Refer to : http://blog.woniper.net/255?category=531455
http://www.javajigi.net/pages/viewpage.action?pageId=5924
https://www.slideshare.net/zipkyh/spring-datajpa?next_slideshow=1
https://www.tutorialspoint.com/jpa/index.htm

1. JPA란 무엇인가?
  1)JPA(Java Persistent API)
  2)JPA는 여러 ORM 전문가가 참여한 EJB 3.0 스펙 작업에서 기존 EJB ORM이던 Entity Bean을 JPA라고 바꾸고 JavaSE, JavaEE를 위한 영속성(persistence) 관리와 ORM을 위한 표준 기술이다.
  3)A collection of classes and methods to persistently store the vast amounts of data into a database.
  4)The basic understanding of Persistence (storing the copy of database object into temporary memory), and The understanding of JAVA Persistence API (JPA).
  5)JPA는 ORM 표준 기술로 Hibernate, OpenJPA, EclipseLink, TopLink Essentials과 같은 구현체가 있고 이에 표준 인터페이스가 바로 JPA이다.
  6)ORM(Object Relational Mapping)이란 RDB 테이블을 객체지향적으로 사용하기 위한 기술이다.
  7)RDB 테이블은 객체지향적 특징(상속, 다형성, 레퍼런스, 오브젝트 등)이 없고 자바와 같은 언어로 접근하기 쉽지 않다.
  8)이 때문에 ORM을 사용해 오브젝트와 RDB 사이에 존재하는 개념과 접근을 객체지향적으로 다루기 위한 기술이다.
  9)장점
    -객체지향적으로 데이터를 관리할 수 있기 때문에 비즈니스 로직에 집중 할 수 있으며, 객체지향 개발이 가능하다.
    -테이블 생성, 변경, 관리가 쉽다. (JPA를 잘 이해하고 있는 경우)
    -로직을 쿼리에 집중하기 보다는 객체자체에 집중 할 수 있다.
    -빠른 개발이 가능하다.
  10)단점
    -어렵다. 장점을 더 극대화 하기 위해서 알아야 할게 많다.
    -잘 이해하고 사용하지 않으면 데이터 손실이 있을 수 있다. (persistence context)
    -성능상 문제가 있을 수 있다.(이 문제 또한 잘 이해해야 해결이 가능하다.)

  11)History
    -Earlier versions of EJB, defined persistence layer combined with 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.

  12)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, etc.

2. JPA - Architecture
  -Java Persistence API is a source to store business entities as relational entities.
  -It shows how to define a PLAIN OLD JAVA OBJECT (POJO) as an entity and how to manage entities with relations.
  1)Class Level Architecture
    -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 factory for Query instance.
Entity	Entities are the persistence objects, stores as records in the database.
EntityTransaction	It has one-to-one relationship with EntityManager. For each EntityManager, operations are maintained by EntityTransaction class.
Persistence	This class contain static methods to obtain EntityManagerFactory instance.
Query	This interface is implemented by each JPA vendor to obtain 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.

2)JPA Class Relationships
-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.
--Many 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.

3. ORM
1)ORM is a programming ability to covert data from object type to relational type and vice versa.
2)The main feature of ORM is mapping or binding an object to its data in the database.
3)While mapping we have to consider the data, type of data and its relations with its self-entity or entity in any other table.
4)Advanced Features
-Idiomatic persistence : It enables you to write the persistence classes using object oriented classes.
-High Performance : It has many fetching techniques and hopeful locking techniques.
-Reliable : It is highly stable and eminent. Used by many industrial programmers.

5)ORM Architecture
-Phase1
--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 schema-
--Employee POJO class contain attributes such as ID, name, salary, and designation.
--And methods like setter and getter methods of those attributes.
--Employee DAO/Service classes contains service methods such as create employee, find employee, and delete employee.

-Phase 2
--The second phase named as mapping or persistence phase which contains JPA provider, mapping file (ORM.xml), JPA Loader, and Object Grid.
--JPA Provider : The vendor product which contains JPA flavor (javax.persistence).
--For example Eclipselink, Toplink, Hibernate, etc.
--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 cache memory, which 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 : The Object grid is a temporary location which can store the copy of relational data, i.e. 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 effects the main database.

-Phase 3
--The third phase is the Relational data phase.
--It contains the relational data which is logically connected to the business component.
--As discussed above, only when the business component commit the data, it is stored into the database physically.
--Until then the modified data is stored in a cache memory as a grid format.
--Same is the process for obtaining data.

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

6)Mapping.xml
-The mapping.xml file is to instruct the JPA vendor for mapping the Entity classes with database tables.
-Let us take an example of Employee entity which 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) {
        this.eid = eid;
        this.ename = ename;
        this.salary = salary;
        this.deg = deg;
    }

    public Employee( ) {}

    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( ) {
        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 are the table fields in the database and eid is the primary key of this table.
-Now we have to design 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>
```

-The above script for mapping the entity class with database table. In this file
--<entity-mappings> : tag defines the schema definition to allow entity tags into xml file.
--<description> : tag defines description about application.
--<entity> : tag defines the entity class which you want to convert into table in a database. Attribute class defines the POJO entity class name.
--<table> : tag defines the table name. If you want to keep class name as table name 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 remaining attributes for table.
--<column-name> : tag is used to define user defined table field name.

7)Annotations
-Generally Xml files are used to configure specific component, or mapping two different specifications of components.
-In our case, we have to maintain xml separately in a framework.
-That means while writing a mapping xml file we need to compare the POJO class attributes with entity tags in mapping.xml file.
-Here is the solution: In the class definition, we can write the configuration part using annotations.
-The annotations are used for classes, properties, and methods.
-Annotations starts with '@' symbol.
-Annotations are declared before the class, property or method is declared.
-All annotations of JPA are defined in javax.persistence package.
-Here follows the list of annotations used in our examples

--@Entity
---This annotation specifies to declare the class as entity or a table.
--@Table
---This annotation specifies to declare table name.
--@Basic
---This annotation specifies non constraint fields explicitly.
--@Embedded
---This annotation specifies the properties of class or an entity whose value instance of an embeddable class.
--@Id
---This annotation specifies the property, use for identity (primary key of a table) of the class.
--@GeneratedValue
---This annotation specifies, how the identity attribute can be initialized such as Automatic, manual, or value taken from sequence table.
--@Transient

---This annotation specifies the property which in not persistent i.e. the value is never stored into database.
--@Column
---This annotation is used to specify column or attribute for persistence property.
--@SequenceGenerator
---This annotation is used to define the value for the property which is specified in @GeneratedValue annotation. It creates a sequence.
--@TableGenerator
---This annotation is used to specify the value generator for property specified in @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 Field wise access will occur. If you set @AccessType(PROPERTY) then Property wise assess will occur.
--@JoinColumn
---This annotation is used to specify an entity association or entity collection. This is used in many- to-one and one-to-many associations.
--@UniqueConstraint
---This annotation is used to specify the field, unique constraint for primary or secondary table.
--@ColumnResult
---This annotation references the name of a column in the SQL query using select clause.
--@ManyToMany
---This annotation is used to define a many-to-many relationship between the join Tables.
--@ManyToOne
---This annotation is used to define a many-to-one relationship between the join Tables.
--@OneToMany
---This annotation is used to define a one-to-many relationship between the join Tables.
--@OneToOne
---This annotation is used to define a one-to-one relationship between the join Tables.
--@NamedQueries
---This annotation is used for specifying list of named queries.
--@NamedQuery
---This annotation is used for specifying a Query using static name.

8)Java Bean Standard
-Java class, encapsulates the instance values and behaviors into a single unit callled object.
-Java Bean is a temporary storage and reusable component or an object.
-It is a serializable class which has default constructor and getter & setter methods to initialize the instance attributes individually.

9)Bean Conventions
-Bean contains the default constructor or a file that contains serialized instance.
-Therefore, a bean can instantiate the bean.
-The properties of a bean can be segregated into Boolean properties and non-Boolean properties.
-Non-Boolean property contains getter and setter methods.
-Boolean property contain 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 capital letter.
-E.g. 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 capital letter and the argument value to set to field.
-E.g. the field name is 'salary' therefore the setter method of this field is 'setSalary (double sal)'.
-For Boolean property, is method to check if it is true or false. E.g. the Boolean property 'empty', the is method of this field is 'isEmpty ()'.

4. Lab
1)JPA Project 생성
-In Package Explorer > right-click > New > Other > JPA > JPA Project
-Project name : Demo
-Target runtime : jdk 1.8.0_162 > Next
-Platform : Generic 2.1
-Type : User Library > Download library...
-Download Library : EclipseLink 2.5.2 > Next > Check I accpet... > Finish
-Finish

-Open Perspective

2)Adding H2 database connector to Project
-Go to Project properties > Java Build Path by right click on it.
-Click on Add External Jars.
-Select C:\Program Files (x86)\H2\bin\h2-1.4.197.jar > 열기
-Apply and Close

3)Entity Managers
-The main modules for this example are as follows:

Model or POJO
Employee.java
Persistence
Persistence.xml
Service
CreatingEmployee.java
UpdatingEmployee.java
FindingEmployee.java
DeletingEmployee.java

4)Creating Entities
-src/com.javasoft package 생성 > righ-click > New > Class
-com.javasoft.Employee.java

```
package com.javasoft;

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

@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) {
      this.eid = eid;
      this.ename = ename;
      this.salary = salary;
      this.deg = deg;
   }

   public Employee( ) {}

   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;
   }
```

```
354
355             public double getSalary( ) {
356                return salary;
357             }
358
359             public void setSalary(double salary) {
360                this.salary = salary;
361             }
362
363             public String getDeg( ) {
364                return deg;
365             }
366
367             public void setDeg(String deg) {
368                this.deg = deg;
369             }
370
371             @Override
372             public String toString() {
373                return "Employee [eid=" + eid + ", ename=" + ename + ", salary=" + salary + ",
                   deg=" + deg + "]";
374             }
375          }
376
377       -In the above code, we have used @Entity annotation to make this POJO class as entity.
378       -Before going to next module we need to create database for relational entity, which will register
          the database in persistence.xml file.
379
380    5)src/META-INF/persistence.xml
381       -This module plays a crucial role in the concept of JPA.
382       -In this xml file we will register the database and specify the entity class.
383       -In the above shown package hierarchy, persistence.xml under JPA Content package is as follows:
384
385          <?xml version="1.0" encoding="UTF-8"?>
386          <persistence version="2.1"
387             xmlns="http://xmlns.jcp.org/xml/ns/persistence"
388             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
389             xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
                http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
390             <persistence-unit name="Demo" transaction-type="RESOURCE_LOCAL">
391                <class>com.javasoft.Employee</class>
392                <properties>
393                   <property name="javax.persistence.jdbc.url"
394                      value="jdbc:h2:tcp://localhost/~/test" />
395                   <property name="javax.persistence.jdbc.driver"
396                      value="org.h2.Driver" />
397                   <property name="javax.persistence.jdbc.user" value="sa" />
398                   <!-- <property name="javax.persistence.jdbc.password" value=""/> -->
399                   <property name="eclipselink.logging.level" value="FINE" />
400                   <property name="eclipselink.ddl-generation" value="create-tables" />
401                </properties>
402             </persistence-unit>
403          </persistence>
404
405       -In the above xml, <persistence-unit> tag is defined with specific name for JPA persistence.
406       -The <class> tag defines entity class with package name.
407       -The <properties> tag defines all the properties, and <property> tag defines each property
          such as database registration, URL specification, username, and password.
408       -These are the Eclipselink properties.
409       -This file will configure the database.
410
411    6)Persistence Operations
412       -Persistence operations are used against database and they are load and store operations.
413       -In a business component all the persistence operations fall under service classes.
414       -In the above shown package hierarchy, create a package named 'com.javasoft.service', under
          'src' (source) package.
415       -All the service classes named as CreateEmloyee.java, UpdateEmployee.java,
```

FindEmployee.java, and DeleteEmployee.java. comes under the given package as follows:

-src/com.javasoft.service package 생성

7)Create Employee

-Creating an Employee class named as CreateEmployee.java as follows:

```
package com.javasoft.service;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

import com.javasoft.Employee;

public class CreateEmployee {
   public static void main( String[ ] args ) {
      EntityManagerFactory emfactory = Persistence.createEntityManagerFactory( "Demo" );

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

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

-In the above code the createEntityManagerFactory () creates a persistence unit by providing the same unique name which we provide for persistence-unit in persistent.xml file.

-The entitymanagerfactory object will create the entitymanger instance by using createEntityManager () method.

-The entitymanager object creates entitytransaction instance for transaction management.

-By using entitymanager object, we can persist entities into database.

-After compilation and execution of 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.

```
SELECT * FROM employee
```

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

Eid	Ename	Salary	Deg
1201	Gopal	40000.0	Technical Manager

8)Update Employee

-src/com.javasoft.service/UpdateEmployee.java

```
package com.javasoft.service;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

import com.javasoft.Employee;

public class UpdateEmployee {
   public static void main( String[ ] args ) {
        EntityManagerFactory emfactory = Persistence.createEntityManagerFactory( "Demo" );

        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();
        }
    }
```

-The salary of employee, 1201 is updated to 46000.

9)Find Employee

-src/com.javasoft.service/FindEmployee.java

```
    package com.javasoft.service;

    import javax.persistence.EntityManager;
    import javax.persistence.EntityManagerFactory;
    import javax.persistence.Persistence;

    import com.javasoft.Employee;

    public class FindEmployee {
        public static void main( String[ ] args ) {
            EntityManagerFactory emfactory = Persistence.createEntityManagerFactory( "Demo" );
            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( ));
        }
    }
```

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

10)Delete Employee

-src/com.javasoft.service/DeleteEmployee.java

```
    package com.javasoft.service;

    import javax.persistence.EntityManager;
    import javax.persistence.EntityManagerFactory;
    import javax.persistence.Persistence;

    import com.javasoft.Employee;

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

            Employee employee = entitymanager.find( Employee.class, 1201 );
            entitymanager.remove( employee );
            entitymanager.getTransaction( ).commit( );
            entitymanager.close( );
            emfactory.close( );
```

```
        }
    }
```

5. JPQL

1)Java Persistence Query language

-JPQL is Java Persistence Query Language defined in JPA specification.
-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 information or data using SELECT clause, can do bulk updates using UPDATE clause and DELETE clause.
-EntityManager.createQuery() API will support for querying language.

2)Query Structure

-JPQL syntax is very similar to the syntax of SQL.
-Having SQL like syntax is an advantage because SQL is a simple structured query language and many developers are using it in applications.
-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 field result set from database, as with SQL.
-The JPQL query structure as follows.

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

-The structure of JPQL DELETE and UPDATE queries is simpler as follows.

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

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

3)Scalar and Aggregate Functions

-Scalar functions returns resultant values based on input values.
-Aggregate functions returns the resultant values by calculating the input values.
-Follow the same example employee management used in previous chapters.
-Here we will go through the service classes using scalar and aggregate functions of JPQL.
-Let us assume the jpadb.employee table contains 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

4)src/com.javasoft.service/ScalarandAggregateFunctions.java

```
package com.javasoft.service;

import java.util.List;

import javax.persistence.EntityManager;
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( "Demo" );
        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);
         }
      }

      Employee NAME :GOPAL
      Employee NAME :MANISHA
      Employee NAME :MASTHANVALI
      Employee NAME :SATISH
      Employee NAME :KRISHNA
      Employee NAME :KIRAN

   5)Between, And, Like Keywords
      -‘Between’, ‘And’, and ‘Like’ are the main keywords of JPQL.
      -These keywords are used after Where clause in a query.
      -src/com.javasoft.service/BetweenAndLikeFunctions.java

         package com.javasoft.service;

         import java.util.List;

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

         import com.javasoft.Employee;

         public class BetweenAndLikeFunctions {
            public static void main( String[ ] args ) {

               EntityManagerFactory emfactory = Persistence.createEntityManagerFactory( "Demo" );
               EntityManager entitymanager = emfactory.createEntityManager();

               //Between
               Query query = entitymanager.createQuery( "Select e " + "from Employee e " + "where
               e.salary " + "Between 30000 and 40000" );

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

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

```
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
[EL Fine]: sql: 2018-06-21 12:23:05.205--ServerSession(403716510)--Connection(445288316)--Thread(Thread[main,5,main])--SELECT EID, DEG, ENAME, SALARY FROM EMPLOYEE WHERE ENAME LIKE ?
   bind => [M%]
Employee ID :1202    Employee name :Manisha
Employee ID :1203    Employee name :Masthanvali
```

6)Ordering

-To Order the records in JPQL we use ORDER BY clause.

-Ordering.java

```
package com.javasoft.service;

import java.util.List;

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

import com.javasoft.Employee;

public class Ordering {
   public static void main( String[ ] args ) {
      EntityManagerFactory emfactory = Persistence.createEntityManagerFactory( "Demo" );
      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( ));
      }
   }
}
```

```
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
```

7)Named Queries

-A @NamedQuery annotation is defined as a query with a predefined unchangeable query string.

-Instead of dynamic queries, usage of named queries may improve code organization by separating the JPQL query strings from POJO.

-It also passes the query parameters rather than embedding literals dynamically into the query string and results in more efficient queries.

-First of all, add @NamedQuery annotation to the Employee entity class named Employee.java.

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

```

-src/com.javasoft.service/NamedQueries.java

```
package com.javasoft.service;

import java.util.List;

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

import com.javasoft.Employee;

public class NamedQueries {
   public static void main( String[ ] args ) {

      EntityManagerFactory emfactory = Persistence.createEntityManagerFactory( "Demo" );
      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( ));
      }
   }
}
```

Employee ID :1204 Employee Name :Satish

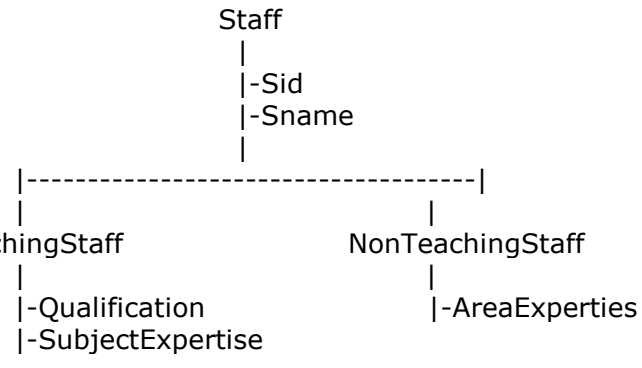
6. Advanced Mappings

1)Inheritance Strategies

-Inheritance is the core concept of object oriented language, therefore we can use inheritance relationships or strategies between entities.

-JPA support three types of inheritance strategies such as SINGLE_TABLE, JOINED_TABLE, and TABLE_PER_CONCRETE_CLASS.

-Let us consider an example of Staff, TeachingStaff, NonTeachingStaff classes and their relationships as follows:



2)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 discriminator value plays key role in differentiating the values of three entities in one table.

-Let us consider the above example, TeachingStaff and NonTeachingStaff are the sub classes of class Staff.

-Remind the concept of inheritance (is a mechanism of inheriting the properties of super class by sub class) and therefore sid, sname are the fields which belongs to both TeachingStaff and NonTeachingStaff.

-Create a JPA project.

-All the modules of this project as follows:

-Creating Entities

--src/com.javasoft.entity package 생성
--src/com.javasoft.entity.Staff.java

```
package com.javasoft.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 ) {
      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;
  }
}
```

-In the above code @DescriminatorColumn specifies the field name (type) and the values of it shows the remaining (Teaching and NonTeachingStaff) fields.
-Create a subclass (class) to Staff class named TeachingStaff.java under the com.javasoft.entity package.
-The TeachingStaff Entity class is shown as follows:

```
package com.javasoft.entity;

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

@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;
    }

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

-Create a subclass (class) to Staff class named NonTeachingStaff.java under the com.javasoft.entity package.

-The NonTeachingStaff Entity class is shown as follows:

```
package com.javasoft.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( );
    }

    public String getAreaexpertise( ) {
        return areaexpertise;
    }

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

-META-INF/persistence.xml

-Persistence.xml file contains the configuration information of database and registration information of entity classes.

-The xml file is shown as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1"
   xmlns="http://xmlns.jcp.org/xml/ns/persistence"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
   http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
   <persistence-unit name="Demo"
      transaction-type="RESOURCE_LOCAL">
      <class>com.javasoft.entity.Staff</class>
      <class>com.javasoft.entity.NonTeachingStaff</class>
      <class>com.javasoft.entity.TeachingStaff</class>
      <properties>
         <property name="javax.persistence.jdbc.url"
            value="jdbc:h2:tcp://localhost/~/test" />
         <property name="javax.persistence.jdbc.driver"
            value="org.h2.Driver" />
         <property name="javax.persistence.jdbc.user" value="sa" />
         <!-- <property name="javax.persistence.jdbc.password" value=""/> -->
         <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 business component.

--Create a package under 'src' package named 'com.javasoft.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.javasoft.service;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

import com.javasoft.entity.NonTeachingStaff;
import com.javasoft.entity.TeachingStaff;

public class SaveClient {
   public static void main( String[ ] args ) {

      EntityManagerFactory emfactory = Persistence.createEntityManagerFactory( "Demo" );
      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);
      entitymanager.persist(ts2);
      entitymanager.persist(nts1);
      entitymanager.persist(nts2);

      entitymanager.getTransaction().commit();

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

```
    }
  }
```

--After compilation and execution of the above program you will get notifications in the console panel of Eclipse IDE.
--The output in a tabular format is shown as follows:

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

--Finally you will get single table which contains all three class's fields and differs with discriminator column named 'Type' (field).

3)Joined table Strategy
-Joined table strategy is to share the referenced column which 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 shown as follows:
-Creating Entities
--Create a package named 'com.javasoft.entity' under 'src' package.
--Create a new java class named Staff.java under given package.
--The Staff entity class is shown as follows:

```
package com.javasoft.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( );
     }

     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.javasoft.entity package.
--The TeachingStaff Entity class is shown as follows:

```
    package com.javasoft.entity;

    import javax.persistence.Entity;
    import javax.persistence.PrimaryKeyJoinColumn;

    @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
        ) {
          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.javasoft.entity package.
--The NonTeachingStaff Entity class is shown as follows:

```
    package com.javasoft.entity;

    import javax.persistence.Entity;
    import javax.persistence.PrimaryKeyJoinColumn;

    @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;
        }
    }
```

-META-INF/persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1" xmlns="http://xmlns.jcp.org/xml/ns/persistence" xmlns:xsi="
http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="
http://xmlns.jcp.org/xml/ns/persistence
http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
    <persistence-unit name="Demo1" transaction-type="RESOURCE_LOCAL">
        <class>com.javasoft.entity.Staff</class>
        <class>com.javasoft.entity.NonTeachingStaff</class>
        <class>com.javasoft.entity.TeachingStaff</class>
        <properties>
            <property name="javax.persistence.jdbc.url"
                value="jdbc:h2:tcp://localhost/~/test" />
            <property name="javax.persistence.jdbc.driver"
                value="org.h2.Driver" />
            <property name="javax.persistence.jdbc.user" value="sa" />
            <!-- <property name="javax.persistence.jdbc.password" value=""/> -->
            <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 business component.

--Create a package under 'src' package named 'com.javasoft.service'.

--Create a class named SaveClient.java under the given package to store Staff, TeachingStaff, and NonTeachingStaff class fields.

--Then SaveClient class as follows:

```
package com.javasoft.service;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

import com.javasoft.entity.NonTeachingStaff;
import com.javasoft.entity.TeachingStaff;

public class SaveClient {
    public static void main( String[ ] args ) {
        EntityManagerFactory emfactory = Persistence.createEntityManagerFactory( "Demo1"
        );
        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);
                entitymanager.persist(ts2);
                entitymanager.persist(nts1);
                entitymanager.persist(nts2);

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

-After compilation and execution of the above program you will get notifications in the console panel of Eclipse IDE.
-Here three tables are created and the result of staff table in a tabular format is shown as follows:

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

-The result of TeachingStaff table in a tabular format is shown 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 in tabular format is shown as follows:

Sid	Areaexpertise
3	Accounts
4	Office Admin

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

4)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 records.
-The field values of Staff table must be shared by TeachingStaff and NonTeachingStaff tables.
-Let us consider the same example as above.
-All modules of this project are shown as follows:

-Creating Entities
--Create a package named 'com.javasoft.entity' under 'src' package.
--Create a new java class named Staff.java under given package.
--The Staff entity class is shown as follows:

```
package com.javasoft.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;

   public Staff(int sid, String sname) {
      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 Staff class named TeachingStaff.java under the com.javasoft.entity package.

--The TeachingStaff Entity class is shown as follows:

```
package com.javasoft.entity;

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;
    }

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

--Create a subclass (class) to Staff class named NonTeachingStaff.java under the com.javasoft.entity package.

--The NonTeachingStaff Entity class is shown as follows:

```
  package com.javasoft.entity;

  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;
    }

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

-META-INF/persistence.xml

```
  <?xml version="1.0" encoding="UTF-8"?>
  <persistence version="2.1" xmlns="http://xmlns.jcp.org/xml/ns/persistence" xmlns:xsi="
  http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="
  http://xmlns.jcp.org/xml/ns/persistence
  http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
     <persistence-unit name="Demo1" transaction-type="RESOURCE_LOCAL">
        <class>com.javasoft.entity.Staff</class>
        <class>com.javasoft.entity.NonTeachingStaff</class>
        <class>com.javasoft.entity.TeachingStaff</class>
        <properties>
           <property name="javax.persistence.jdbc.url"
              value="jdbc:h2:tcp://localhost/~/test" />
           <property name="javax.persistence.jdbc.driver"
              value="org.h2.Driver" />
           <property name="javax.persistence.jdbc.user" value="sa" />
           <!-- <property name="javax.persistence.jdbc.password" value=""/> -->
           <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 business component.

--Create a package under 'src' package named 'com.javasoft.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.javasoft.service;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

import com.javasoft.entity.NonTeachingStaff;
import com.javasoft.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);
      entitymanager.persist(ts2);
      entitymanager.persist(nts1);
      entitymanager.persist(nts2);

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

--After compilation and execution of the above program you will get notifications in the console panel of Eclipse IDE.
--Here the three tables are created and the Staff table contains null records.
--The result of TeachingStaff in a tabular format is shown 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 in a tabular format is shown as follows:

Sid	Areaexpertise	Sname
3	Accounts	Satish
4	Office Admin	Krishna

--The above table NonTeachingStaff contains fields of both Staff and NonTeachingStaff Entities.

7. Entity Relationships

1)This chapter takes you through the relationships between Entities.
2)Generally the relations are more effective between tables in the database.
3)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

4)@ManyToOne Relation
-Where one entity (column or set of columns) is/are referenced with another entity (column or set of columns) which contain unique values.
-In relational databases these relations are applicable by using foreign key/primary key between tables.
-Let us consider an example of 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 Department table.
-Here in the Employee table, Department id is foreign Key.
-Create a JPA project in eclipse IDE named JPA_MTO.
-All the modules of this project are shown as follows:

-Creating Entities
--Create a package named 'com.javasoft.entity' under 'src' package.
--Create a class named Department.java under given package.
--The class Department entity is shown as follows:

```
package com.javasoft.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;
   }
}
```

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

```
package com.javasoft.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;
      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;
   }

   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 file 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 file is shown as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1" xmlns="http://xmlns.jcp.org/xml/ns/persistence" xmlns:xsi="
http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="
http://xmlns.jcp.org/xml/ns/persistence
http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
   <persistence-unit name = "JPA_MTO" transaction-type = "RESOURCE_LOCAL">
      <class>com.javasoft.entity.Employee</class>
      <class>com.javasoft.entity.Department</class>

      <properties>
         <property name = "javax.persistence.jdbc.url" value = "jdbc:h2:
         tcp://localhost/~/test"/>
         <property name = "javax.persistence.jdbc.user" value = "sa"/>
         <property name = "javax.persistence.jdbc.driver" value="org.h2.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 'src' package named 'com.javasoft.service'.
--The DAO class named ManyToOne.java is created under given package.
--The DAO class is shown as follows:

```
package com.javasoft.service;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

import com.javasoft.entity.Department;
import com.javasoft.entity.Employee;

public class ManyToOne {
   public static void main(String[] args) {

      EntityManagerFactory emfactory =
      Persistence.createEntityManagerFactory("JPA_MTO");
      EntityManager entitymanager = emfactory.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);

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

-After compilation and execution of the above program you will get notifications in the console panel of Eclipse IDE.
-In this example two tables are created.
-Pass the following query in H2Database interface and the result of Department table in a tabular format is shown as follows in the query:

```
Select * from department;

Id      Name
1       Development
```

-Pass the following query in H2Database interface and the result of Employee table in a tabular format is shown as follows in the query:

```
Select * from employee;

Eid     Deg               Ename          Salary  Department_Id
2       Technical Writer  Satish         45000   1
3       Technical Writer  Krishna        45000   1
4       Technical Writer  Masthan Wali   50000   1
```

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

5)@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 is linked to 0, 1 or many rows in Table B.
-Let us consider the above example.
-If Employee and Department is in a reverse unidirectional manner, relation is Many-To-One relation.
-Create a JPA project in eclipse IDE named JPA_OTM.
-All the modules of this project are shown as follows:

-Creating Entities

--Create a package named 'com.javasoft.entity' under 'src' package.
--Create a class named Department.java under given package.
--The class Department entity is shown as follows:

```
package com.javasoft.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) {
      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 'com.javasoft.entity' package.
--The Employee entity class is shown as follows:

```
package com.javasoft.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;

   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;
    }

    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

--Persistence.xml file 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 file is shown as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1"
    xmlns="http://xmlns.jcp.org/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
    http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
    <persistence-unit name="JPA_OTM" transaction-type="RESOURCE_LOCAL">
        <class>com.javasoft.entity.Employee</class>
        <class>com.javasoft.entity.Department</class>

        <properties>
            <property name="javax.persistence.jdbc.url" value="jdbc:h2:
            tcp://localhost/~/test" />
            <property name="javax.persistence.jdbc.user" value="sa" />
            <property name="javax.persistence.jdbc.driver" value="org.h2.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 'src' package named 'com.javasoft.service'.

--The DAO class named OneToMany.java is created under given package.

--The DAO class is shown as follows:

```
package com.javasoft.service;

import java.util.ArrayList;
import java.util.List;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

import com.javasoft.entity.Department;
import com.javasoft.entity.Employee;

public class OneToMany {
   public static void main(String[] args) {

      EntityManagerFactory emfactory =
      Persistence.createEntityManagerFactory("JPA_OTM");
      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);

      // 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();
        }
    }
```

-After compilation and execution of the above program you will get notifications in the console panel of Eclipse IDE.
-In this example two tables are created.
-Pass the following query in H2Database interface and the result of Department_employee table in a tabular format is shown as follows in the query:

```
SELECT * FROM department_employee;

Department_Id    Employee_Eid
54               51
54               52
54               53
```

-In the above table, deparment_id and employee_id fields are the foreign keys (reference fields) from department and employee tables.
-Pass the following query in H2Database interface and the result of department table in a tabular format is shown as follows in the query:

```
Select * from department;

Id      Name
54      Development
```

-Pass the following query in H2Database interface and the result of employee table in a tabular format is shown as follows in the query:

```
Select * from employee;

Eid     Deg                 Ename          Salary
51      Technical Writer    Satish         45000
52      Technical Writer    Krishna        45000
53      Technical Writer    Masthanvali      50000
```

6)OneToOne Relation
-In One-To-One relationship, one item can belong 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_OTO.
-All the modules of this project are shown as follows:

-Creating Entities
--Create a package named 'com.javasoft.entity' under 'src' package.
--Create a class named Department.java under given package.
--The class Department entity is shown as follows:

```
package com.javasoft.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;
    }
}
```

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

```
package com.javasoft.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;
    }

    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() {
            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 file 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 file is shown as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1"
    xmlns="http://xmlns.jcp.org/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
    http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
    <persistence-unit name="JPA_OTO" transaction-type="RESOURCE_LOCAL">
        <class>com.javasoft.entity.Employee</class>
        <class>com.javasoft.entity.Department</class>

        <properties>
            <property name="javax.persistence.jdbc.url" value="jdbc:h2:
            tcp://localhost/~/test" />
            <property name="javax.persistence.jdbc.user" value="sa" />
            <property name="javax.persistence.jdbc.driver" value="org.h2.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 ‘src’ package named ‘com.javasoft.service’.

--The DAO class named OneToOne.java is created under given package.

--The DAO class is shown as follows:

```
package com.javasoft.service;
```

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

import com.javasoft.entity.Department;
import com.javasoft.entity.Employee;

public class OneToOne {
   public static void main(String[] args) {

      EntityManagerFactory emfactory =
      Persistence.createEntityManagerFactory("JPA_OTO");
      EntityManager entitymanager = emfactory.createEntityManager();
      entitymanager.getTransaction().begin();

      // 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();
   }
}
```

-After compilation and execution of the above program you will get notifications in the console panel of Eclipse IDE.
-In this example two tables are created.
-Pass the following query in H2Database interface and the result of Department table in a tabular format is shown as follows in the query:

```
SELECT * FROM department;

Id      Name
101     Development
```

-Pass the following query in H2Database interface and the result of employee table in a tabular format is shown as follows in the query:

```
Select * from employee

Eid     Deg               Ename    Salary    Department_id
102     Technical Writer  Satish   45000     101
```

7)ManyToMany Relation

-Many-To-Many relationship is where one or more rows from one entity are associated with more than one row in other entity.
-Let us consider an example of relation between Class and Teacher entities.
-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 Teacher table and stored in Teacher_Class table and vice versa. Here, Teachers_Class table contains both foreign Key fields.
-Create a JPA project in eclipse IDE named JPA_MTM.

-All the modules of this project are shown as follows:

-Creating Entities
--Create a package named 'com.javasoft.entity' under 'src' package.
--Create a class named Clas.java under given package.
--The class Cals entity is shown as follows:

```
package com.javasoft.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;
   }

   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 - Teacher entity class named Teacher.java under

'com.javasoft.entity' package.
--The Teacher entity class is shown as follows:

```
package com.javasoft.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 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) {
      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
    --Persistence.xml file 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 file is shown as follows:

        <?xml version="1.0" encoding="UTF-8"?>
        <persistence version="2.1"
            xmlns="http://xmlns.jcp.org/xml/ns/persistence"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
            http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
            <persistence-unit name="JPA_MTM" transaction-type="RESOURCE_LOCAL">
                <class>com.javasoft.entity.Clas</class>
                <class>com.javasoft.entity.Teacher</class>

                <properties>
                    <property name="javax.persistence.jdbc.url" value="jdbc:h2:
                    tcp://localhost/~/test" />
                    <property name="javax.persistence.jdbc.user" value="sa" />
                    <property name="javax.persistence.jdbc.driver" value="org.h2.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 'src' package named 'com.javasoft.service'.
    --The DAO class named ManyToMany.java is created under given package.
    --The DAO class is shown as follows:

        package com.javasoft.service;

        import java.util.HashSet;
        import java.util.Set;

        import javax.persistence.EntityManager;
        import javax.persistence.EntityManagerFactory;
        import javax.persistence.Persistence;

        import com.javasoft.entity.Clas;
        import com.javasoft.entity.Teacher;

        public class ManyToMany {
            public static void main(String[] args) {

                EntityManagerFactory emfactory =
                Persistence.createEntityManagerFactory("JPA_MTM");
                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);

                // 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();
            }
        }
```

-After compilation and execution of the above program you will get notifications in the console panel of Eclipse IDE.
-In this example three tables are created.
-Pass the following query in H2Database interface and the result of teacher_clas table in a tabular format is shown as follows in the query:

```
Select * from teacher_clas;

Teacher _tid     Classet_cid
154                  151
155                  151
156                  151
154                  152
155                  152
156                  152
154                  153
155                  153
156                  153
```

-In the above table teacher_tid is the foreign key from teacher table, and classet_cid is the foreign key from class table.
-Therefore different teachers are allotted to different class.
-Pass the following query in H2Database interface and the result of teacher table in a tabular format is shown as follows in the query:

```
Select * from teacher;

Tid        Subject        Tname
154        Java           Satish
155        Adv Java       Krishna
156        DB2            Masthanvali
```

-Pass the following query in H2Database interface and the result of clas table in a tabular format is shown as follows in the query:

```
Select * from clas;
```

cid	Cname
151	1st
152	2nd
153	3rd

8. Criteria API

1)The Criteria API is a predefined API used to define queries for entities.
2)It is the alternative way of defining a JPQL query.
3)These queries are type-safe, and portable and easy to modify by changing the syntax.
4)Similar to JPQL it follows abstract schema (easy to edit schema) and embedded objects.
5)The metadata API is mingled with criteria API to model persistent entity for criteria queries.
6)The major advantage of the criteria API is that errors can be detected earlier during compile time.
7)String based JPQL queries and JPA criteria based queries are same in performance and efficiency.
8)History of criteria API
-The criteria API is included into all versions of JPA therefore each step of criteria API is notified in the specifications of JPA.
-In JPA 2.0, the criteria query API, standardization of queries are developed.
-In JPA 2.1, Criteria update and delete (bulk update and delete) are included.

9)Criteria Query Structure
-The Criteria API and the JPQL are closely related and are allowed to design using similar operators in their queries.
-It follows javax.persistence.criteria package to design a query.
-The query structure means the syntax criteria query.
-The following simple criteria query returns all 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.
--EntityManager instance is used to create a CriteriaBuilder object.
--CriteriaQuery instance is used to create a query object.
--This query object's attributes will be modified with the details of the query.
--CriteriaQuery.from method is called to set the query root.
--CriteriaQuery.select is called to set the result list type.
--TypedQuery<T> instance is used to prepare a query for execution and specifying the type of the query result.
--getResultList method on the TypedQuery<T> object to execute a query.
--This query returns a collection of entities, the result is stored in a List.

10)Example of criteria API
-Let us consider the example of employee database.
-Let us assume the jpadb.employee table contains 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__Criteria.

-All the modules of this project are shown as follows:

-Creating Entities
--Create a package named com.javasoft.entity under 'src' package.
--Create a class named Employee.java under given package.
--The class Employee entity is shown as follows:

```
package com.javasoft.entity;

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;
   }

   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 + "]";
    }
}
```

-Persistence.xml

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

--Persistence.xml will be created by the eclipse IDE while cresting a JPA Project.

--The configuration details are user specification.

--The persistence.xml file is shown as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1"
    xmlns="http://xmlns.jcp.org/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
    http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">

    <persistence-unit name="JPA_Criteria" transaction-type="RESOURCE_LOCAL">

        <class>com.javasoft.entity.Employee</class>
        <properties>
            <property name="javax.persistence.jdbc.url"
            value="jdbc:h2:tcp://localhost/~/test" />
            <property name="javax.persistence.jdbc.user" value="sa" />
            <property name="javax.persistence.jdbc.driver"
            value="org.h2.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 Criteria query part using the MetaData API initialization.

--Create a package named 'com.javasoft.service'.

--The class named CriteriaAPI.java is created under given package.

--The DAO class is shown as follows:

```
package com.javasoft.service;

import java.util.List;

import javax.persistence.EntityManager;
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.javasoft.entity.Employee;

public class CriteriaApi {
    public static void main(String[] args) {

        EntityManagerFactory emfactory = Persistence.createEntityManagerFactory(
        "JPA_Criteria" );
        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());
                }

                //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 compilation and execution of the above program you will get output in the console panel of Eclipse IDE as follows:

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

9. 왜 ORM 인가?

1)보통 한국은 iBatis/MyBatis를 현업에서 대부분 사용하는데, query 자체에 집중하고 business logic을 query에 의존하게 되면 객체지향의 장점을 놓칠 수 있는 가능성이 크다.

2)예를 들어 User table과 Order table이 있다.

3)크게 복잡하지 않은 1(user):N(order) 관계의 table이다.

```
tbl_user
user_id INT
username  VARCHAR(20)   PRIMARY KEY
nick_name  VARCHAR(20)
address     VARCHAR(100)

tbl_order
order_id   INT            PRIMARY KEY
user_id    INT             FOREIGN KEY REFERENCES tbl_user(user_id)
order_name    VARCHAR(45)
note    VARCHAR(45)
```

4)즉 한명의 user는 여러 order를 가질 수 있고 order는 fk로 user_id를 가지고 있다.
5)보통 query를 이용해서 이 table을 객체로 만들때는

```
public class User {
    private int userId;
    private String username;
    private String nickName;
    private String address;
}

public class Order {
    private int orderId;
    private String orderName;
    private String note;
    private int userId;
}
```

6)이런 식의 DTO 또는 VO class를 만드는데 User가 자신이 생성한 Order data를 가져오기 위해서는

```
SELECT * FROM tbl_user u JOIN tbl_order o ON u.user_id = o.user_id
WHERE u.user_id = #{value}
```

7)또는

```
SELECT * FROM tbl_order WHERE user_id = #{value}
```

8)이렇게 query가 작성 된다.
9)여기서 주의해서 봐야될 부분은 바로 Order class에 userId 변수다.
10)이 변수는 Order를 생성한 User의 id를 가지고 있다.
11)즉 table 관점으로 본다면 userId 변수는 fk가 되는 것이다.
12)User 객체가 아니라 User의 pk만 가지고 있는 것이다.
13)그럼, ORM에서는 객체를 어떻게 구성할까?

```
@Entity(name = "tbl_user")
public class User {
    @Id @GeneratedValue(strategy = GenerationType.AUTO)
    private Integer userId;
    private String username;
    private String nickName;
    private String address;

    @OneToMany(mappedBy = "user", cascade = CascadeType.ALL)
    private List<Order> orders;
}

@Entity(name = "tbl_order")
public class Order {
    @Id @GeneratedValue(strategy = GenerationType.AUTO)
    private Integer orderId;
    private String orderName;
    private String note;
    private int price;

    @ManyToOne(fetch = FetchType.LAZY, cascade = CascadeType.ALL)
    @JoinColumn(name = "user_id")
    private User user;
}
```

14)User 객체는 Order List 객체를, Order 객체는 User 객체를 가지고 있다.
15)Table 관점으로 보나, 객체 관점으로 보나 pk, fk도 만족을 하며, 객체지향적으로 객체를 만든 것이다.
16)심지어 data를 조작하기 위한 query 자체도 신경쓰지 않는다.
17)Query 자체를 신경쓰지 않는다는 뜻은 실제 User 객체나 Order 객체를 table에서 조회해서 만드는 과정에 있어 위 예제와 같이 query를 작성해서 만들지 않는다는 뜻이다.
18)내부적으로는 query가 실행되고 실제로 우리는 그 query가 어떻게 만들어지고 실행되는지 정확히 인지하고 있어야 정확한 JPA 사용이 가능하다.
19)우리가 query를 작성하지 않는다는 뜻이지 query가 실제로 실행 안된다는 뜻은 아니다.
20)이렇게 객체지향적인 객체가 생성되면 우린 business logic 자체를 query에 집중하지 않고 Java code 자체에 집중

할 수 있다는 것이다.
21)Code에 business logic을 집중할 수 있다는 것은 많은 장점이 있는데 유지보수, testing, debugging, 객체지향과 같은 여러 장점을 갖을 수 있다.
22)물론 xBatis나 JDBC, Spring JDBC Template과 같은 query를 작성하는 기술을 사용해도 객체지향적인 객체 설계가 가능하지만 좀 더 쉽고 빠르게 쓸 수 있는 것이 ORM이다.

10. 영속성 컨텍스트(Persistence Context)
1)EntityManager를 하나 만들때 마다 생성되며 조회, 저장 시에 EntityManager는 영속성 context에 entity를 보관하게 된다.
2)Entity의 생명주기
-비영속(new/transient)
--영속성 context와 전혀 관계없는 상태 순수한 객체 상태, 아직 DB와 전혀 상관없는 상태이다.
--entityManager.persist()호출 이전
-영속(managed)
--영속성 context에 저장된 상태 entityManagerpersist() 호출 이후의 상태.
--즉 영속 상태란 , 영속성 context에 의해 관리된다는 의미이다
-준영속(detached)
--영속성 context에 저장되었다가 분리된 상태.
--detach(), close()를 호출한 상태
--entityManager.clear()를 호출해서 영속성 context를 초기화 해도 해당 entity는 준영속 상태로 유지된다
-삭제(removed)
--삭제된 상태
--entityManager.remove(entity)와 같이 영속성 context와 DB에서 삭제된 상태.

3)영속성 context의 특정
-영속성 context에는 @Id로 식별자가 꼭 존재해야 한다.
-영속성 context에서 DB로의 저장은 commit(flush)시에 수행된다.
-영속어 conext를 사용하면 1차 cache, 동일성 보장, transaction을 지원하는 쓰기지연, 변경감지, 지연로딩 등의 장점이 존재한다.

4)Entity조회
-영속성 context 내부의 cache를 1차 cache라 부른다.
-내부에 Map형태의 정보를 저장하고 @Id로 mapping된 정보가 저장되어 있다.
-즉 1차 cache의 key는 식별자(@Id)값이다
-entityManager.find()를 호출하면 우선 1차 cache에서 검색한다.
-1차 cache에 없다면 DB를 검색하고 결과를 1차 cache에 저장 한다.
-이를 통해서 객채의 동일성을 보장 할 수 있다.

5)Entity 등록
-Entity Manager는 transaction을 commit하기 전까지 entity를 DB에 저장하지 않고 query 저장소에 INSERT SQL문을 차곡 차곡 쌓아 놓는다.
-그리고 commit시에 한번에 query를 보낸다.
-이것이 바로 transaction을 지원하는 쓰기 지연이다.
-이 기능을 잘 사용하면 모아둔 등록 query를 database에 한번에 전달해 성능을 최적화 할 수 있다.

6)Entity 수정
-Project가 커져가면서 수정 사항은 항상 늘어난다.
-SQL문으로 이러한 문제를 대처 하고 있다면 수정 사항이 생길때 마다 query를 추가 하거나 수정 해야 한다.
-JPA에서의 entity 수정은 entity manger에서 받아온 entity의 값을 변경해 줌으로써 가능 하다.
-변경사항의 DB적용은 commit 시에 저장 된다.
-수정 query문은 변경 사항만 가지고 만들어지지 않고 전체 field를 이용해서 만들어진다.
-이는 항상 동일한 query(값은 다르지만)가 만들어진다는 장점과 용량이 커질수 있다는 단점이 있다.
-용량문제가 critical하다면 hibernate의 DynamicUpdate 기능을 사용하는 방법도 있다(30개 이상의 column에서는 효과가 있다고 한다).

7)변경 감시
-Transaction을 commit하면 EntityManager 내부에서 flush()가 일어나고 entity와 snapshot을 비교해서 변경사항을 SQL query로 만들어 SQL 저장소에 보낸다.
-그리고 쓰기 지연 저장소의 SQL을 DB에 보내고 commit한다.
-변경 감시는 영속성 context가 관리하는 영속 상태의 entity에만 적용된다.

8)Entity 삭제
-삭제를 하려면 먼저 entity를 조회 해야 한다.
-조회된 entity를 넘겨주면 삭제가 된다.
-이때도 당연히 삭제 query는 쓰기지연 SQL 저장소에 저장 되고 commit 시에 수행 된다.

9)Flush
-영속성 context의 변경 내용을 DB에 반영하는 것을 말한다.
-변경 감지가 동작해서 영속성 context에 있는 모든 entity를 snapshot과 비교해서 수정된 정보를 찾고 query로 만든다.
-쓰기 지연 SQL 저장소의 query를 DB로 전송하다.
-Flush가 호출되는 시점은 크게 3가지가 있다.
--직접 호출
--Transaction commit 시 자동 호출
--JPQL query 실행시 자동 호출

10)Flush mode
-FlushModeType.AUTO (default)
--commit이나 commit 시에 수행됨
-FlushModeType.COMMIT
--commit시에만 수행

11. Entity Mapping
1)Mapping annotation 을 크게 4가지로 분류하면
-객체와 table mapping : @Entity, @Table
-기본 key mapping : @Id
-field와 column mapping : @column
-연관관계 mapping : @ManyToOne, @JoinColumn

2)@Entity
-class에 @Entity를 붙여주면 JPA가 Entity로서 관리 한다는 것을 의미한다.
-속성
--name
---다른 Entity와 충돌이 우려될 경우 이름을 바꿔준다.
---기본적으로는 Class명을 따른다.
-주의사항
--기본 생성자 필수
--final, enum, interface, inner-class 사용 못함
--저장 field에 final 사용 못함

3)@Table
-Entity와 mapping할 DB Table을 지정 한다.
-속성
--name
---mapping할 table 이름, 기본은 Entity 이름을 사용한다.
--catalog
---catalog 기능이 있는 DB에서 catalog를 매핑
--schema
---schema 기능이 있는 DB에서 schema를 매핑
--uniqueConstraints
---DDL 생성 시에 unique 제약조건을 만든다.

4) Database schema 자동 생성
-자동으로 schema를 생성하는 기능은 아래 값을 설정 함으로써 가능하다.

<property name="hibernate.hbm2ddl.auto" value="create-drop"/>

-value
--create
---기존 Table Drop + 생성
--create-drop
---create후 종료시 drop까지 실행
--update
---변경된 내용만 수정한다.
---이건 JPA spec에는 없고 hibernate에만 있는 설정 이다.
--validate
---기존 DB Table정보와 비교해서 차이가 있다면 경고하고 application을 실행 하지 않는다.
---이건 JPA spec에는 없고 hibernate에만 있는 설정 이다.
--none
---설정이 없거나 유효하지 않은 값을 설정하면 기능을 사용하지 않게된다.

5)기본 key mapping
-primary key를 설정하는 것을 말한다.

-직접할당
--em.persist()를 호출하기전에 사용자가 직접 ID를 설정하는 것을 말한다.

```
Board board = new Board();
board.setId("board1");
em.persist(board);
```

-자동생성
--IDENTITY
---기본 key의 생성을 DB에 위임하는것을 말한다.
---MySQL의 AUTO_INCREMENT와 같은것을 말한다.
---@GeneratedValue(strategy = GenerationType.IDENTITY) 로 설정 가능

--SEQUENCE
---유일한 값을 순서대로 생성하는 특별한 database object를 이용하는 방법을 말한다.
---@GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "BOARD_SEQ_GENERATOR") 로 설정 가능

```
@Entity
@SequenceGenerator(
   name = "BOARD_SEQ_GENERATOR",
   sequenceName = "BOARD_SEQ", // 실제 DB의 Sequence Name
   initialValue = 1, allocationSize = 1 )
public class Board {
   ...
   @Id
   @GeneratedValue(strategy = GenerationType.SEQUENCE, generator =
   "BOARD_SEQ_GENERATOR")
   private Long id;
   ...
}
```

--TABLE
---key 생성 전용 Table을 만들어서 이를 SEQUENCE 처럼 사용하는 것을 말한다.

```
@Entity
@TableGenerator(
   name = "BOARD_SEQ_GENERATOR",
   table = "MY_SEQUENCE", // 실제 DB의 Table name
   pkColumnValue = "BOARD_SEQ", allocationSize = 1 )
public class Board {
   ...
   @Id
   @GeneratedValue(strategy = GenerationType.TABLE, generator =
   "BOARD_SEQ_GENERATOR")
   private Long id;
   ...
}
```

--AUTO
---DB종류에 따라 JPA가 알맞은 것을 선택하는것을 의미한다.
---Oracle의 경우 SEQUENCE, MySQL의 경우 IDENTITY를 선택하게 된다.
---또하나의 장점은 DB종류가 바뀌어도 source를 수정하지 않아도 된다는 것이다.
---@GeneratedValue(strategy = GenerationType.AUTO) 로 설정 가능

6)field와 column mapping : reference
-field와 column mapping에 사용되는 annotation으로는 @Column, @Enumerated, @Temporal, @Lob, @Transient, @Access 가 있다.
-@Column
--객체 field를 table column과 mapping해주는 가장 대표적인 annotation이다.
--name, nullable이 가장 많이 사용되고 나머지는 많이 사용되지는 않는다.
--속성
---name
----mapping할 table column 이름, 기본은 객체의 field 이름을 사용한다.
---insertable
----거의 사용안됨, entity 저장시 이 field도 저장하라는 의미로 기본은 true이다.
----단 false로 하면 Readonly일때 사용가능 하다.

---updatable
----거의 사용안됨, entity 수정시 이 field도 수정하라는 의미로 기본은 true이다.
----단 false로 하면 Readonly일때 사용가능 하다.
---table
----거의 사용안됨, 하나의 entity를 두 개 이상의 table에 mapping할때 사용
---nullable
----false로 설정하면 DDL생성시에 "NOT NULL" 제약조건을 추가해 준다.
---unique
----@Table 의 uniqueConstraints와 같지만 한 column에 대해서 적용할때는 간단하게 이것을 이용가능
----단 여러 column을 사용할때는 @Table의 uniqueConstraints를 사용해야 한다.
---columnDefinition
----사용자가 직접 column의 정보를 입력해준다.
---length
----문자 길이에 대한 제약조건을 준다.
----String 타입에만 적용되며 기본값은 255이다.
---precision, scale
----BigDecimal 타입에서 사용된다.
----precision은 소수점을 포함한 전체 지릿수, scale은 소수 자리수를 의미.
----float, double에는 해당 되지 않는다.

-@Enumerated
--enum type을 mapping할 때 사용한다.
--속성
---name
----EnumType.ORDINAL
-----enum의 순서를 DB에 저장, 이값이 Default이다.
-----숫자로 저장되므로 data 크기가 작아지고 빠르다.
-----enum의 순서를 변경할 수 없는 단점이 있다.
----EnumType.STRING
-----enum이름을 DB에 저장
-----문자로 저장되므로 data 크기가 커지고 느리다. 하지만 enum의 순서와 상관없이 사용가능해 진다.
-----Default는 ORDINAL이지만 STRING을 더 추천 한다.

-@Temporal
--날짜 type을 mapping할 때 사용 속성을 입력 하지 않으면 Java의 Date과 가장 유사한 Timestamp로 저장된다(H2, Oracle, PostgreSQL).
--하지만 이는 DB의 종류에 따라 Datetime으로 저장되기도 한다(MySQL).
--속성
---value
----TemporalType.DATE
-----2013-01-23 와 같은 날짜 타입
----TemporalType.TIME
-----11:23:18 과 같은 시간 타입
----TemporalType.TIMESTAMP
-----2013-01-23 11:23:18 과 같이 DB의 Timestamp 타입과 매핑

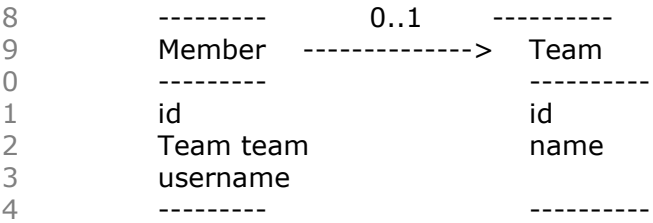
-@Lob
--@Lob는 별도의 속성은 없다.
--대신 mapping을 문자열이면 CLOB, 그외의 type에는 BLOB으로 매핑한다.
---CLOB : String, char[], java.sql.CLOB
---BLOB : byte[], java.sql.BLOB

-@Transient
--이 field는 매핑하지 말라는 의미이다.
--이는 임시로 중간 값을 저장하는 용도로 사용가능하다.

-@Access
--JPA가 entity data에 접근하는 방식
--@Access를 설정하지 않으면 @Id의 설정위치에 따라서 접근 방식이 결정 된다.
--@Id가 필드에 붙어 있으면 FIELD접근 방식을 의미하므로 Getter가 없어도 된다.
--@Id가 property에 있으면 PROPERTY 접근 방식을 의미하게 된다.
--하지만 이 두가지 방식을 섞어서 사용도 가능하다.
---AccessType.FIELD
----field 접근
----Private이어도 접근 가능하다.
---AccessType.PROPERTY
----property 접근

----접근자(Getter)를 이용한다.

12. 연관관계 mapping 기초
1)단방향 연관관계
-연관관계중에서는 다대일(N:1)을 가정 먼저 이해 하고 넘어가야 한다.
-회원과 팀의 관계가 그에 해당 한다.
-회원과 팀이 있다.
-회원은 하나의 팀에만 속할 수 있다.



-객체 연관관계
--Member 객체는 Member.team field로 Team객체와 연관 관계를 맺고 있고 이는 단방향 관계이다.
--즉 Member는 team 필드로 팀을 알수 있지만 Team은 Member를 알아 낼 수 없다.

-테이블 연관관계
--반면 table은 외래 key를 이용해서 양방향으로 조회 할 수 있다.
--즉, 외래 key를 이용해서 서로 JOIN해주면 양방향의 관계를 알아 낼 수 있다.

-객체 관계 매핑

```
@Entity
public class Member {
    @Id
    @Column(name = "MEMBER_ID")
    private String id;

    private String username;

    @ManyToOne
    @JoinColumn(name = "TEAM_ID")
    private Team team;

    // Getter, Setter...
}

@Entity
public class Team {
    @Id
    @Column(name = "TEAM_ID")
    private String id;

    private String name;

    // Getter, Setter...
}
```

-@ManyToOne
--말 그대로 N:1의 관계를 라는 mapping 정보 이다.
--속성
---optional
----기본은 true인데 false로 설정하면 연관된 Entity가 꼭 있어야 한다.
---fetch
----FetchType.EAGER
----FetchType.LAZY
---cascade
----영속성 전이 기능을 사용
---targetEntity
----연관된 entity의 type 정보를 설정

-@OneToMany

@OneToMany
private List<Member> members // generic으로 type 정보를 알수 있다.

@OneToMany(targetEntity=Member.class)
private List member // generic이 없으면 type 정보를 알수 없다.

-@JoinColumn(name="TEAM_ID")
--JoinColumn 은 외래 key를 mapping할 때 사용한다.
--속성
---name
----mapping할 외래 key 이름
---referencedColumnName
----외래 key가 참조하는 대상 table의 column 이름
---foreignKey(DDL)
----외래 key 제약조건을 직접 설정하기위한 속성이다.
---unique
---nullable
---insertable
---updatable
---columnDefinition
---table
----@Column속성과 같다.

-연관관계 사용
--저장
---JPA에서 entity를 저장할 때 연관된 모든 entity는 영속 상태여야 한다.

```
Team team1 = new Team("team1", "팀1");
em.persist(team1);

Member member1 = new Member("member1", "회원1");
member1.setTeam(team1);
em.persist(member1);

Member member2 = new Member("member2", "회원2");
member2.setTeam(team1);
em.persist(member2);
```

--조회
---연관관계가 있는 entity를 조회 하는 방법은 "객체 graph 탐색" 방법과 "객체지향 query 사용" 이 있다.
----객체 graph 탐색

```
Member member = em.find(Member.class, "member1");
Team team = member.getTeam();   // <---- 이것이 객체 그래프 탐색이다.
```

----객체지향 query 사용
-----객체지향 query인 JPQL을 이용한 방법이다.

```
String jpql = "select m from Member m join m.team t where t.name = :teamName";

List<Member> resultList = em.createQuery(jpql, Member.class)
                            .setParmeter("teamName", "팀1")
                            .getResultList();
```

-수정
--em.update()와 같은 함수는 없다.
--단순히 아래와 같이 처리하면 된다.

```
Team team2 = new Team("team2", "팀2");
em.persist(team2);

Member member = em.fine(Member.class, "member1");
member.setTeam(team2);
```

-연관 관계 제거
--아래와 같이 null로 처리해 주면 관계는 삭제 된다.

```
Member member = em.fine(Member.class, "member1");
member.setTeam(null);
```

-연관된 entity 삭제
--연관된 entity를 삭제 하려면 해당 entity를 사용하는 모든 entityt의 연관 관계를 제거 해줘야 한다.

```
member1.setTeam(null);
member2.setTeam(null);
em.remove(team);
```

2)양방향 연관관계
-단방향에서는 "회원" -> "팀" 으로만 확인이 가능했다면 양방향에서는 "팀" -> "회원" 으로도 확인이 가능해야 한다.
-RDB의 Table에서의 관계는 양방향과 단방향이 모두 동일하다.(FK로 서로 교차 검색이 가능하다)

```
@Entity
public class Team {
    @Id
    @Column(name = "TEAM_ID")
    private String id;

    private String name;

    // 추가
    @OneToMany(mappedBy = "team")
    private List<Member> members = new ArrayList<Member>();

    // Getter, Setter...
}
```

-mappedBy는 양방향 매핑을 할때 반대쪽에 매핑할 필드의 이름이다.
-연관관계의 주인
--mappedBy는 왜? 필요할까??
---객체에서 양방향 관계라는 것은 없다. -> 2개의 단방향 관계가 있는 것이다.
---mappedBy는 연관관계의 주인을 정해주는 작업이다.
---연관관계의 주인은 외래 key가 존재하는 entity가 된다.
---즉, 연관관계 주인 만이 연관관계를 갱신할 수 있고, 반대편(inverse, non-owning side)는 읽기만 가능하다.

-양방향 연관관계 저장 및 주의할 점
--예제에서는 Member.team 연관관계의 주인이므로 아래 코드의 내용에 주의 해줘야한다.

```
// 정상적인 연관관계의 설정은 Member 에서 이루어져야 한다.
Team team1 = new Team("team1", "팀1");
em.persist(team1);

Member member2 = new Member("member2", "회원2");
member.setTeam(team1);
em.persist(member2);

// Team에서 연관관계를 업데이트 하려고 시도하는것은 무시된다.
Member member2 = new Member("member2", "회원2");
em.persist(member2);

Team team1 = new Team("team1", "팀1");
team1.getMembers().add(member2);
em.persist(team1);
```

--하지만 순수 객체로 따져 보면 위 예제의 정상과 비정상 예제의 내용이 모두 실행되어야 정상적으로 동작하게 된다.
--그래서 Member의 setTeam을 아래와 같이 수정해 주면 순수 객체에서도 정상동작 하게 된다.

```
public void setTeam(Team team){
    // 기존팀 관계 제거
    //   -> 영속성 컨텍스트가 새로 시작되면 문제가 없지만 영속성 컨텍스트를 계속 사용중이라면
    //      문제가 발생 할 수 있다.
```

```
            if (this.team != null){
               this.team.getMembers().remove(this);
            }
            this.team = team;
            team.getMembers().add(this);
         }
```

13. 다양한 연관관계 mappinig

1)다대일

-다대일 단방향 [N:1]

```
---------  *          1     ----------
Member    ------------->      Team
---------                    ----------
id                           id
Team team                    name
username
---------                    ----------
```

--Member에서는 @MenyToOne 으로 Team을 참조 할 수 있도록 해주고 Team에서는 별도의 설정을 하지 않는다.

```
<Member>
@Entity
public class Member {
   @Id @GeneratedValue
   @Column(name = "MEMBER_ID")
   private Long id;

   private String username;

   @ManyToOne
   @JoinColumn(name = "TEAM_ID")
   private Team team;
}

<Team>
@Entity
public class Team {
   @Id @GeneratedValue
   @Column(name = "TEAM_ID")
   private Long id;

   private String name;
}
```

-다대일 양방향 [N:1, 1:N]

```
---------          team          ----------
Member       -------------->      Team
---------       members     1    ----------
id              <--------------      id
Team team  *                     List members
username                             name
---------                            ----------
```

--Member에서는 @MenyToOne 으로 Team을 참조 할 수 있도록 해주고 Team에서는 @OneToMany 와 함께 mappedBy값을 설정 해서 연관관계의 주인이 Member table임을 알려준다.

--항상 1:N, N:1 관계에서는 항상 N쪽에 외래 key가 있다.

--여기에서는 N쪽인 Member table이 외래 key를 가지게 된다.

--그러므로 Member.team이 연관관계의 주인이 된다.

```
<Member>
@Entity
public class Member {
   @Id @GeneratedValue
   @Column(name = "MEMBER_ID")
```

```
    private Long id;

    private String username;

    @ManyToOne
    @JoinColumn(name = "TEAM_ID")
    private Team team;

    public void setTeam(Team team){
        this.team = team;

        if(!team.getMembers().contains(this)){
            team.getMembers.().add(this);
        }
    }
}

<Team>
@Entity
public class Team {
    @Id @GeneratedValue
    @Column(name = "TEAM_ID")
    private Long id;

    @OneToMany(meppedBy = "team")
    private List<Member> members

    private String name;

    public void addMember(Member member){
        this.members.add(member);
        if(member.getTeam() != this){
            member.setTeam(this);
        }
    }
}
```

2)일대다

-일대다 단방향 [N:1]

```
--------- 1              * ----------
Team      -------------->     Member
---------      members     ----------
id                         id
name                       username
List members
---------                     ----------
```

--보통은 자신이 mapping한 table의 외래 key를 가지는데 반해 여기서는 Member table에 외래 key가 있다.
--일대다 관계에서는 꼭 JoinColumn을 선언 해줘야 한다.
--안그러면 join table을 중간에 두는 방식으로 설정 된다.
--일대다의 단점은 mapping한 객체가 관리하는 외래 key가 다른 table에 있으므로 Insert SQL한번으로 조작이 끝나는것이 아닌 연관 테이블의 Update도 해줘야하는 문제가 있다.
--그러므로, 일대다 보다는 다대일 단방향 mapping을 사용하자.

```
<Team>
@Entity
public class Team {
    @Id @GeneratedValue
    @Column(name = "TEAM_ID")
    private Long id;

    @OneToMany
    @JoinColumn(name = "TEAM_ID") // Member table의 TEAM_ID 를 나타냄
    private List<Member> members;
```

```
    private String name;
  }

  <Member>
  @Entity
  public class Member {
    @Id @GeneratedValue
    @Column(name = "MEMBER_ID")
    private Long id;

    private String username;
  }
```

-일대다 단방향 [N:1]
--일대다 양방향 mapping은 존재 하지 않는다.
--대시 다대일 mapping을 사용해야 한다.
--다시 말해 양방향 mapping에서는 @OneToMany는 연관관계의 주인이 될 수 없다는 것이다.

3)일대일
-양쪽이 서로 하나의 관계만을 가진다.
-즉, 일대일 관계는 그 반대도 일대일 관계다.
-주 table에 외래 key
--JPA에서는 주 table에 외래 key가 있는것이 좀더 편리하게 mapping할 수 있게된다.
-단방향
--Member 와 Locker의 관계로 알아보자.
--아래 모습은 다대일(N:1) 단방향 모습과 매우 흡사하다.

```
------------  1          1    ----------
Member      -------------->     Locker
------------                   ----------
id                             id
Locker locker                     name
username
------------                   ----------

@Entity
public class Member {
  @Id @GeneratedValue
  @Column(name = "MEMBER_ID")
  private Long id;

  private String username;

  @OneToOne
  @JoinColumn(name = "LOCKER_ID")
  private Locker locker;
}

@Entity
public class Locker {
  @Id @GeneratedValue
  @Column(name = "LOCKER_ID")
  private Long id;

  private String name;
}
```

-양방향
--양방향의 주인이 정해 졌다.
--Member table의 외래 key를 가지고 있으며 Member.locker가 연관관계의 주인이다.
--Locker는 mappedBy를 이용해서 연관관계의 주인을 선언한다.

```
------------        locker           ----------
Member      -------------->          Locker
------------                1    ----------
id            1                     id
```

```
Locker locker     <- - - - - - --        name
username            member        Member member
------------                      ----------

   @Entity
   public class Member {
      @Id @GeneratedValue
      @Column(name = "MEMBER_ID")
      private Long id;

      private String username;

      @OneToOne
      @JoinColumn(name = "LOCKER_ID")
      private Locker locker;
   }

   @Entity
   public class Locker {
      @Id @GeneratedValue
      @Column(name = "LOCKER_ID")
      private Long id;

      private String name;

      @OneToOne(mappedBy = "locker")
      private Member member;
   }

-대상 table에 외래 key
   --JPA에서는 주 table에 외래 key가 있는것이 좀더 편리하게 매핑할 수 있게된다.
   --단방향
      ---이는 JPA에서 지원 하지 않는다.
   --양방향
      ---대상 table인 Locker에서 외래 key를 관리하는 모습이다.

      ------------          locker         ----------
      Member       - - - - - - - >        Locker
      ------------                      1 ----------
      id          1                       id
      Locker  locker   <------------       name
      username        member               Member member
      ------------                         ----------

      @Entity
      public class Member {
         @Id @GeneratedValue
         @Column(name = "MEMBER_ID")
         private Long id;

         private String username;

         @OneToOne(mappedBy = "member")
         private Locker locker;
      }

      @Entity
      public class Locker {
         @Id @GeneratedValue
         @Column(name = "LOCKER_ID")
         private Long id;

         private String name;

         @OneToOne
         @JoinColumn(name = "MEMBER_ID")
         private Member member;
```

```
        }


4)다대다
  -다대다는 한쪽에 외래 key를 두고 2개의 table로 관계를 표현 할 수가 없어서 중간에 연결 table을 추가 하게 된다.
  -하지만 객체의 관계는 2개의 객체로 표현이 가능하다.
  -다대다 관계인 "Member"와 "Product"로 살펴보자.
  -단방향
    --Member에서 Product를 @ManyToMany로 연결한다.
    --중요한 점은 @JoinTable을 이용해서 연결 table을 설정해 주는 것이다.
    --MEMBER_PRODUCT table은 다대다 관계를 "일대다, 다대일" 관계로 풀어내기 위한 연결 table이다.

      @Entity
      public class Member {
        @Id @GeneratedValue
        @Column(name = "MEMBER_ID")
        private Long id;

        private String username;

        @ManyToMany
        @JoinTable(name = MEMBER_PRODUCT,
        joinColumns = @JoinColumn(name = "MEMBER_ID"),
        inverseJoinColumns = @JoinColumn(name = "PRODUCT_ID")
        private List<Product> products = new ArrayList<Product>();
      }

      @Entity
      public class Product {
        @Id @GeneratedValue
        @Column(name = "PRODUCT_ID")
        private Long id;

        private String name;
      }

      public void save(){
        Product productA = new Product();
        productA.setId("ProductA");
        productA.setName("상품A");
        em.persist(productA);

        Member member1 = new Member();
        member1.setId("member1");
        member1.setUsername("회원1");
        member1.getProducts().add(ProductA); // 연관관계 설정
        em.persist(member1);
      }

      public void find(){
        Member member = em.find(Member.class, "member1");
        List<Product> products = member.getProducts();
        for(Product product : products){
          System.out.println("product.name = " + product.getName());
        }
      }

  -양방향
    --단방향과 마찬가지로 양방향은 반대쪽도 @ManyToMany를 사용한다.
    --그리고 양방향중 한곳에 mappedBy를 설정해서 연관관계 주인을 지정해 준다.
    --mappedBy가 없는 쪽이 연관관계의 주인이다.

    @Entity
    public class Product {
      @Id
      private String id;
```

```
    @ManyToMany(mappedBy = "products")
    private List<Member> members;

    ...
  }
```

14. Lab : JavaSE 환경에서 JPA 설정 및 CRUD

1)Project 구조

-DB는 편의상 h2 DB 사용

-tbl_user, tbl_order table을 예제로 사용

2)SQL

```
CREATE TABLE tbl_user
(
    user_id   INT,
    username     VARCHAR(20),
    nick_name   VARCHAR(20),
    address      VARCHAR(100),
    CONSTRAINT tbl_user_user_id_pk  PRIMARY KEY(user_id)
);

CREATE TABLE tbl_order
(
    order_id    INT,
    user_id     INT,
    order_name    VARCHAR(45),
    note     VARCHAR(100),
    price INT,
    CONSTRAINT tbl_order_order_id_pk  PRIMARY KEY(order_id),
    CONSTRAINT tbl_order_user_id_fk  FOREIGN KEY(user_id)
    REFERENCES tbl_user(user_id)
);
```

3)Eclipse, JPA J2SE Project

-Package Explorer > right-click > New > Other > JPA > JPA Project > Next

-Project name : JPADemo

-Target runtime : jdk1.8.0_162

-JPA version : 2.1

-Configuration : Default Configuration for jdk1.8.0_162

-Next > Next

-Platform :Generic 2.1

-Type : User Library > Download library...

-Download Library : EclipseLink 2.5.2 > Next > Check I accpet... > Finish

-Connection : Add connection... > Generic JDBC

-Name : H2 JDBC > Next > New Driver Definition

-Name/Type tab > Select Generic JDBC Driver > JAR List tab > Add JAR/Zip > C:\Program Files (x86)\H2\bin\h2-1.4.197.jar

-Properties tab >

--Connection URL :jdbc:h2:tcp://localhost/~/test

--Database Name : test

--Driver Class : Click ... > Select [Browse for class] > org.h2.Driver

--User ID as sa

-Test Connection > Finish

-Finish

-Open Perspective

4)Data Source Explorer view에 H2 database 등록할 경우

-https://ibytecode.com/blog/eclipse-dtp-configure-h2-datasource-using-data-source-explorer/

-Right click on Database Connections in Data Source Explorer -> New

-From the main menu, select File -> New -> Other. Under Connection Profiles, select Connection Profile and click Next.

-Select Generic JDBC from Connection Profile Types

--Name : H2 JDBC

--Click Next.

-Now select an existing JDBC Driver and provide Connection Details in the “New Connection Profile” dialog.

-If this is the first installation of H2 Database Profile in your Eclipse workspace, you have to create a new driver definition by providing the location of the driver JAR and connection properties.
--Click on the New Driver Definition icon next to the Drivers combo box.
--Select driver version under Name/Type tab.
--Select [Generic JDBC Driver]
--Click on JAR list tab and add the location of the H2 database JAR file.
C:\Program Files (x86)\H2\bin\h2-1.4.197.jar
--Click on Properties tab and enter the properties and click on OK.
--Connection URL :jdbc:h2:tcp://localhost/~/test
--Database Name : test
--Driver Class : Click ... > Select [Browse for class] > org.h2.Driver
--User ID as sa
(empty password) and click on OK.
--Click [Test Connection]
-Click Finish to create the connection profile.

5)H2Database Driver를 project build path에 등록

6)Entity(Domain)
-JPA에서 Entity는 하나의 테이블 객체를 표현한 것이라고 생각해 된다.
-@Entity가 테이블 정보이며 변수가 필드가 되는 것이다.

-src/com.javasoft.entity package
-com.javasoft.entity > right-click > New > Class
-Class name : User > Finish
-User.java

```
package com.javasoft.entity;

import java.io.Serializable;
import java.util.List;

import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.OneToMany;
import javax.persistence.Table;

@Entity
@Table(name = "tbl_user")
public class User implements Serializable {
   @Id @GeneratedValue(strategy = GenerationType.AUTO)
   private Integer user_id;
   private String username;
   private String nick_name;
   private String address;

   @OneToMany(mappedBy = "user", cascade = CascadeType.ALL)
   private List<Order> orders;

   public User() {}

   public User(Integer user_id, String username, String nick_name, String address) {
      this.user_id = user_id;
      this.username = username;
      this.nick_name = nick_name;
      this.address = address;
   }

   public Integer getUser_id() {
      return user_id;
   }

   public void setUser_id(Integer user_id) {
```

```
3631            this.user_id = user_id;
3632        }
3633
3634        public String getUsername() {
3635            return username;
3636        }
3637
3638        public void setUsername(String username) {
3639            this.username = username;
3640        }
3641
3642        public String getNick_name() {
3643            return nick_name;
3644        }
3645
3646        public void setNick_name(String nick_name) {
3647            this.nick_name = nick_name;
3648        }
3649
3650        public String getAddress() {
3651            return address;
3652        }
3653
3654        public void setAddress(String address) {
3655            this.address = address;
3656        }
3657
3658        public List<Order> getOrders() {
3659            return orders;
3660        }
3661
3662        public void setOrders(List<Order> orders) {
3663            this.orders = orders;
3664        }
3665
3666        public int totalPrice() {
3667            int totalPrice = 0;
3668            for (Order order : orders) {
3669                totalPrice += order.getPrice();
3670            }
3671            return totalPrice;
3672        }
3673
3674        @Override
3675        public int hashCode() {
3676            // TODO Auto-generated method stub
3677            return super.hashCode();
3678        }
3679
3680        @Override
3681        public boolean equals(Object obj) {
3682            // TODO Auto-generated method stub
3683            return super.equals(obj);
3684        }
3685    }
3686
3687
3688
3689    -com.javasoft.entity.Order.java
3690
3691        package com.javasoft.entity;
3692
3693        import javax.persistence.CascadeType;
3694        import javax.persistence.Entity;
3695        import javax.persistence.FetchType;
3696        import javax.persistence.GeneratedValue;
3697        import javax.persistence.GenerationType;
```

```
import javax.persistence.Id;
import javax.persistence.IdClass;
import javax.persistence.JoinColumn;
import javax.persistence.ManyToOne;
import javax.persistence.Table;

@Entity
@Table(name = "tbl_order")
public class Order {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Integer order_id;
    private String order_name;
    private String note;
    private int price;

    @ManyToOne(fetch = FetchType.LAZY, cascade = CascadeType.ALL)
    @JoinColumn(name = "user_id")
    private User user;

    public Order() {}

    public Order(String order_name, String note, int price, User user) {
        this.order_name = order_name;
        this.note = note;
        this.price = price;
        this.user = user;
    }

    public Integer getOrder_id() {
        return order_id;
    }

    public void setOrder_id(Integer order_id) {
        this.order_id = order_id;
    }

    public String getOrder_name() {
        return order_name;
    }

    public void setOrder_name(String order_name) {
        this.order_name = order_name;
    }

    public String getNote() {
        return note;
    }

    public void setNote(String note) {
        this.note = note;
    }

    public int getPrice() {
        return price;
    }

    public void setPrice(int price) {
        this.price = price;
    }

    @Override
    public String toString() {
        return "Order{" + "orderId=" + order_id + ", orderName='" + order_name + '\'' + ",
        note='" + note + '}' + "\n";
    }
}
```

7)src/META-INF/persistence.xml
-hibernate.connection.driver_class : DB Driver
-hibernate.connection.url : DB url 및 DB파일이 저장될 경로(h2 DB에 한함)
-hibernate.connection.user : username
-hibernate.show_sql : JPA 내부적으로 사용되는 쿼리를 log로 나타낼지 설정
-hibernate.hbm2ddl.auto : Entity에 의한 테이블 설정 (create-drop은 프로젝트 실행시 기존 테이블을 삭제하고 다시 생성한다. 즉 테스트하기 위한 초기화)
-구현체와 DB 종류별 설정 참고(https://gist.github.com/mortezaadi/8619433)

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1"
   xmlns="http://xmlns.jcp.org/xml/ns/persistence"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
   http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">

   <persistence-unit name="JPADemo" transaction-type="RESOURCE_LOCAL">
      <class>com.javasoft.entity.User</class>
      <class>com.javasoft.entity.Order</class>

      <properties>
         <property name="javax.persistence.jdbc.url"
         value="jdbc:h2:tcp://localhost/~/test" />
         <property name="javax.persistence.jdbc.driver"
         value="org.h2.Driver" />
         <property name="javax.persistence.jdbc.user" value="sa" />
         <!-- <property name="javax.persistence.jdbc.password" value=""/> -->
         <property name="eclipselink.logging.level" value="FINE" />
         <property name="eclipselink.ddl-generation" value="create-tables" />
      </properties>
   </persistence-unit>
</persistence>
```

8)JPACRUDTest.java
-com.javasoft.service package 생성
-com.javasoft.service > right-click > New > Other > Java > JUnit > JUnit Test Case
-Select New JUnit 4 test
-Name : JPACRUDTest > Next >
-Select Perform the following action : Add JUnit 4 library to the build path > OK

-EntityManager : Entity의 Lifecycle과 persistence context, transaction을 관리한다.
-즉 insert, update, delete, select를 할 수 있다.
--select : find(Class, Object);
--insert : persist(Object);
--update : merge(T);
--delete : remove(Object);
-EntityManagerFactory : EntityManager를 생성하기 위한 클래스이며 persistence.xml 설정에 기반한다.

```
package com.javasoft.test;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertNull;

import java.util.ArrayList;
import java.util.List;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

import org.junit.After;
import org.junit.Before;
import org.junit.Test;

import com.javasoft.entity.Order;
```

```
import com.javasoft.entity.User;

public class JPACRUDTest {
   private EntityManager entityManager;
   private EntityManagerFactory entityManagerFactory;
   private User user;

   @Before
   public void setUp() throws Exception {
      entityManagerFactory = Persistence.createEntityManagerFactory("JPADemo");
      entityManager = entityManagerFactory.createEntityManager();
      entityManager.getTransaction().begin();

      // fixture
      user = new User();
      user.setUsername("한지민");
      user.setNick_name("jimin");
      user.setAddress("서울");
      List<Order> orders = new ArrayList<>();
      for (int i = 0; i < 10; i++) {
         Order order = new Order("order" + i, "note" + i, i + 10,  user);
         entityManager.persist(order);
         orders.add(order);
      }
      user.setOrders(orders);
      entityManager.persist(user);
      System.out.println("============ fixture ==========\n" + user);
   }

   @After
   public void after() {
      entityManager.getTransaction().commit();
      entityManager.close();
      entityManagerFactory.close();
   }

   @Test
   public void select() {
      User findUser = entityManager.find(User.class, user.getUser_id());
      assertEquals(user.getUser_id(), findUser.getUser_id());
      assertEquals(user.getUsername(), findUser.getUsername());
      assertEquals(user.getNick_name(), findUser.getNick_name());
      assertEquals(user.getAddress(), findUser.getAddress());
      assertEquals(user.totalPrice(), 145);

      // order
      assertEquals(user.getOrders().size(), 10);
   }

   @Test
   public void update() {
      // update
      User updateUser = entityManager.find(User.class, user.getUser_id());
      updateUser.setNick_name("update nickName");
      updateUser.setAddress("update address");

      entityManager.merge(updateUser);

      // persistence Context Test
      assertEquals("update nickName", user.getNick_name());
      assertEquals("update address", user.getAddress());

      // update Tests
      assertEquals("update nickName", updateUser.getNick_name());
      assertEquals("update address", updateUser.getAddress());
   }

```

```
    @Test
    public void delete() {
       User getUser = entityManager.find(User.class, user.getUser_id());
       entityManager.remove(getUser);
       User deleteUser = entityManager.find(User.class, user.getUser_id());
       assertNull(deleteUser);
    }
  }
```

15. Lab : OneToOne Demo

1)Project 구조

-DB는 편의상 h2 DB 사용

-person, cellular table을 예제로 사용

2)SQL

```
CREATE TABLE Person
(
   id     INT,
   cellular_id    INT,
   name   VARCHAR(45),
   CONSTRAINT person_id_pk  PRIMARY KEY(id)
);

CREATE TABLE Cellular
(
   id    INT,
   tel_number   CHAR(13),
   CONSTRAINT  cellular_id_pk  PRIMARY KEY(id)
);
```

3)Eclipse, JPA J2SE Project

-Package Explorer > right-click > New > Other > JPA > JPA Project > Next

-Project name : OneToOneDemo

-Target runtime : jdk1.8.0_162

-JPA version : 2.1

-Configuration : Default Configuration for jdk1.8.0_162

-Next > Next

-Platform :Generic 2.1

-Type : User Library > Download library...

-Download Library : EclipseLink 2.5.2 > Next > Check I accpet... > Finish

-Connection : Add connection... > Generic JDBC

-Name : H2 JDBC > Next > New Driver Definition

-Name/Type tab > Select Generic JDBC Driver > JAR List tab > Add JAR/Zip > C:\Program Files (x86)\H2\bin\h2-1.4.197.jar

-Properties tab >

--Connection URL :jdbc:h2:tcp://localhost/~/test

--Database Name : test

--Driver Class : Click ... > Select [Browse for class] > org.h2.Driver

--User ID as sa

-Test Connection > Finish

-Finish

-Open Perspective

4)Data Source Explorer view에 H2 database 등록할 경우

-https://ibytecode.com/blog/eclipse-dtp-configure-h2-datasource-using-data-source-explorer/

-Right click on Database Connections in Data Source Explorer -> New

-From the main menu, select File -> New -> Other. Under Connection Profiles, select Connection Profile and click Next.

-Select Generic JDBC from Connection Profile Types

--Name : H2 JDBC

--Click Next.

-Now select an existing JDBC Driver and provide Connection Details in the “New Connection Profile” dialog.

-If this is the first installation of H2 Database Profile in your Eclipse workspace, you have to create a new driver definition by providing the location of the driver JAR and connection properties.

--Click on the New Driver Definition icon next to the Drivers combo box.
--Select driver version under Name/Type tab.
--Select [Generic JDBC Driver]
--Click on JAR list tab and add the location of the H2 database JAR file.
C:\Program Files (x86)\H2\bin\h2-1.4.197.jar
--Click on Properties tab and enter the properties and click on OK.
--Connection URL :jdbc:h2:tcp://localhost/~/test
--Database Name : test
--Driver Class : Click ... > Select [Browse for class] > org.h2.Driver
--User ID as sa
(empty password) and click on OK.
--Click [Test Connection]
-Click Finish to create the connection profile.

5)H2Database Driver를 project build path에 등록

6)Entity(Domain)
-src/com.javasoft.entity package
-com.javasoft.entity > right-click > New > Class
-Class name : Person > Finish
-Person.java

```
package com.javasoft.entity;

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

@Entity
public class Person {
   @Id
   @GeneratedValue
   private int id;

   private String name;

   @OneToOne
   @JoinColumn(name = "cellular_id")
   private Cellular cellular;

   public Person() {}

   public Person(String name, Cellular cellular) {
      this.name = name;
      this.cellular = cellular;
   }

   public int getId() {
      return id;
   }

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

   public String getName() {
      return name;
   }

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

   public Cellular getCellular() {
      return cellular;
```

```
    }

    public void setCellular(Cellular cellular) {
        this.cellular = cellular;
    }

    @Override
    public String toString() {
        return "Person{" +
            "id=" + id +
            ", name='" + name + '\'' +
            ", cellular=" + cellular +
            '}';
    }
}


-com.javasoft.entity.Cellular.java

    package com.javasoft.entity;

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

    @Entity
    public class Cellular {
        @Id
        @GeneratedValue
        private int id;

        @Column(name="tel_number")
        private String number;

        public Cellular() {}

        public Cellular(String number) {
            this.number = number;
        }

        public int getId() {
            return id;
        }

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

        public String getNumber() {
            return number;
        }

        public void setNumber(String number) {
            this.number = number;
        }

        @Override
        public String toString() {
            return "Cellular{" +
                "id=" + id +
                ", number=" + number +
                '}';
        }
    }
```

7)src/META-INF/persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1" xmlns="http://xmlns.jcp.org/xml/ns/persistence" xmlns:xsi="
http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="
http://xmlns.jcp.org/xml/ns/persistence
http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
   <persistence-unit name="OneToOneDemo" transaction-type="RESOURCE_LOCAL">

      <class>com.javasoft.entity.Person</class>
      <class>com.javasoft.entity.Cellular</class>
      <properties>
         <property name="javax.persistence.jdbc.url"
            value="jdbc:h2:tcp://localhost/~/test" />
         <property name="javax.persistence.jdbc.driver"
            value="org.h2.Driver" />
         <property name="javax.persistence.jdbc.user" value="sa" />
         <!-- <property name="javax.persistence.jdbc.password" value=""/> -->
         <property name="eclipselink.logging.level" value="FINE" />
         <property name="eclipselink.ddl-generation" value="create-tables" />
      </properties>
   </persistence-unit>
</persistence>
```

8)OneToOneTest.java

-com.javasoft.service package 생성
-com.javasoft.service > right-click > New > Other > Java > JUnit > JUnit Test Case
-Select New JUnit 4 test
-Name : OneToOneTest > Next >
-Select Perform the following action : Add JUnit 4 library to the build path > OK

```
package com.javasoft.test;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

import org.junit.After;
import org.junit.Assert;
import org.junit.Before;
import org.junit.Test;

import com.javasoft.entity.Cellular;
import com.javasoft.entity.Person;

public class OneToOneTest {
   private EntityManagerFactory entityManagerFactory;
   private EntityManager entityManager;

   @Test
   public void oneToOneTest() {
      Cellular cellular = new Cellular();
      cellular.setNumber("010-1234-5678");
      entityManager.persist(cellular);

      Person person = new Person();
      person.setName("한지민");
      person.setCellular(cellular);
      entityManager.persist(person);

      Assert.assertEquals(person.getCellular().getId(), cellular.getId());
   }

   @Before
   public void setUp() throws Exception {
```

```
            entityManagerFactory = Persistence.createEntityManagerFactory("OneToOneDemo");
            entityManager = entityManagerFactory.createEntityManager();
            entityManager.getTransaction().begin();
        }

        @After
        public void after() {
            entityManager.getTransaction().commit();
            entityManager.close();
        }
    }
```

16. Lab : ManyToOne Demo

1)Project 구조

-DB는 편의상 h2 DB 사용

-tbl_User, tbl_Order table을 예제로 사용

2)SQL

```
CREATE TABLE tbl_user
(
    user_id   INT,
    username     VARCHAR(20),
    nick_name   VARCHAR(20),
    address      VARCHAR(100),
    CONSTRAINT tbl_user_user_id_pk  PRIMARY KEY(user_id)
);

CREATE TABLE tbl_order
(
    order_id    INT,
    user_id     INT,
    order_name    VARCHAR(45),
    note     VARCHAR(100),
    price INT,
    CONSTRAINT tbl_order_order_id_pk  PRIMARY KEY(order_id),
    CONSTRAINT tbl_order_user_id_fk  FOREIGN KEY(user_id)
    REFERENCES tbl_user(user_id)
);
```

3)Eclipse, JPA J2SE Project

-Package Explorer > right-click > New > Other > JPA > JPA Project > Next

-Project name : ManyToOneDemo

-Target runtime : jdk1.8.0_162

-JPA version : 2.1

-Configuration : Default Configuration for jdk1.8.0_162

-Next > Next

-Platform :Generic 2.1

-Type : User Library > Download library...

-Download Library : EclipseLink 2.5.2 > Next > Check I accpet... > Finish

-Open Perspective

4)H2Database Driver를 project build path에 등록

5)Entity(Domain)

-src/com.javasoft.entity package

-com.javasoft.entity > right-click > New > Class

-Class name : User > Finish

-User.java

```
package com.javasoft.entity;

import java.util.ArrayList;
import java.util.List;

import javax.persistence.CascadeType;
```

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

@Entity
@Table(name = "tbl_user")
public class User {
    @Id @GeneratedValue(strategy = GenerationType.AUTO)
    private Integer user_id;
    private String username;
    private String nick_name;
    private String address;

    @OneToMany(mappedBy = "user", cascade = CascadeType.ALL)
    private List<Order> orders;

    public User() {}

    public User(Integer user_id, String username, String nick_name, String address) {
        this.user_id = user_id;
        this.username = username;
        this.nick_name = nick_name;
        this.address = address;
    }

    public Integer getUser_id() {
        return user_id;
    }

    public void setUser_id(Integer user_id) {
        this.user_id = user_id;
    }

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getNick_name() {
        return nick_name;
    }

    public void setNick_name(String nick_name) {
        this.nick_name = nick_name;
    }

    public String getAddress() {
        return address;
    }

    public void setAddress(String address) {
        this.address = address;
    }

    public List<Order> getOrders() {
        return orders;
    }

    public void setOrders(List<Order> orders) {
        this.orders = orders;
    }
```

```
4291
4292            public boolean addOrder(Order order) {
4293                if(orders == null)
4294                    orders = new ArrayList<>();
4295
4296                return this.orders.add(order);
4297            }
4298
4299            @Override
4300            public String toString() {
4301                return "User{" +
4302                    "userId=" + this.user_id +
4303                    ", username='" + this.username + '\'' +
4304                    ", nickName='" + this.nick_name + '\'' +
4305                    ", address='" + this.address + '\'' +
4306                    ", orders=" + orders +
4307                    '}';
4308            }
4309        }
4310
4311
4312    -com.javasoft.entity.Order.java
4313
4314        package com.javasoft.entity;
4315
4316        import javax.persistence.CascadeType;
4317        import javax.persistence.Entity;
4318        import javax.persistence.FetchType;
4319        import javax.persistence.GeneratedValue;
4320        import javax.persistence.GenerationType;
4321        import javax.persistence.Id;
4322        import javax.persistence.IdClass;
4323        import javax.persistence.JoinColumn;
4324        import javax.persistence.ManyToOne;
4325        import javax.persistence.Table;
4326
4327        @Entity
4328        @Table(name = "tbl_order")
4329        public class Order {
4330            @Id
4331            @GeneratedValue(strategy = GenerationType.AUTO)
4332            private Integer order_id;
4333            private String order_name;
4334            private String note;
4335            private int price;
4336
4337            @ManyToOne(fetch = FetchType.LAZY, cascade = CascadeType.ALL)
4338            @JoinColumn(name = "user_id")
4339            private User user;
4340
4341            public Order() {}
4342
4343            public Order(String order_name, String note, int price, User user) {
4344                this.order_name = order_name;
4345                this.note = note;
4346                this.price = price;
4347                this.user = user;
4348            }
4349
4350            public Integer getOrder_id() {
4351                return order_id;
4352            }
4353
4354            public void setOrder_id(Integer order_id) {
4355                this.order_id = order_id;
4356            }
4357
```

```
        public String getOrder_name() {
            return order_name;
        }

        public void setOrder_name(String order_name) {
            this.order_name = order_name;
        }

        public String getNote() {
            return note;
        }

        public void setNote(String note) {
            this.note = note;
        }

        public int getPrice() {
            return price;
        }

        public void setPrice(int price) {
            this.price = price;
        }

        public User getUser() {
            return user;
        }

        public void setUser(User user) {
            this.user = user;
        }

        @Override
        public String toString() {
            return "Order{" + "orderId=" + order_id + ", orderName='" + order_name + '\'' + ",
            note='" + note + '}' + "\n";
        }
    }
```

6)src/META-INF/persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1" xmlns="http://xmlns.jcp.org/xml/ns/persistence" xmlns:xsi="
http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="
http://xmlns.jcp.org/xml/ns/persistence
http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
    <persistence-unit name="ManyToOneDemo" transaction-type="RESOURCE_LOCAL">

        <class>com.javasoft.entity.User</class>
        <class>com.javasoft.entity.Order</class>
        <properties>
            <property name="javax.persistence.jdbc.url"
                value="jdbc:h2:tcp://localhost/~/test" />
            <property name="javax.persistence.jdbc.driver"
                value="org.h2.Driver" />
            <property name="javax.persistence.jdbc.user" value="sa" />
            <!-- <property name="javax.persistence.jdbc.password" value=""/> -->
            <property name="eclipselink.logging.level" value="FINE" />
            <property name="eclipselink.ddl-generation" value="create-tables" />
        </properties>
    </persistence-unit>
</persistence>
```

7)ManyToOneTest.java

-com.javasoft.service package 생성

-com.javasoft.service > right-click > New > Other > Java > JUnit > JUnit Test Case
-Select New JUnit 4 test
-Name : ManyToOneTest > Next >
-Select Perform the following action : Add JUnit 4 library to the build path > OK

```
package com.javasoft.test;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

import org.junit.After;
import org.junit.Assert;
import org.junit.Before;
import org.junit.Test;

import com.javasoft.entity.Order;
import com.javasoft.entity.User;

public class ManyToOneTest {
   private EntityManagerFactory entityManagerFactory;
   private EntityManager entityManager;

   @Test
   public void oneToManyAndManyToOneTest() {
      Order order = new Order();
      order.setOrder_name("test order");
      order.setPrice(123);
      order.setNote("test note");
      User user = new User();
      user.setUsername("한지민");
      user.setNick_name("jimin");
      user.setAddress("서울");

      // relationship
      user.addOrder(order);
      order.setUser(user);
      entityManager.persist(user);

      Assert.assertEquals(user.getOrders().get(0).getOrder_id(), order.getOrder_id());
   }

   @Before
   public void setUp() throws Exception {
      entityManagerFactory = Persistence.createEntityManagerFactory("ManyToOneDemo");
      entityManager = entityManagerFactory.createEntityManager();
      entityManager.getTransaction().begin();
   }

   @After
   public void after() {
      entityManager.getTransaction().commit();
      entityManager.close();
   }
}
```