, viz. Staff, TeachingStaff, and NonTeachingStaff, and their relationships.



In the above diagram, Staff is an entity, while TeachingStaff and NonTeachingStaff are the sub-entities of Staff. Here we will use the above example to demonstrate all three three strategies of inheritance.

Single Table strategy

Single-table strategy takes all classes fields (both super and sub classes) and map them down into a single table known as SINGLE_TABLE strategy. Here the discriminator value plays a key role in differentiating the values of three entities in one table.

tutorialspoint
SIMPLYEASYLEARNING

Let us consider the above example. TeachingStaff and NonTeachingStaff are the sub-classes of Staff. As per the concept of inheritance, a sub-class inherits the properties of its super-class. Therefore sid and sname are the fields that belong to both TeachingStaff and NonTeachingStaff. Create a JPA project. All the modules of this project are as follows:

Creating Entities

Create a package named **com.tutorialspoint.eclipselink.entity** under the **src** package. Create a new java class named **Staff.java** under the given package. The Staff entity class is shown as follows:

```
package com.tutorialspoint.eclipselink.entity;

import java.io.Serializable;
import javax.persistence.DiscriminatorColumn;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Inheritance;
import javax.persistence.InheritanceType;
import javax.persistence.Table;

@Entity
@Table
@Inheritance( strategy = InheritanceType.SINGLE_TABLE )
@DiscriminatorColumn( name="type" )
public class Staff implements Serializable
{
      @Id
      @GeneratedValue( strategy = GenerationType.AUTO )
      private int sid;
      private String sname;
      public Staff( int sid, String sname )
      {
            super( );
            this.sid = sid;
```


tutorialspoint
SIMPLYEASYLEARNING

```
            this.sname = sname;
      }

      public Staff( )
      {
            super( );
      }
      public int getSid( )
      {
            return sid;
      }
      public void setSid( int sid )
      {
            this.sid = sid;
      }
      public String getSname( )
      {
            return sname;
      }
      public void setSname( String sname )
      {
            this.sname = sname;
      }
}
```

In the above code, **@DescriminatorColumn** specifies the field name (**type**) and its values show the remaining (Teaching and NonTeachingStaff) fields.

Create a subclass (class) to Staff class named **TeachingStaff.java** under the **com.tutorialspoint.eclipselink.entity** package. The TeachingStaff Entity class is shown as follows:

```
package com.tutorialspoint.eclipselink.entity;


import javax.persistence.DiscriminatorValue;
import javax.persistence.Entity;
```


tutorialspoint
SIMPLYEASYLEARNING

```
@Entity
@DiscriminatorValue( value="TS" )
public class TeachingStaff extends Staff
{
      private String qualification;
      private String subjectexpertise;

      public TeachingStaff( int sid, String sname,
                  String qualification,String subjectexpertise )
      {
            super( sid, sname );
            this.qualification = qualification;
            this.subjectexpertise = subjectexpertise;
      }

      public TeachingStaff( )
      {
            super( );

      }

      public String getQualification( )
      {
            return qualification;
      }

      public void setQualification( String qualification )
      {
            this.qualification = qualification;
      }

      public String getSubjectexpertise( )
      {
            return subjectexpertise;
```


tutorialspoint
SIMPLYEASYLEARNING

```
	}

	public void setSubjectexpertise( String subjectexpertise )
	{
		this.subjectexpertise = subjectexpertise;
	}
}
```

Create a subclass (class) to Staff class named **NonTeachingStaff.java** under the **com.tutorialspoint.eclipselink.entity** package. The NonTeachingStaff Entity class is shown as follows:

```
package com.tutorialspoint.eclipselink.entity;

import javax.persistence.DiscriminatorValue;
import javax.persistence.Entity;

@Entity
@DiscriminatorValue( value = "NS" )
public class NonTeachingStaff extends Staff
{
	private String areaexpertise;

	public NonTeachingStaff( int sid, String sname,
			String areaexpertise )
	{
		super( sid, sname );
		this.areaexpertise = areaexpertise;
	}

	public NonTeachingStaff( )
	{
		super( );
	}
```


tutorialspoint
SIMPLYEASYLEARNING

```
	public String getAreaexpertise( )
	{
		return areaexpertise;
	}

	public void setAreaexpertise( String areaexpertise )
	{
		this.areaexpertise = areaexpertise;
	}
}
```

Persistence.xml

Persistence.xml contains the configuration information of the database and the registration information of entity classes. The xml file is shown as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0" 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">
	<persistence-unit name="Eclipselink_JPA"
			transaction-type="RESOURCE_LOCAL">
	<class>com.tutorialspoint.eclipselink.entity.Staff</class>

	<class>com.tutorialspoint.eclipselink.entity.NonTeachingStaff</class>
		<class>com.tutorialspoint.eclipselink.entity.TeachingStaff</class>
		<properties>
			<property name="javax.persistence.jdbc.url"
				value="jdbc:mysql://localhost:3306/jpadb"/>
			<property name="javax.persistence.jdbc.user" value="root"/>
			<property name="javax.persistence.jdbc.password"
				value="root"/>
			<property name="javax.persistence.jdbc.driver"
				value="com.mysql.jdbc.Driver"/>
			<property name="eclipselink.logging.level" value="FINE"/>
			<property name="eclipselink.ddl-generation"
				value="create-tables"/>
```


tutorialspoint
SIMPLYEASYLEARNING

```
                </properties>
    </persistence-unit>
</persistence>
```

Service class

Service classes are the implementation part of the business component. Create a package under the **src** package named **com.tutorialspoint.eclipselink.service**.

Create a class named **SaveClient.java** under the given package to store Staff, TeachingStaff, and NonTeachingStaff class fields. The SaveClient class is shown as follows:

```
package com.tutorialspoint.eclipselink.service;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import com.tutorialspoint.eclipselink.entity.NonTeachingStaff;
import com.tutorialspoint.eclipselink.entity.TeachingStaff;

public class SaveClient
{
    public static void main( String[ ] args )
    {
        EntityManagerFactory emfactory = Persistence.
                createEntityManagerFactory( "Eclipselink_JPA" );
        EntityManager entitymanager = emfactory.
                createEntityManager( );
        entitymanager.getTransaction( ).begin( );

        //Teaching staff entity
        TeachingStaff ts1=new TeachingStaff(
                1,"Gopal","MSc MEd","Maths");
        TeachingStaff ts2=new TeachingStaff(
                2, "Manisha", "BSc BEd", "English");
        //Non-Teaching Staff entity
```


tutorialspoint
SIMPLYEASYLEARNING

```
            NonTeachingStaff nts1=new NonTeachingStaff(
                        3, "Satish", "Accounts");
            NonTeachingStaff nts2=new NonTeachingStaff(
                        4, "Krishna", "Office Admin");

            //storing all entities
            entitymanager.persist(ts1);
            entitymanager.persist(ts2);
            entitymanager.persist(nts1);
            entitymanager.persist(nts2);

            entitymanager.getTransaction().commit();
            entitymanager.close();
            emfactory.close();
      }
}
```

After compiling and executing the above program, you will get notifications on the console panel of Eclipse IDE. Check MySQL workbench for output. The output is displayed in a tabular format as follows:

Sid	Type	Sname	Areaexpertise	Qualification	Subjectexpertise
1	TS	Gopal		MSC MED	Maths
2	TS	Manisha		BSC BED	English
3	NS	Satish	Accounts		
4	NS	Krishna	Office Admin		

Finally you will get a single table containing the field of all the three classes with a discriminator column named **Type** (field).

Joined Table Strategy

Joined table strategy is to share the referenced column that contains unique values to join the table and make easy transactions. Let us consider the same example as above.

Create a JPA Project. All the project modules are shown below.

tutorialspoint
SIMPLYEASYLEARNING

Creating Entities

Create a package named **com.tutorialspoint.eclipselink.entity** under the **src** package. Create a new java class named **Staff.java** under the given package. The Staff entity class is shown as follows:

```
package com.tutorialspoint.eclipselink.entity;

import java.io.Serializable;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Inheritance;
import javax.persistence.InheritanceType;
import javax.persistence.Table;

@Entity
@Table
@Inheritance( strategy = InheritanceType.JOINED )
public class Staff implements Serializable
{
     @Id
     @GeneratedValue( strategy = GenerationType.AUTO )
     private int sid;
     private String sname;
     public Staff( int sid, String sname )
     {
            super( );
            this.sid = sid;
            this.sname = sname;
     }
     public Staff( )
     {
            super( );
     }
```


tutorialspoint
SIMPLYEASYLEARNING

```
	public int getSid( )
	{
		return sid;
	}
	public void setSid( int sid )
	{
		this.sid = sid;
	}
	public String getSname( )
	{
		return sname;
	}
	public void setSname( String sname )
	{
		this.sname = sname;
	}
}
```

Create a subclass (class) to Staff class named **TeachingStaff.java** under the **com.tutorialspoint.eclipselink.entity** package. The TeachingStaff Entity class is shown as follows:

```
package com.tutorialspoint.eclipselink.entity;

import javax.persistence.DiscriminatorValue;
import javax.persistence.Entity;

@Entity
@PrimaryKeyJoinColumn(referencedColumnName="sid")
public class TeachingStaff extends Staff
{
	private String qualification;
	private String subjectexpertise;

	public TeachingStaff( int sid, String sname,
			String qualification,String subjectexpertise )
	{
```


tutorialspoint
SIMPLYEASYLEARNING

```
            super( sid, sname );
            this.qualification = qualification;
            this.subjectexpertise = subjectexpertise;
      }

      public TeachingStaff( )
      {
            super( );

      }

      public String getQualification( )
      {
            return qualification;
      }

      public void setQualification( String qualification )
      {
            this.qualification = qualification;
      }

      public String getSubjectexpertise( )
      {
            return subjectexpertise;
      }

      public void setSubjectexpertise( String subjectexpertise )
      {
            this.subjectexpertise = subjectexpertise;
      }
}
```

Create a subclass (class) to Staff class named **NonTeachingStaff.java** under the **com.tutorialspoint.eclipselink.entity** package. The NonTeachingStaff Entity class is shown as follows:

tutorialspoint
SIMPLYEASYLEARNING

```
package com.tutorialspoint.eclipselink.entity;

import javax.persistence.DiscriminatorValue;
import javax.persistence.Entity;

@Entity
@PrimaryKeyJoinColumn(referencedColumnName="sid")
public class NonTeachingStaff extends Staff
{
      private String areaexpertise;

      public NonTeachingStaff( int sid, String sname,
                  String areaexpertise )
      {
            super( sid, sname );
            this.areaexpertise = areaexpertise;
      }

      public NonTeachingStaff( )
      {
            super( );
      }

      public String getAreaexpertise( )
      {
            return areaexpertise;
      }

      public void setAreaexpertise( String areaexpertise )
      {
            this.areaexpertise = areaexpertise;
      }
}
```


tutorialspoint
SIMPLYEASYLEARNING

Persistence.xml

Persistence.xml file contains the configuration information of the database and the registration information of the entity classes. The xml file is shown as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0" 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">
     <persistence-unit name="Eclipselink_JPA"
                       transaction-type="RESOURCE_LOCAL">
     <class>com.tutorialspoint.eclipselink.entity.Staff</class>

     <class>com.tutorialspoint.eclipselink.entity.NonTeachingStaff</class>
           <class>com.tutorialspoint.eclipselink.entity.TeachingStaff</class>
           <properties>
                 <property name="javax.persistence.jdbc.url"
                           value="jdbc:mysql://localhost:3306/jpadb"/>
                 <property name="javax.persistence.jdbc.user" value="root"/>
                 <property name="javax.persistence.jdbc.password"
                           value="root"/>
                 <property name="javax.persistence.jdbc.driver"
                           value="com.mysql.jdbc.Driver"/>
                 <property name="eclipselink.logging.level" value="FINE"/>
                 <property name="eclipselink.ddl-generation"
                           value="create-tables"/>
                 </properties>
     </persistence-unit>
</persistence>
```

Service class

Service classes are the implementation part of the business component. Create a package under the **src** package named **com.tutorialspoint.eclipselink.service**.

tutorialspoint
SIMPLYEASYLEARNING

Create a class named **SaveClient.java** under the given package to store the fields of Staff, TeachingStaff, and NonTeachingStaff classes. Then SaveClient class is shown as follows:

```
package com.tutorialspoint.eclipselink.service;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import com.tutorialspoint.eclipselink.entity.NonTeachingStaff;
import com.tutorialspoint.eclipselink.entity.TeachingStaff;

public class SaveClient
{
      public static void main( String[ ] args )
      {
            EntityManagerFactory emfactory = Persistence.
                        createEntityManagerFactory( "Eclipselink_JPA" );
            EntityManager entitymanager = emfactory.
                        createEntityManager( );
            entitymanager.getTransaction( ).begin( );

            //Teaching staff entity
            TeachingStaff ts1=new TeachingStaff(
                        1,"Gopal","MSc MEd","Maths");
            TeachingStaff ts2=new TeachingStaff(
                        2, "Manisha", "BSc BEd", "English");
            //Non-Teaching Staff entity
            NonTeachingStaff nts1=new NonTeachingStaff(
                        3, "Satish", "Accounts");
            NonTeachingStaff nts2=new NonTeachingStaff(
                        4, "Krishna", "Office Admin");

            //storing all entities
            entitymanager.persist(ts1);
```


tutorialspoint
SIMPLYEASYLEARNING

```
            entitymanager.persist(ts2);
            entitymanager.persist(nts1);
            entitymanager.persist(nts2);

            entitymanager.getTransaction().commit();
            entitymanager.close();
            emfactory.close();
      }
}
```

After compiling and executing the above program, you will get notifications on the console panel of Eclipse IDE. For output, check MySQL workbench.

Here three tables are created and the result of **staff** table is displayed in a tabular format.

Sid	Dtype	Sname
1	TeachingStaff	Gopal
2	TeachingStaff	Manisha
3	NonTeachingStaff	Satish
4	NonTeachingStaff	krishna

The result of **TeachingStaff** table is displayed as follows.

Sid	Qualification	Subjectexpertise
1	MSC MED	Maths
2	BSC BED	English

In the above table, sid is the foreign key (reference field form staff table). The result of **NonTeachingStaff** table is displayed as follows.

Sid	Areaexpertise
3	Accounts
4	Office Admin

tutorialspoint
SIMPLYEASYLEARNING

Finally, the three tables are created using their respective fields and the SID field is shared by all the three tables. In the Staff table, SID is the primary key. In the remaining two tables (TeachingStaff and NonTeachingStaff), SID is the foreign key.

Table per Class Strategy

Table per class strategy is to create a table for each sub-entity. The Staff table will be created, but it will contain null values. The field values of Staff table must be shared by both TeachingStaff and NonTeachingStaff tables.

Let us consider the same example as above.

Creating Entities

Create a package named **com.tutorialspoint.eclipselink.entity** under the **src** package. Create a new java class named **Staff.java** under the given package. The Staff entity class is shown as follows:

```
package com.tutorialspoint.eclipselink.entity;

import java.io.Serializable;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Inheritance;
import javax.persistence.InheritanceType;
import javax.persistence.Table;

@Entity
@Table
@Inheritance( strategy = InheritanceType.TABLE_PER_CLASS )
public class Staff implements Serializable
{
     @Id
     @GeneratedValue( strategy = GenerationType.AUTO )
     private int sid;
     private String sname;
```


tutorialspoint
SIMPLYEASYLEARNING

```
	public Staff( int sid, String sname )
	{
		super( );
		this.sid = sid;
		this.sname = sname;
	}
	public Staff( )
	{
		super( );
	}
	public int getSid( )
	{
		return sid;
	}
	public void setSid( int sid )
	{
		this.sid = sid;
	}
	public String getSname( )
	{
		return sname;
	}
	public void setSname( String sname )
	{
		this.sname = sname;
	}
}
```

Create a subclass (class) to the Staff class named **TeachingStaff.java** under the **com.tutorialspoint.eclipselink.entity** package. The TeachingStaff Entity class is shown as follows:

```
package com.tutorialspoint.eclipselink.entity;


import javax.persistence.DiscriminatorValue;
```


tutorialspoint
SIMPLYEASYLEARNING

```
import javax.persistence.Entity;

@Entity
public class TeachingStaff extends Staff
{
      private String qualification;
      private String subjectexpertise;

      public TeachingStaff( int sid, String sname,
                  String qualification,String subjectexpertise )
      {
            super( sid, sname );
            this.qualification = qualification;
            this.subjectexpertise = subjectexpertise;
      }

      public TeachingStaff( )
      {
            super( );

      }

      public String getQualification( )
      {
            return qualification;
      }

      public void setQualification( String qualification )
      {
            this.qualification = qualification;
      }

      public String getSubjectexpertise( )
      {
            return subjectexpertise;
```


tutorialspoint
SIMPLYEASYLEARNING

```
	}

	public void setSubjectexpertise( String subjectexpertise )
	{
		this.subjectexpertise = subjectexpertise;
	}
}
```

Create a subclass (class) to the Staff class named **NonTeachingStaff.java** under the **com.tutorialspoint.eclipselink.entity** package. The NonTeachingStaff Entity class is shown as follows:

```
package com.tutorialspoint.eclipselink.entity;

import javax.persistence.DiscriminatorValue;
import javax.persistence.Entity;

@Entity
public class NonTeachingStaff extends Staff
{
	private String areaexpertise;

	public NonTeachingStaff( int sid, String sname,
			String areaexpertise )
	{
		super( sid, sname );
		this.areaexpertise = areaexpertise;
	}

	public NonTeachingStaff( )
	{
		super( );
	}

	public String getAreaexpertise( )
	{
		return areaexpertise;
```


tutorialspoint
SIMPLYEASYLEARNING

```
    }

    public void setAreaexpertise( String areaexpertise )
    {
        this.areaexpertise = areaexpertise;
    }
}
```

Persistence.xml

Persistence.xml file contains the configuration information of the database and the registration information of the entity classes. The xml file is shown as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0" 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">
    <persistence-unit name="Eclipselink_JPA"
                      transaction-type="RESOURCE_LOCAL">
    <class>com.tutorialspoint.eclipselink.entity.Staff</class>

    <class>com.tutorialspoint.eclipselink.entity.NonTeachingStaff</class>
        <class>com.tutorialspoint.eclipselink.entity.TeachingStaff</class>
        <properties>
            <property name="javax.persistence.jdbc.url"
                      value="jdbc:mysql://localhost:3306/jpadb"/>
            <property name="javax.persistence.jdbc.user" value="root"/>
            <property name="javax.persistence.jdbc.password"
                      value="root"/>
            <property name="javax.persistence.jdbc.driver"
                      value="com.mysql.jdbc.Driver"/>
            <property name="eclipselink.logging.level" value="FINE"/>
            <property name="eclipselink.ddl-generation"
                      value="create-tables"/>
            </properties>
    </persistence-unit>
```


tutorialspoint
SIMPLYEASYLEARNING

```
</persistence>
```

Service class

Service classes are the implementation part of business component. Create a package under the **src** package named **com.tutorialspoint.eclipselink.service**.

Create a class named **SaveClient.java** under the given package to store the class fields of Staff, TeachingStaff, and NonTeachingStaff. The SaveClient class is shown as follows:

```
package com.tutorialspoint.eclipselink.service;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import com.tutorialspoint.eclipselink.entity.NonTeachingStaff;
import com.tutorialspoint.eclipselink.entity.TeachingStaff;

public class SaveClient
{
      public static void main( String[ ] args )
      {
            EntityManagerFactory emfactory = Persistence.
                        createEntityManagerFactory( "Eclipselink_JPA" );
            EntityManager entitymanager = emfactory.
                        createEntityManager( );
            entitymanager.getTransaction( ).begin( );

            //Teaching staff entity
            TeachingStaff ts1=new TeachingStaff(
                        1,"Gopal","MSc MEd","Maths");
            TeachingStaff ts2=new TeachingStaff(
                        2, "Manisha", "BSc BEd", "English");
            //Non-Teaching Staff entity
            NonTeachingStaff nts1=new NonTeachingStaff(
                        3, "Satish", "Accounts");
```


tutorialspoint
SIMPLYEASYLEARNING

```
            NonTeachingStaff nts2=new NonTeachingStaff(
                        4, "Krishna", "Office Admin");


            //storing all entities
            entitymanager.persist(ts1);
            entitymanager.persist(ts2);
            entitymanager.persist(nts1);
            entitymanager.persist(nts2);

            entitymanager.getTransaction().commit();
            entitymanager.close();
            emfactory.close();
      }
}
```

After compiling and executing the above program, you will get notifications on the console panel of Eclipse IDE. For output, check MySQL workbench.

Here three tables are created and the **Staff** table contains null records.

The result of **TeachingStaff** is displayed as follows:

Sid	Qualification	Sname	Subjectexpertise
1	MSC MED	Gopal	Maths
2	BSC BED	Manisha	English

The above table TeachingStaff contains fields of both Staff and TeachingStaff Entities.

The result of **NonTeachingStaff** is displayed as follows:

Sid	Areaexpertise	Sname
3	Accounts	Satish
4	Office Admin	Krishna

tutorialspoint
SIMPLYEASYLEARNING

The above table NonTeachingStaff contains the fields of both Staff and NonTeachingStaff entities.

tutorialspoint
SIMPLYEASYLEARNING

8. JPA – ENTITY RELATIONSHIPS

This chapter takes you through the relationships between Entities. Generally the relations are more effective between tables in the database. Here the entity classes are treated as relational tables (concept of JPA), therefore the relationships between Entity classes are as follows:

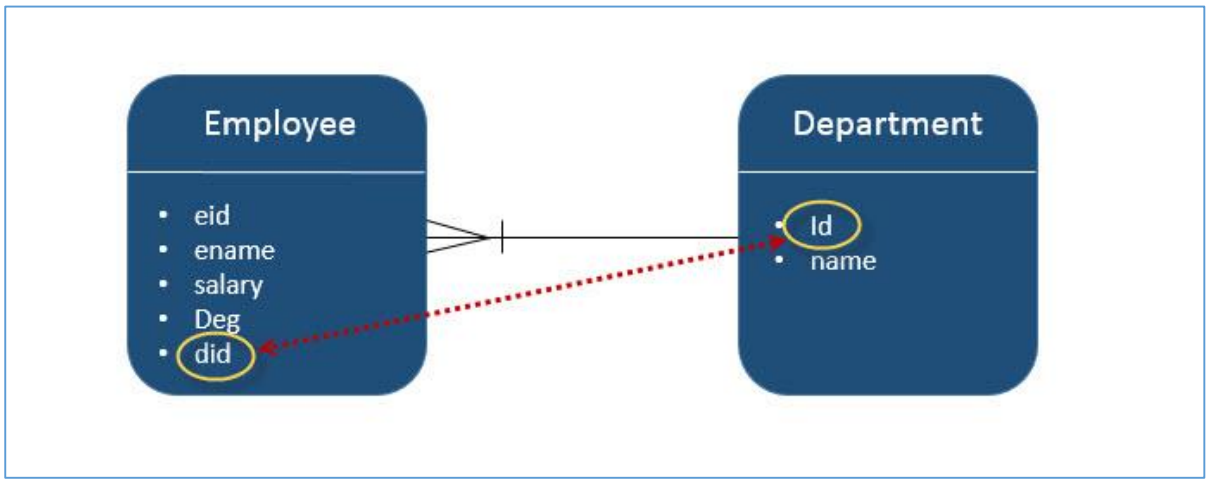
- @ManyToOne Relation
- @OneToMany Relation
- @OneToOne Relation
- @ManyToMany Relation

@ManyToOne Relation

Many-To-One relation between entities exists where one entity (column or set of columns) is referenced with another entity (column or set of columns) containing unique values. In relational databases, these relations are applied by using foreign key/primary key between the tables.

Let us consider an example of a relation between Employee and Department entities. In unidirectional manner, i.e., from Employee to Department, Many-To-One relation is applicable. That means each record of employee contains one department id, which should be a primary key in the Department table. Here in the Employee table, Department id is the foreign Key.

The following diagram shows the Many-To-One relation between the two tables.



Create a JPA project in eclipse IDE named **JPA_Eclipselink_MTO**. All the modules of this project are discussed below.

tutorialspoint
SIMPLYEASYLEARNING

Creating Entities

Follow the above diagram for creating entities. Create a package named **com.tutorialspoin.eclipselink.entity** under the **src** package. Create a class named **Department.java** under the given package. The class Department entity is shown as follows:

```
package com.tutorialspoint.eclipselink.entity;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
public class Department
{
    @Id
    @GeneratedValue( strategy=GenerationType.AUTO )
    private int id;
    private String name;

    public int getId()
    {
     return id;
    }

    public void setId(int id)
    {
     this.id = id;
    }

    public String getName( )
    {
     return name;
    }
```


tutorialspoint
SIMPLYEASYLEARNING

```
    public void setName( String deptName )
    {
     this.name = deptName;
    }
}
```

Create the second entity in this relation - Employee entity class named **Employee.java** under the **com.tutorialspoint.eclipselink.entity** package. The Employee entity class is shown as follows:

```
package com.tutorialspoint.eclipselink.entity;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.ManyToOne;

@Entity
public class Employee
{
     @Id
     @GeneratedValue( strategy= GenerationType.AUTO )
     private int eid;
     private String ename;
     private double salary;
     private String deg;
     @ManyToOne
     private Department department;

     public Employee(int eid,
                  String ename, double salary, String deg)
     {
            super( );
            this.eid = eid;
            this.ename = ename;
            this.salary = salary;
```


tutorialspoint
SIMPLYEASYLEARNING

```
        this.deg = deg;
    }

    public Employee( )
    {
        super();
    }

    public int getEid( )
    {
        return eid;
    }
    public void setEid(int eid)
    {
        this.eid = eid;
    }

    public String getEname( )
    {
        return ename;
    }
    public void setEname(String ename)
    {
        this.ename = ename;
    }

    public double getSalary( )
    {
        return salary;
    }
    public void setSalary(double salary)
    {
        this.salary = salary;
    }
```


tutorialspoint
SIMPLYEASYLEARNING

```
	public String getDeg( )
	{
		return deg;
	}
	public void setDeg(String deg)
	{
		this.deg = deg;
	}

	public Department getDepartment() {
		return department;
	}

	public void setDepartment(Department department) {
		this.department = department;
	}
}
```

Persistence.xml

Persistence.xml is required to configure the database and the registration of entity classes.

Persitence.xml will be created by the eclipse IDE while creating a JPA Project. The configuration details are user specifications. The persistence.xml is shown as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0"
		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">
	<persistence-unit name="Eclipselink_JPA"
			transaction-type="RESOURCE_LOCAL">
	<class>com.tutorialspoint.eclipselink.entity.Employee</class>
	<class>com.tutorialspoint.eclipselink.entity.Department</class>
		<properties>
```


tutorialspoint
SIMPLYEASYLEARNING

```
                <property name="javax.persistence.jdbc.url"
                          value="jdbc:mysql://localhost:3306/jpadb"/>
                <property name="javax.persistence.jdbc.user" value="root"/>
                <property name="javax.persistence.jdbc.password"
                          value="root"/>
                <property name="javax.persistence.jdbc.driver"
                          value="com.mysql.jdbc.Driver"/>
                <property name="eclipselink.logging.level" value="FINE"/>
                <property name="eclipselink.ddl-generation"
                          value="create-tables"/>
                </properties>
     </persistence-unit>
</persistence>
```

Service Classes

This module contains the service classes, which implements the relational part using the attribute initialization. Create a package under the **src** package named **com.tutorialspoint.eclipselink.service**. The DAO class named **ManyToOne.java** is created under the given package. The DAO class is shown as follows:

```
package com.tutorialspointeclipselink.service;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import com.tutorialspoint.eclipselink.entity.Department;
import com.tutorialspoint.eclipselink.entity.Employee;

public class ManyToOne
{
     public static void main( String[ ] args )
     {
          EntityManagerFactory emfactory = Persistence.
                         createEntityManagerFactory( "Eclipselink_JPA" );
          EntityManager entitymanager = emfactory.
```


tutorialspoint
SIMPLYEASYLEARNING

```
                createEntityManager( );
        entitymanager.getTransaction( ).begin( );

        //Create Department Entity
        Department department = new Department();
        department.setName("Development");
        //Store Department
        entitymanager.persist(department);

        //Create Employee1 Entity
        Employee employee1 = new Employee();
        employee1.setEname("Satish");
        employee1.setSalary(45000.0);
        employee1.setDeg("Technical Writer");
        employee1.setDepartment(department);

        //Create Employee2 Entity
        Employee employee2 = new Employee();
        employee2.setEname("Krishna");
        employee2.setSalary(45000.0);
        employee2.setDeg("Technical Writer");
        employee2.setDepartment(department);

        //Create Employee3 Entity
        Employee employee3 = new Employee();
        employee3.setEname("Masthanvali");
        employee3.setSalary(50000.0);
        employee3.setDeg("Technical Writer");
        employee3.setDepartment(department);

        //Store Employees
        entitymanager.persist(employee1);
        entitymanager.persist(employee2);
        entitymanager.persist(employee3);
```


tutorialspoint
SIMPLYEASYLEARNING

```
            entitymanager.getTransaction().commit();
            entitymanager.close();
            emfactory.close();
      }
}
```

After compiling and executing the above program, you will get notifications on the console panel of Eclipse IDE. For output, check MySQL workbench. In this example, two tables are created.

Pass the following query in MySQL interface and the result of **Department** table will be displayed as follows.

```
Select * from department
```

Id	Name
101	Development

Pass the following query in MySQL interface and the result of **Employee** table will be displayed as follows.

```
Select * from employee
```

Eid	Deg	Ename	Salary	Department_Id
102	Technical Writer	Satish	45000	101
103	Technical Writer	Krishna	45000	101
104	Technical Writer	Masthan Wali	50000	101

In the above table Deparment_Id is the foreign key (reference field) from the Department table.

@OneToMany Relation

In this relationship, each row of one entity is referenced to many child records in other entity. The important thing is that child records cannot have multiple parents. In a one-to-many relationship between Table A and Table B, each row in Table A can be linked to one or multiple rows in Table B.

tutorialspoint
SIMPLYEASYLEARNING

Let us consider the above example. Suppose **Employee** and **Department** tables in the above example are connected in a reverse unidirectional manner, then the relation becomes One-To-Many relation. Create a JPA project in eclipse IDE named **JPA_Eclipselink_OTM**. All the modules of this project are discussed below.

Creating Entities

Follow the above given diagram for creating entities. Create a package named **com.tutorialspoin.eclipselink.entity** under the **src** package. Create a class named **Department.java** under the given package. The class Department entity is shown as follows:

```
package com.tutorialspoint.eclipselink.entity;

import java.util.List;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.OneToMany;

@Entity
public class Department
{
    @Id
    @GeneratedValue( strategy=GenerationType.AUTO )
    private int id;
    private String name;

    @OneToMany( targetEntity=Employee.class )
    private List employeelist;

    public int getId()
    {
     return id;
    }

    public void setId(int id)
    {
```


tutorialspoint
SIMPLYEASYLEARNING

```
      this.id = id;
    }

    public String getName( )
    {
      return name;
    }

    public void setName( String deptName )
    {
      this.name = deptName;
    }

    public List getEmployeelist()
    {
      return employeelist;
    }

    public void setEmployeelist(List employeelist)
   {
     this.employeelist = employeelist;
   }
}
```

Create the second entity in this relation – Employee entity class – named **Employee.java** under the **com.tutorialspoint.eclipselink.entity** package. The Employee entity class is shown as follows:

```
package com.tutorialspoint.eclipselink.entity;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
```


tutorialspoint
SIMPLYEASYLEARNING

```
@Entity
public class Employee
{
      @Id
      @GeneratedValue( strategy= GenerationType.AUTO )
      private int eid;
      private String ename;
      private double salary;
      private String deg;

      public Employee(int eid,
                  String ename, double salary, String deg)
      {
            super( );
            this.eid = eid;
            this.ename = ename;
            this.salary = salary;
            this.deg = deg;
      }

      public Employee( )
      {
            super();
      }

      public int getEid( )
      {
            return eid;
      }
      public void setEid(int eid)
      {
            this.eid = eid;
      }
```


tutorialspoint
SIMPLYEASYLEARNING

```
    public String getEname( )
    {
        return ename;
    }
    public void setEname(String ename)
    {
        this.ename = ename;
    }

    public double getSalary( )
    {
        return salary;
    }
    public void setSalary(double salary)
    {
        this.salary = salary;
    }

    public String getDeg( )
    {
        return deg;
    }
    public void setDeg(String deg)
    {
        this.deg = deg;
    }
}
```

Persistence.xml

The persistence.xml file is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0"
             xmlns="http://java.sun.com/xml/ns/persistence"
```


tutorialspoint
SIMPLYEASYLEARNING

```
            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">
     <persistence-unit name="Eclipselink_JPA"
                       transaction-type="RESOURCE_LOCAL">
     <class>com.tutorialspoint.eclipselink.entity.Employee</class>
     <class>com.tutorialspoint.eclipselink.entity.Department</class>
           <properties>
                 <property name="javax.persistence.jdbc.url"
                           value="jdbc:mysql://localhost:3306/jpadb"/>
                 <property name="javax.persistence.jdbc.user" value="root"/>
                 <property name="javax.persistence.jdbc.password"
                           value="root"/>
                 <property name="javax.persistence.jdbc.driver"
                           value="com.mysql.jdbc.Driver"/>
                 <property name="eclipselink.logging.level" value="FINE"/>
                 <property name="eclipselink.ddl-generation"
                           value="create-tables"/>
                 </properties>
     </persistence-unit>
</persistence>
```

Service Classes

This module contains the service classes that implement the relational part using the attribute initialization. Create a package under the **src** package named **com.tutorialspoint.eclipselink.service**. The DAO class named **OneToMany.java** is created under the given package. The DAO class is as follows:

```
package com.tutorialspointeclipselink.service;


import java.util.List;
import java.util.ArrayList;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
```


tutorialspoint
SIMPLYEASYLEARNING

```
import com.tutorialspoint.eclipselink.entity.Department;
import com.tutorialspoint.eclipselink.entity.Employee;

public class OneToMany
{
      public static void main(String[] args)
      {
            EntityManagerFactory emfactory = Persistence.
                        createEntityManagerFactory( "Eclipselink_JPA" );
            EntityManager entitymanager = emfactory.
                        createEntityManager( );
            entitymanager.getTransaction( ).begin( );

            //Create Employee1 Entity
            Employee employee1 = new Employee();
            employee1.setEname("Satish");
            employee1.setSalary(45000.0);
            employee1.setDeg("Technical Writer");

            //Create Employee2 Entity
            Employee employee2 = new Employee();
            employee2.setEname("Krishna");
            employee2.setSalary(45000.0);
            employee2.setDeg("Technical Writer");

            //Create Employee3 Entity
            Employee employee3 = new Employee();
            employee3.setEname("Masthanvali");
            employee3.setSalary(50000.0);
            employee3.setDeg("Technical Writer");

            //Store Employee
            entitymanager.persist(employee1);
            entitymanager.persist(employee2);
            entitymanager.persist(employee3);
```


tutorialspoint
SIMPLYEASYLEARNING

```
            //Create Employeelist
            List<Employee> emplist = new ArrayList();
            emplist.add(employee1);
            emplist.add(employee2);
            emplist.add(employee3);

            //Create Department Entity
            Department department= new Department();
            department.setName("Development");
            department.setEmployeelist(emplist);

            //Store Department
            entitymanager.persist(department);
            entitymanager.getTransaction().commit();
            entitymanager.close();
            emfactory.close();
      }
}
```

In this project three tables are created. Pass the following query in MySQL interface and the result of **department_employee** table will be displayed as follows:

```
Select * from department_Id
```

Department_Id	Employee_Eid
254	251
254	252
254	253

tutorialspoint
SIMPLYEASYLEARNING

In the above table, **deparment_id** and **employee_id** are the foreign keys (reference fields) from **department** and **employee** tables.

Pass the following query in MySQL interface and the result of **department** table will be displayed.

```
Select * from department
```

Id	Name
254	Development

Pass the following query in MySQL interface and the result of **employee** table will be displayed as follows:

```
Select * from employee
```

Eid	Deg	Ename	Salary
251	Technical Writer	Satish	45000
252	Technical Writer	Krishna	45000
253	Technical Writer	Masthanvali	50000

@OneToOne Relation

In One-To-One relationship, one item can be linked to only one other item. It means each row of one entity is referred to one and only one row of another entity.

Let us consider the above example. **Employee** and **Department** in a reverse unidirectional manner, the relation is One-To-One relation. It means each employee belongs to only one department. Create a JPA project in eclipse IDE named **JPA_Eclipselink_OTO**. All the modules of this project are discussed below.

Creating Entities

Follow the above given diagram for creating entities. Create a package named **com.tutorialspoin.eclipselink.entity** under the **src** package. Create a class named **Department.java** under the given package. The class Department entity is as follows:

tutorialspoint
SIMPLYEASYLEARNING

```
package com.tutorialspoint.eclipselink.entity;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
public class Department
{
    @Id
    @GeneratedValue( strategy=GenerationType.AUTO )
    private int id;
    private String name;

    public int getId()
    {
     return id;
    }

    public void setId(int id)
    {
     this.id = id;
    }

    public String getName( )
    {
     return name;
    }

    public void setName( String deptName )
    {
     this.name = deptName;
    }
}
```


tutorialspoint
SIMPLYEASYLEARNING

Create the second entity in this relation – Employee entity class – named **Employee.java** under the **com.tutorialspoint.eclipselink.entity** package. The Employee entity class is as follows:

```
package com.tutorialspoint.eclipselink.entity;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.OneToOne;

@Entity
public class Employee
{
      @Id
      @GeneratedValue( strategy= GenerationType.AUTO )
      private int eid;
      private String ename;
      private double salary;
      private String deg;

      @OneToOne
      private Department department;

      public Employee(int eid,
                  String ename, double salary, String deg)
      {
            super( );
            this.eid = eid;
            this.ename = ename;
            this.salary = salary;
            this.deg = deg;
```


tutorialspoint
SIMPLYEASYLEARNING

```
}

public Employee( )
{
	super();
}

public int getEid( )
{
	return eid;
}
public void setEid(int eid)
{
	this.eid = eid;
}

public String getEname( )
{
	return ename;
}
public void setEname(String ename)
{
	this.ename = ename;
}

public double getSalary( )
{
	return salary;
}
public void setSalary(double salary)
{
	this.salary = salary;
}

public String getDeg( )
```


tutorialspoint
SIMPLYEASYLEARNING

```
	{
		return deg;
	}
	public void setDeg(String deg)
	{
		this.deg = deg;
	}

	public Department getDepartment()
	{
		return department;
	}

	public void setDepartment(Department department)
	{
		this.department = department;
	}
}
```

Persistence.xml

The persistence.xml file is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0"
             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">
     <persistence-unit name="Eclipselink_JPA"
                       transaction-type="RESOURCE_LOCAL">
     <class>com.tutorialspoint.eclipselink.entity.Employee</class>
     <class>com.tutorialspoint.eclipselink.entity.Department</class>
            <properties>
                  <property name="javax.persistence.jdbc.url"
                            value="jdbc:mysql://localhost:3306/jpadb"/>
```


tutorialspoint
SIMPLYEASYLEARNING

```
                <property name="javax.persistence.jdbc.user" value="root"/>
                <property name="javax.persistence.jdbc.password"
                          value="root"/>
                <property name="javax.persistence.jdbc.driver"
                          value="com.mysql.jdbc.Driver"/>
                <property name="eclipselink.logging.level" value="FINE"/>
                <property name="eclipselink.ddl-generation"
                          value="create-tables"/>
                </properties>
     </persistence-unit>
</persistence>
```

Service Classes

Create a package under the **src** package named **com.tutorialspoint.eclipselink.service**. The DAO class named **OneToOne.java** is created under the given package. The DAO class is sas follows:

```
package com.tutorialspointeclipselink.service;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import com.tutorialspoint.eclipselink.entity.Department;
import com.tutorialspoint.eclipselink.entity.Employee;

public class OneToOne
{
     public static void main(String[] args)
     {
            EntityManagerFactory emfactory = Persistence.
                        createEntityManagerFactory( "Eclipselink_JPA" );
            EntityManager entitymanager = emfactory.
                        createEntityManager( );
            entitymanager.getTransaction( ).begin( );
```


tutorialspoint
SIMPLYEASYLEARNING

```
            //Create Department Entity
            Department department = new Department();
            department.setName("Development");

            //Store Department
            entitymanager.persist(department);

            //Create Employee Entity
            Employee employee = new Employee();
            employee.setEname("Satish");
            employee.setSalary(45000.0);
            employee.setDeg("Technical Writer");
            employee.setDepartment(department);

            //Store Employee
            entitymanager.persist(employee);

            entitymanager.getTransaction().commit();
            entitymanager.close();
            emfactory.close();
      }
}
```

In the above example, two tables are created. Pass the following query in MySQL interface and the result of **department** table will be displayed as follows:

```
Select * from department
```

Id	Name
301	Development

Pass the following query in MySQL interface and the result of **employee** table will be displayed as follows:

tutorialspoint
SIMPLYEASYLEARNING

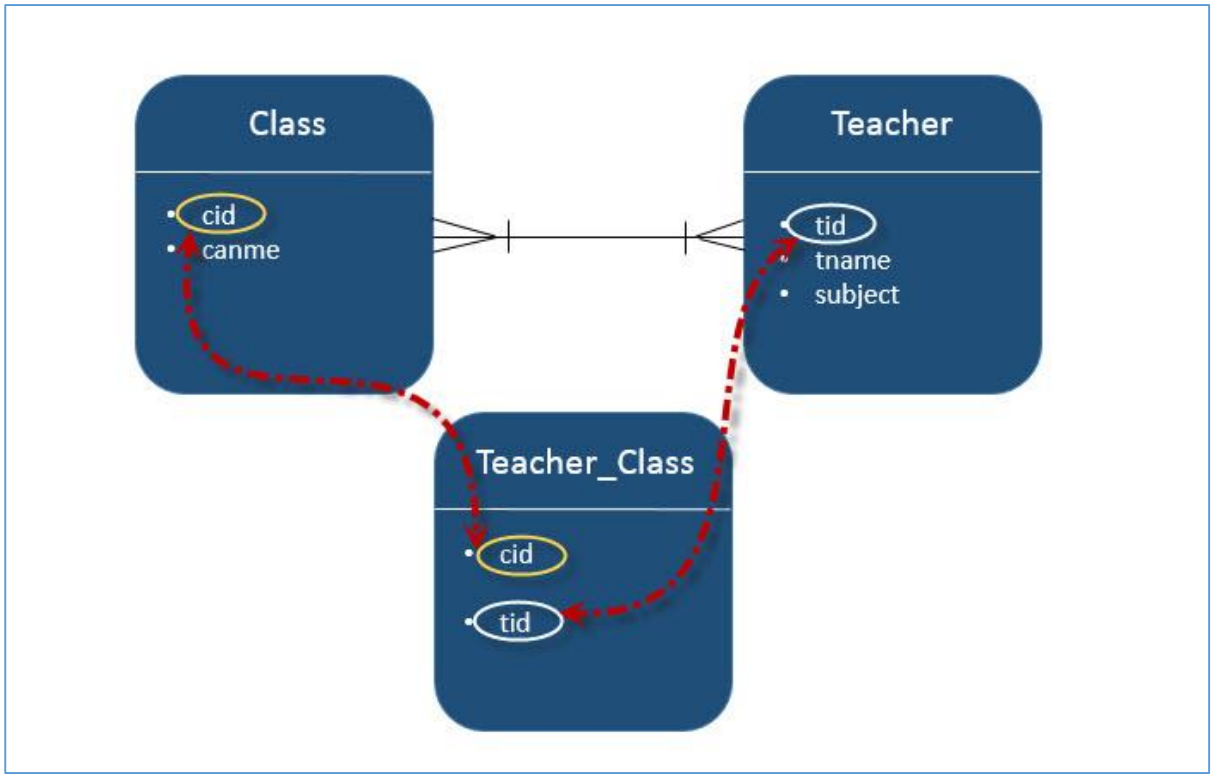
```
Select * from employee
```

Eid	Deg	Ename	Salary	Department_id
302	Technical Writer	Satish	45000	301

@ManyToMany Relation

Many-To-Many relationship is where one or more rows from one entity are associated with more than one rows in another entity.

Let us consider an example of a relation between two entities: **Class** and **Teacher**. In bidirectional manner, both Class and Teacher have Many-To-One relation. That means each record of Class is referred by Teacher set (teacher ids), which should be primary keys in the Teacher table and stored in the Teacher_Class table and vice versa. Here, the Teachers_Class table contains both the foreign key fields. Create a JPA project in eclipse IDE named **JPA_Eclipselink_MTM**. All the modules of this project are discussed below.



tutorialspoint
SIMPLYEASYLEARNING

Creating Entities

Create entities by following the schema shown in the diagram above. Create a package named **com.tutorialspoin.eclipselink.entity** under the **src** package. Create a class named **Clas.java** under the given package. The class Department entity is as follows:

```
package com.tutorialspoint.eclipselink.entity;

import java.util.Set;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.ManyToMany;

@Entity
public class Clas
{
     @Id
     @GeneratedValue( strategy = GenerationType.AUTO )
     private int cid;
     private String cname;

     @ManyToMany(targetEntity=Teacher.class)
     private Set teacherSet;

     public Clas()
     {
           super();
     }
     public Clas(int cid,
                 String cname, Set teacherSet)
     {
           super();
           this.cid = cid;
           this.cname = cname;
           this.teacherSet = teacherSet;
```


tutorialspoint
SIMPLYEASYLEARNING

```
    }
    public int getCid()
    {
        return cid;
    }
    public void setCid(int cid)
    {
        this.cid = cid;
    }
    public String getCname()
    {
        return cname;
    }
    public void setCname(String cname)
    {
        this.cname = cname;
    }
    public Set getTeacherSet()
    {
        return teacherSet;
    }
    public void setTeacherSet(Set teacherSet)
    {
        this.teacherSet = teacherSet;
    }
}
```

Create the second entity in this relation – Employee entity class – named **Teacher.java** under the **com.tutorialspoint.eclipselink.entity** package. The Employee entity class is as follows:

```
package com.tutorialspoint.eclipselink.entity;

import java.util.Set;
import javax.persistence.Entity;
```


tutorialspoint
SIMPLYEASYLEARNING

```
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.ManyToMany;

@Entity
public class Teacher
{
	@Id
	@GeneratedValue( strategy = GenerationType.AUTO )
	private int tid;
	private String tname;
	private String subject;

	@ManyToMany(targetEntity=Clas.class)
	private Set clasSet;

	public Teacher()
	{
		super();
	}
	public Teacher(int tid, String tname, String subject,
			Set clasSet)
	{
		super();
		this.tid = tid;
		this.tname = tname;
		this.subject = subject;
		this.clasSet = clasSet;
	}
	public int getTid()
	{
		return tid;
	}
	public void setTid(int tid)
```


tutorialspoint
SIMPLYEASYLEARNING

```
	{
		this.tid = tid;
	}
	public String getTname()
	{
		return tname;
	}
	public void setTname(String tname)
	{
		this.tname = tname;
	}
	public String getSubject()
	{
		return subject;
	}
	public void setSubject(String subject)
	{
		this.subject = subject;
	}
	public Set getClasSet()
	{
		return clasSet;
	}
	public void setClasSet(Set clasSet)
	{
		this.clasSet = clasSet;
	}
}
```

Persistence.xml

The persistence.xml file is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0"
             xmlns="http://java.sun.com/xml/ns/persistence"
```


tutorialspoint
SIMPLYEASYLEARNING

```
            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">
     <persistence-unit name="Eclipselink_JPA"
                       transaction-type="RESOURCE_LOCAL">
     <class>com.tutorialspoint.eclipselink.entity.Employee</class>
     <class>com.tutorialspoint.eclipselink.entity.Department</class>
           <properties>
                 <property name="javax.persistence.jdbc.url"
                           value="jdbc:mysql://localhost:3306/jpadb"/>
                 <property name="javax.persistence.jdbc.user" value="root"/>
                 <property name="javax.persistence.jdbc.password"
                           value="root"/>
                 <property name="javax.persistence.jdbc.driver"
                           value="com.mysql.jdbc.Driver"/>
                 <property name="eclipselink.logging.level" value="FINE"/>
                 <property name="eclipselink.ddl-generation"
                           value="create-tables"/>
                 </properties>
     </persistence-unit>
</persistence>
```

Service Classes

Create a package under the **src** package named **com.tutorialspoint.eclipselink.service**. The DAO class named **ManyToMany.java** is created under the given package. The DAO class is as follows:

```
package com.tutorialspoint.eclipselink.service;

import java.util.HashSet;
import java.util.Set;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
```


tutorialspoint
SIMPLYEASYLEARNING

```
import com.tutorialspoint.eclipselink.entity.Clas;
import com.tutorialspoint.eclipselink.entity.Teacher;

public class ManyToMany
{
      public static void main(String[] args)
      {
            EntityManagerFactory emfactory = Persistence.
                        createEntityManagerFactory( "Eclipselink_JPA" );
            EntityManager entitymanager = emfactory.
                        createEntityManager( );
            entitymanager.getTransaction( ).begin( );

            //Create Clas Entity
            Clas clas1=new Clas(0,"1st",null);
            Clas clas2=new Clas(0,"2nd",null);
            Clas clas3=new Clas(0,"3rd",null);

            //Store Clas
            entitymanager.persist(clas1);
            entitymanager.persist(clas2);
            entitymanager.persist(clas3);

            //Create Clas Set1
            Set<Clas> classSet1 = new HashSet();
            classSet1.add(clas1);
            classSet1.add(clas2);
            classSet1.add(clas3);

            //Create Clas Set2
            Set<Clas> classSet2 = new HashSet();
            classSet2.add(clas3);
            classSet2.add(clas1);
            classSet2.add(clas2);
```


tutorialspoint
SIMPLYEASYLEARNING

```
            //Create Clas Set3
            Set<Clas> classSet3 = new HashSet();
            classSet3.add(clas2);
            classSet3.add(clas3);
            classSet3.add(clas1);

            //Create Teacher Entity
            Teacher teacher1 = new Teacher(0,
                        "Satish","Java",classSet1);
            Teacher teacher2 = new Teacher(0,
                        "Krishna","Adv Java",classSet2);
            Teacher teacher3 = new Teacher(0,
                        "Masthanvali","DB2",classSet3);

            //Store Teacher
            entitymanager.persist(teacher1);
            entitymanager.persist(teacher2);
            entitymanager.persist(teacher3);



            entitymanager.getTransaction( ).commit( );
            entitymanager.close( );
            emfactory.close( );
      }
}
```

In this example project, three tables are created. Pass the following query in MySQL interface and the result of **teacher_clas** table will be displayed as follows:

```
Select * form teacher_clas
```

Teacher _tid	Classet_cid
354	351
355	351

tutorialspoint
SIMPLYEASYLEARNING

356	351
354	352
355	352
356	352
354	353
355	353
356	353

In the above table, **teacher_tid** is the foreign key from the table **teacher**, and **classet_cid** is the foreign key from the table **class**. Therefore multiple teachers are assigned multiple classes.

Pass the following query in MySQL interface and the result of **teacher** table will be displayed as follows:

```
Select * from teacher
```

Tid	Subject	Tname
354	Java	Satish
355	Adv Java	Krishna
356	DB2	Masthanvali

Pass the following query in MySQL interface and the result of **class** table will be displayed as follows:

```
Select * from class
```

cid	Cname
351	1st
352	2nd

tutorialspoint
SIMPLYEASYLEARNING

353	3rd

tutorialspoint
SIMPLYEASYLEARNING

9. JPA – CRITERIA API

Criteria is a predefined API that is used to define queries for entities. It is an alternative way of defining a JPQL query. These queries are type-safe, portable, and easy to modify by changing the syntax. Similar to JPQL, it follows an abstract schema (easy to edit schema) and embedded objects. The metadata API is mingled with criteria API to model persistent entity for criteria queries.

The major advantage of Criteria API is that errors can be detected earlier during the compile time. String-based JPQL queries and JPA criteria based queries are same in performance and efficiency.

History of Criteria

Criteria is included in all the versions of JPA, therefore each step of Criteria is notified in the specifications of JPA.

- In JPA 2.0, Criteria query API, standardization of queries are developed.
- In JPA 2.1, Criteria update and delete (bulk update and delete) are included.

Criteria Query Structure

Criteria and JPQL are closely related and are allowed to design using similar operators in their queries. It follows the **javax.persistence.criteria** package to design a query. The query structure means the syntax criteria query.

The following simple criteria query returns all the instances of the entity class in the data source.

```
EntityManager em = ...;
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Entity class> cq = cb.createQuery(Entity.class);
Root<Entity> from = cq.from(Entity.class);
cq.select(Entity);
TypedQuery<Entity> q = em.createQuery(cq);
List<Entity> allitems = q.getResultList();
```

The query demonstrates the basic steps to create a criteria.

tutorialspoint
SIMPLYEASYLEARNING

1. `EntityManager` instance is used to create a `CriteriaBuilder` object.
2. `CriteriaQuery` instance is used to create a query object. The attributes of this object will be modified with the details of the query.
3. `CriteriaQuery.form` method is called to set the query root.
4. `CriteriaQuery.select` is called to set the result list type.
5. `TypedQuery<T>` instance is used to prepare a query for execution and specifying the type of the query result.
6. `getResultList` method is used on the `TypedQuery<T>` object to execute a query. This query returns a collection of entities and the result is stored in a `List`.

Example of Criteria API

Let us consider the example of employee database. Let us assume the jpadb.employee table contains the following records:

Eid	Ename	Salary	Deg
401	Gopal	40000	Technical Manager
402	Manisha	40000	Proof reader
403	Masthanvali	35000	Technical Writer
404	Satish	30000	Technical writer
405	Krishna	30000	Technical Writer
406	Kiran	35000	Proof reader

Create a JPA Project in the eclipse IDE named **JPA_Eclipselink_Criteria**. All the modules of this project are discussed below.

Creating Entities

Create a package named **com.tutorialspoint.eclipselink.entity** under the **src** package. Create a class named **Employee.java** under the given package. The class Employee entity is as follows:

```
package com.tutorialspoint.eclipselink.entity;
```


tutorialspoint
SIMPLYEASYLEARNING

```
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;

@Entity
public class Employee
{
      @Id
      @GeneratedValue(strategy= GenerationType.AUTO)
      private int eid;
      private String ename;
      private double salary;
      private String deg;
      public Employee(int eid, String ename, double salary, String deg)
      {
            super( );
            this.eid = eid;
            this.ename = ename;
            this.salary = salary;
            this.deg = deg;
      }

      public Employee( )
      {
            super();
      }

      public int getEid( )
      {
            return eid;
      }
      public void setEid(int eid)
      {
            this.eid = eid;
```


tutorialspoint
SIMPLYEASYLEARNING

```
	}

	public String getEname( )
	{
		return ename;
	}
	public void setEname(String ename)
	{
		this.ename = ename;
	}

	public double getSalary( )
	{
		return salary;
	}
	public void setSalary(double salary)
	{
		this.salary = salary;
	}

	public String getDeg( )
	{
		return deg;
	}
	public void setDeg(String deg)
	{
		this.deg = deg;
	}
	@Override
	public String toString() {
		return "Employee [eid=" + eid + ", ename=" + ename + ", salary="
				+ salary + ", deg=" + deg + "]";
	}
}
```


tutorialspoint
SIMPLYEASYLEARNING

Persistence.xml

The persistence.xml file is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0" 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">
     <persistence-unit name="Eclipselink_JPA"
                       transaction-type="RESOURCE_LOCAL">
     <class>com.tutorialspoint.eclipselink.entity.Employee</class>
            <properties>
                  <property name="javax.persistence.jdbc.url"
                            value="jdbc:mysql://localhost:3306/jpadb"/>
                  <property name="javax.persistence.jdbc.user" value="root"/>
                  <property name="javax.persistence.jdbc.password"
                            value="root"/>
                  <property name="javax.persistence.jdbc.driver"
                            value="com.mysql.jdbc.Driver"/>
                  <property name="eclipselink.logging.level" value="FINE"/>
                  <property name="eclipselink.ddl-generation"
                            value="create-tables"/>
                  </properties>
     </persistence-unit>
</persistence>
```

Service classes

This module contains the service classes to implement the Criteria query part using the MetaData API initialization. Create a package named **com.tutorialspoint.eclipselink.service**. The class named **CriteriaAPI.java** is created under the given package. The DAO class is as follows:

```
package com.tutorialspoint.eclipselink.service;


import java.util.List;
import javax.persistence.EntityManager;
```


tutorialspoint
SIMPLYEASYLEARNING

```
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import javax.persistence.TypedQuery;
import javax.persistence.criteria.CriteriaBuilder;
import javax.persistence.criteria.CriteriaQuery;
import javax.persistence.criteria.Root;

import com.tutorialspoint.eclipselink.entity.Employee;

public class CriteriaApi
{
      public static void main(String[] args)
      {
            EntityManagerFactory emfactory = Persistence.
                        createEntityManagerFactory( "Eclipselink_JPA" );
            EntityManager entitymanager = emfactory.
                        createEntityManager( );
            CriteriaBuilder criteriaBuilder = entitymanager
                        .getCriteriaBuilder();
            CriteriaQuery<Object> criteriaQuery = criteriaBuilder
                        .createQuery();
            Root<Employee> from = criteriaQuery.from(Employee.class);

            //select all records
            System.out.println(“Select all records”);
            CriteriaQuery<Object> select =criteriaQuery.select(from);
            TypedQuery<Object> typedQuery = entitymanager
                        .createQuery(select);
            List<Object> resultlist= typedQuery.getResultList();

            for(Object o:resultlist)
            {
                  Employee e=(Employee)o;
                  System.out.println("EID : "+e.getEid()
                              +" Ename : "+e.getEname());
```


tutorialspoint
SIMPLYEASYLEARNING

```
            }

      //Ordering the records
       System.out.println("Select all records by follow ordering");
      CriteriaQuery<Object> select1 = criteriaQuery.select(from);
       select1.orderBy(criteriaBuilder.asc(from.get("ename")));
       TypedQuery<Object> typedQuery1 = entitymanager
                   .createQuery(select);
       List<Object> resultlist1= typedQuery1.getResultList();

            for(Object o:resultlist1)
            {
                  Employee e=(Employee)o;
                  System.out.println("EID : "+e.getEid()
                              +" Ename : "+e.getEname());
            }

            entitymanager.close( );
            emfactory.close( );
      }
}
```

After compiling and executing the above program, you will get the following output.

```
Select All records
EID : 401 Ename : Gopal
EID : 402 Ename : Manisha
EID : 403 Ename : Masthanvali
EID : 404 Ename : Satish
EID : 405 Ename : Krishna
EID : 406 Ename : Kiran
Select All records by follow Ordering
EID : 401 Ename : Gopal
EID : 406 Ename : Kiran
EID : 405 Ename : Krishna
EID : 402 Ename : Manisha
```


tutorialspoint
SIMPLYEASYLEARNING

```
EID : 403 Ename : Masthanvali
EID : 404 Ename : Satish
```


tutorialspoint
SIMPLYEASYLEARNING