Day27: Association Mapping

DB designing team designs the tables of the project according to “normalization rules”. There are six normalization rules/forms. The second normalization rule says to design the tables having integrity constraints, which means tables should be designed having relations like many-to-one, one-to-many, one-to-one, and many-to-many. When tables are in association/relationship we can access one table’s data based on another table’s data, because the records of one table represent the records of another table. DB team takes the support of primary key and foreign key constraints to design the tables having relationships.

When two DB tables are in a relationship their JPA persistence classes must be designed and configured supporting that relationship, this work is called “**Association Mapping**”. When persistence classes are designed to support relationships then the objects of these classes are actually in object-level relationship/association.

Example:

Citizen-Passport	one-to-one	One Citizen contains only one Passport
User-Phone Number	one-to-many	One user can have many phone Numbers
Phone Number- User	many-to-one	Many phone Numbers belong to one user.
Employee-Department	many-to-one	Many Employees belong to one Department
Department- Employee	one-to-many	One Department can have many Employees.
Student-Course	many-to-many	One Student can do many Course, One Course can have many Students
Project-Employee	many-to-many	One Project can has many Employees, One Employee will be involved in many Projects

Using Phone Number and User we can have both one-to-many, and many-to-one relationships, but generally, Phone Number doesn’t need User Information, only the User needs Phone Number Information. (User class has Phone Number, Phone Number class need not have User).

Using Employee-Department we can have both one-to-many, and many-to-one relationships, here Employee and Department both may need the information of other, so we create 12M bidirectional association mapping because one-to-many bidirectional association mapping and M21 bidirectional association mapping are same.

To define one-to-one, many-to-one, and one-to-many two tables are enough (parent, child) but to define many-to-many association three tables are required (table1, table2, relationship table).

By using parent persistence class object of parent table we are able to access the associated child class object/objects data of child table, and if reverse is not possible then it is called Unidirectional association. If reverse is also possible then it is called Bidirectional association.

Parent persistence class => Persistence class for Parent table (table having PRIMARY KEY)

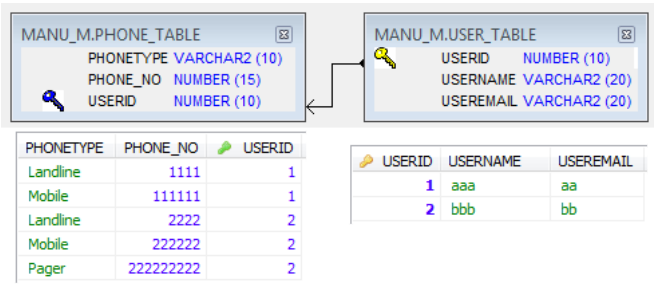
Child persistence class => Persistence class for Child table (table having FOREIGN KEY)

The JPA programmer should always design persistence classes based on the E-R diagrams (Entity-Relationship Diagram) given by DB Team.

To perform association mapping, persistence class should follow below conditions:

Persistence classes should have a property whose type should be of other persistence class, for example User class should have a property of type Phone Number. In persistence class we use following annotations: ManyToMany, OneToMany, ManyToOne,

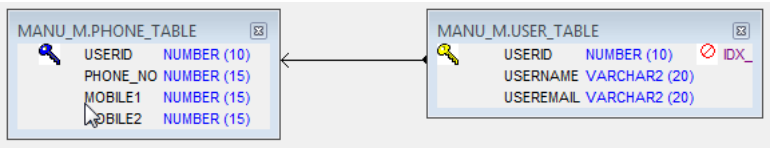
OneToOne



PHONETYPE	PHONE_NO	USERID
Landline	1111	1
Mobile	111111	1
Landline	2222	2
Mobile	222222	2
Pager	222222222	2

USERID	USERNAME	USEREMAIL
1	aaa	aa
2	bbb	bb

In the above tables between User and Phone Number, tables are designed such that one User has Many Phone Numbers, so between User and Phone Number we should use 12M bidirectional association mapping(One user has many phone numbers and many phone numbers belongs to one user). But generally user needs phone number information and phone number doesn't need user information hence we use 12M unidirectional association mapping.



USERID	PHONE_NO	MOBILE1	MOBILE2
1	111	1111	11111
2	222	2222	22222
3	333	3333	3333

USERID	USERNAME	USEREMAIL
1	aaa	aa
2	bbb	bb
3	ccc	cc

In the above tables between User and Phone Number, tables are designed such that one User record has only one record in the Phone Number table. This is also may be the chance that Database tables are designed. Here we should use 121 Bidirectional association mapping, but as said earlier phone number doesn't need user information, hence generally we use 121 unidirectional association mapping. The JPA programmer should always design persistence classes based on the E-R diagrams (Entity-Relationship Diagram) given by the DB team.

Many-to-One Unidirectional Association Mapping:

Many-To-One mapping is an association between collection of same persistence objects and their related persistence object. Many persistence objects mapped to one related persistence object. Tables of both the persistence classes will be related in database. As an example, many students belong to one address. If one persistence object uses other and in back if other is not using the first persistence object then it becomes unidirectional.

Now let us see how this mapping is done in persistence class by going through an example. The following code shows how to do many to one mapping. We created two entities, Address and Student. Many students belong to one address. In the Student class we annotate the address field by **@ManyToOne annotation**.

Database script (MySQL)

```
CREATE TABLE ADDRESS(

AID INT(5) PRIMARY KEY AUTO_INCREMENT,

CITY VARCHAR(30),

ZIPCODE VARCHAR(30)

);



CREATE TABLE STUDENT(

SID INT(5) PRIMARY KEY AUTO_INCREMENT,

SNAME VARCHAR(30),

AID INT(5),

CONSTRAINT FOREIGN KEY (AID) REFERENCES ADDRESS (AID)

);
```

Address.java:

```
@Entity
@Table(name = "ADDRESS")
public class Address {


        @Id
        private int aid;
        private String city;
        private String zipcode;

//getter and setters

}
```

Student.java:

```
@Entity
@Table(name = "STUDENT")
public class Student {

        @Id
        @GeneratedValue(strategy = GenerationType.AUTO)
        private int sid;
        private String sname;

        @ManyToOne(cascade = CascadeType.ALL)
        @JoinColumn(name= "AID")
        private Address address;

//getters and setters

}
```

Demo.java:

```
public class Demo {

        public static void main(String[] args) {

                EntityManager em = EMUtil.provideEntityManager();

                em.getTransaction().begin();

                Address add1 = new Address();
```

```
                add1.setCity("BANGALORE");
                add1.setZipcode("560010");
                add1.setAid(1);

                Student stu1 = new Student();
                stu1.setSname("Manu");
                stu1.setAddress(add1);

                Student stu2 = new Student();
                stu2.setSname("Manjunath");
                stu2.setAddress(add1);

                Student stu3 = new Student();
                stu3.setSname("Advith");
                stu3.setAddress(add1);

                em.persist(stu1);
                em.persist(stu2);
                em.persist(stu3);

                // STUDENT CAN GET ADDRESS
                Student stu = em.find(Student.class, 1);
                System.out.println("student id is " + stu.getSid());
                System.out.println("student name is " + stu.getSname());
                System.out.println("student city is " + stu.getAddress().getCity());
                System.out.println("student zipcode is " + stu.getAddress().getZipcode());

                // ADDRESS CANNOT GET STUDENT
                /*
                Address add1 = em.find(Address.class, 1);
                System.out.println("address id is " + add1.getAid());
                System.out.println("address city is " + add1.getCity());
                System.out.println("address zipcode is " + add1.getZipcode());
                */
                em.getTransaction().commit();
        }
}
```

One-to-Many Unidirectional Association Mapping:

One-to-many mapping is an association between one persistence object holding the collection of same related persistence objects. One persistence object mapped to many persistence objects. Tables of both the persistence classes will be related in database. As an example, one user can have many phone numbers. If one persistence object uses other and in back if other is not using the first persistence object then it becomes unidirectional.

Now let us see how this mapping is done in persistence class by going through an example. The following code shows how to do one-to-many mapping. We created two entities, Phoneuser and Phone. One Phoneuser holds many Phone. In the Phoneuser class we annotate the Phone field by **@OneToMany annotation**

.

Database script (MySQL):

```
CREATE TABLE PHONEUSER
(
USERID NUMERIC(10) Primary key,
USERNAME VARCHAR(20) ,
USEREMAIL VARCHAR(20)
);



CREATE TABLE PHONE
(
PHONEID INT(10) PRIMARY KEY AUTO_INCREMENT,
PHONETYPE VARCHAR(10),
PHONENO NUMERIC(15),
USERID NUMERIC(10)
);

ALTER TABLE PHONE ADD
(
FOREIGN KEY(USERID) REFERENCES PHONEUSER (USERID)
)
```

Phone.java:

```
@Entity
@Table(name = "PHONE")
public class Phone {

        @Id
        @GeneratedValue(strategy = GenerationType.AUTO)
        private int phoneid;
        private String phonetype;
        private long phoneno;
        private int userid;

//getters and setters

}
```

Phoneuser.java:

```
@Entity
@Table(name="PHONEUSER")
public class Phoneuser {

        @Id
        private int userid;
        private String useremail;
        private String username;

        @OneToMany(cascade=CascadeType.ALL)
        @JoinColumn(name="USERID")
        private List<Phone> phoneTables;
```

```
//getters and setters

}
```

Demo.java:

```
public class Demo {

	public static void main(String[] args) {



		EntityManager em = EMUtil.provideEntityManager();

		em.getTransaction().begin();


		Phoneuser phoneUser = new Phoneuser();
		phoneUser.setUsername("AAA");
		phoneUser.setUseremail("AAA@mail.com");
		phoneUser.setUserid(1);

		Phone phone1=new Phone();
		phone1.setPhoneno(111111);
		phone1.setPhonetype("Mobile");
		phone1.setUserid(1);

		Phone phone2=new Phone();
		phone2.setPhoneno(1111);
		phone2.setPhonetype("LandLine");
		phone2.setUserid(1);

		List<Phone> list = new ArrayList<>();
		list.add(phone1);
		list.add(phone2);

		phoneUser.setPhoneTables(list);

		em.persist(phoneUser);

		em.getTransaction().commit();



		em.getTransaction().begin();

		// User can access Phone Number

		Phoneuser u = em.find(Phoneuser.class, 1);

		System.out.println("userId is "+u.getUserid());
		System.out.println("UserName is "+u.getUsername());

		List<Phone> list1 = u.getPhoneTables();

		for(Phone p:list1){

			System.out.println("Phone Number is "+p.getPhoneno());

			System.out.println("phone Type is "+p.getPhonetype());

		}



		//Phone Number cannot access User Details

		/*Phone ph=(Phone)ses.load(Phone.class, new Integer(1));

		System.out.println("Phone Number is "+ph.getPhoneno());

		System.out.println("phone Type is "+ph.getPhonetype());

		*/

	}
```

```
}
```

One-to-Many, Many-to-One Bidirectional Association Mapping:

Many-To-One mapping is an association between collection of same persistence objects and their related persistence object. Many persistence objects mapped to one related persistence object.

One-to-many mapping is an association between one persistence object holding the collection of same related persistence objects. One persistence object mapped to many persistence objects.

Tables of both the persistence classes will be related in database.

If one persistence object uses other and in back if other using the first persistence object then it becomes bidirectional. If it is bidirectional, then one-to-many bidirectional association mapping and many-to-one bidirectional association mapping both become one and same.

As an example, the association between Employee and Department. Here Employee needs Department information and as well as Department needs Employee information and hence it becomes bidirectional mapping.

Example:

Now let us see how this mapping is done in persistence class by going through an example. The following code shows how to do one-to-many bidirectional association mapping and many-to-one bidirectional association mapping. We created two entities, Employee and Department. One Employee belongs to one Department but one Department can have many Employee. In Employee class we annotate the Department field with @ManyToOne annotation. In Department class we annotate the collection of Employee by @OneToMany annotation.

Database script (MySQL):

```
CREATE TABLE DEPARTMENT(

  DEPTID INT(5) PRIMARY KEY AUTO_INCREMENT,

  DNAME VARCHAR(30)

);

CREATE TABLE EMPLOYEE(

EMPID INT(5) PRIMARY KEY AUTO_INCREMENT,

ENAME VARCHAR(30),

EMAIL VARCHAR(30),

DEPTID INT(5),

CONSTRAINT FOREIGN KEY (DEPTID) REFERENCES DEPARTMENT (DEPTID)

);
```

Employee.java:

```
@Entity
@Table(name = "EMPLOYEE")
public class Employee {

        @Id
        @GeneratedValue(strategy = GenerationType.AUTO)
        @Column(name = "EMPID")
        private int empid;

        @Column(name = "ENAME")
```

```
        private String ename;

        @Column(name = "EMAIL")
        private String email;

        @ManyToOne
        @JoinColumn(name = "DEPTID")
        private Department department;

//getters and setters

}
```

Department.java:

```
@Entity
@Table(name = "DEPARTMENT")
public class Department {

        @Id
        @GeneratedValue(strategy = GenerationType.AUTO)
        @Column(name = "DEPTID")
        private int deptid;

        @Column(name = "DNAME")
        private String dname;

        @OneToMany(mappedBy = "department", cascade = CascadeType.ALL)
        private Set<Employee> employees = new HashSet<Employee>();

  //getters and setters

}
```

Demo.java:

```
public class Demo{

        public static void main(String[] args) {


                EntityManager em = EMUtil.provideEntityManager();

                em.getTransaction().begin();

                Department dept = new Department();
                dept.setDname("Engineering");

                Employee emp1 = new Employee();
                emp1.setEname("Manu Manjunatha");
                emp1.setEmail("manu.m@java4coding.com");

                //ASSOCIATE DEPARTMENT WITH EMPLOYEE
                emp1.setDepartment(dept);


                Employee emp2 = new Employee();
                emp2.setEname("Advith");
                emp2.setEmail("advith@java4coding.com");

                //ASSOCIATE DEPARTMENT WITH EMPLOYEE
                emp2.setDepartment(dept);

                Employee emp3 = new Employee();
                emp3.setEname("Likitha Tyagraj");
                emp3.setEmail("likitha@java4coding.com");

                //ASSOCIATE DEPARTMENT WITH EMPLOYEE
                emp3.setDepartment(dept);

                Set<Employee> s = new HashSet<>();
                s.add(emp1);
```

```
                s.add(emp2);
                s.add(emp3);


                //ASSOCIATE EMPLOYEES WITH DEPARTMENT
                dept.setEmployees(s);

                em.persist(dept);

                em.getTransaction().commit();




                em.getTransaction().begin();

                // DEPARTMENT CAN ACCESS EMPLOYEE

                Department d = em.find(Department.class, 1);

                System.out.println("Department id is " + d.getDeptid());

                System.out.println("Department name is " + d.getDname());




                Set<Employee> e = d.getEmployees();


                for (Employee emp : e) {

                        System.out.println("employee id is " + emp.getEmpid());

                        System.out.println("employee name is " + emp.getEname());

                        System.out.println("employee email is " + emp.getEmail());

                }


                // EMPLOYEE CAN ACCESS DEPARTMENT BECAUSE IT IS BIDRECTIONAL

                Employee emp = em.find(Employee.class, 2);

                System.out.println("Employee id is " + emp.getEmpid());

                System.out.println("Employee name is " + emp.getEname());

                System.out.println("Employee email is " + emp.getEmail());


                System.out.println("Employee Department is "+emp.getDepartment().getDeptid());

                System.out.println("Employee Department is "+emp.getDepartment().getDname());


                em.getTransaction().commit();

        }

}
```

One-to-One Unidirectional Association Mapping:

One-To-One mapping is an association between one persistence object and another one related persistence object. If one persistence object uses other and in back if other is not using the first persistence object then it becomes unidirectional. If one persistence object uses other and in back if other using the first persistence object then it becomes bidirectional.

As an example, the association between Employee and Employeedetail, Here One Employee has one Employeedetail. Employee needs Employeedetail and Employeedetail does not need Employee and hence it becomes **unidirectional**.

Example:

Now let us see how this mapping is done in persistence class by going through an example. The following code shows how to do one-to-one unidirectional association mapping. We created two entities, Employee and Employeedetail. In the Employee class we annotate the Employeedetail field by **@OneToOne annotation**

.

Database script (MySQL):

```
CREATE TABLE EMPLOYEE (

EMPID INT(10) PRIMARY KEY AUTO_INCREMENT,

NAME VARCHAR(50)

);



CREATE TABLE EMPLOYEEDETAIL (

EMPDETAILID  INT(10) PRIMARY KEY AUTO_INCREMENT,

EMPID INT(10),

STATE VARCHAR(20),

AGE NUMERIC(10),

CONSTRAINT FOREIGN KEY (EMPID) REFERENCES EMPLOYEE (EMPID)

);
```

Employee.java:

```
@Entity
@Table(name="Employee")
public class Employee  {


        @Id
        @Column(name = "EMPID")
        private int empid;

        private String name;

        @OneToOne(cascade = CascadeType.ALL )
        @JoinColumn(name = "EMPID")
        private Employeedetail employeedetails;

      //getters and setters

}
```

Employeedetail.java:

```
@Entity
@Table(name = "EMPLOYEEDETAIL")
public class Employeedetail  {
```

```
        @Id
        @GeneratedValue(strategy = GenerationType.AUTO)
        private int empdetailid;

        private int age;

        private String state;

        private int empid;

    //getters and setters

}
```

Demo.java:

```
public class Demo{

        public static void main(String[] args) {


                EntityManager em = EMUtil.provideEntityManager();

                em.getTransaction().begin();

                Employee emp = new Employee();

                emp.setName("Manu Manjunatha");

                emp.setEmpid(1);

                Employeedetail empdetail = new Employeedetail();

                empdetail.setAge(26);

                empdetail.setState("Karnataka");

                empdetail.setEmpid(1);

                emp.setEmployeedetails(empdetail);

                 em.persist(emp);

                // Employee can access EmployeeDetails

                Employee e = em.find(Employee.class, 1);

                System.out.println(e.getName());

                System.out.println(e.getEmployeedetails().getState());

                // EmployeeDetails can not access Employee

                em.getTransaction().commit();

        }

}
```

One-to-One Bidirectional Association Mapping:

One-To-One mapping is an association between one persistence object and another one related persistence object. If one persistence object uses other and in back if other using the first persistence object then it becomes bidirectional. If one persistence object uses other and in back if other is not using the first persistence object then it becomes unidirectional. As an example, the association between, Citizen and Passport, here one citizen has one Passport. Citizen needs Passport and also Passport does

need Citizen and hence it becomes **bidirectional association**
.

Example:

Now let us see how this mapping is done in persistence class by going through an example. The following code shows how to do one-to-one unidirectional association mapping. We created two entities, Citizen and Passport. In the Citizen class we annotate the Passport field by @OneToOne annotation and also in Passport class we annotated the Citizen field by **@OneToOne annotation**
.

Database script (MySQL):

```
CREATE TABLE PASSPORT (

  PID INT(5) PRIMARY KEY AUTO_INCREMENT,

  PNUMBER VARCHAR(30)

);



CREATE TABLE CITIZEN (

  CID INT(5) PRIMARY KEY AUTO_INCREMENT,

  CNAME VARCHAR(30),

  PID INT(5),

  CONSTRAINT FOREIGN KEY (PID) REFERENCES PASSPORT (PID)

);
```

Citizen.java:

```
@Entity

@Table(name = "CITIZEN")

public class Citizen {



       @Id
       @GeneratedValue(strategy = GenerationType.AUTO)
        @Column(name = "CID")
        private int cid;

        @Column(name = "CNAME")
        private String cname;

        @OneToOne
        @PrimaryKeyJoinColumn
        private Passport passport;

        //getters and setters

}
```

Passport.java:

```
@Entity
@Table(name = "PASSPORT")
public class Passport {
```

```
        @Id
        @GeneratedValue(strategy = GenerationType.AUTO)
        @Column(name = "PID")
        private int pid;

        @Column(name = "PNUMBER")
        private String pnumber;

        @OneToOne( mappedBy = "passport", cascade = CascadeType.ALL)
        @JoinColumn(name="PID")
        private Citizen citizen;

      //getters and setter

  }
```

Demo.java:

```
public class Demo{

        public static void main(String[] args) {



                EntityManager em = EMUtil.provideEntityManager();

                em.getTransaction().begin();

                Citizen c = new Citizen();
                c.setCname("Manu Manjunatha");

                Passport passport = new Passport();
                passport.setPnumber("123456");

                // ASSOCIATE CITIZEN WITH PASSPORT
                passport.setCitizen(c);

                // ASSOCIATE PASSPORT WITH CITIZEN
                c.setPassport(passport);

                em.persist(passport);

                em.getTransaction().commit();



                em.getTransaction().begin();

                // CITIZEN CAN ACCESS PASSPORT

                Citizen cz = em.find(Citizen.class,1);

                System.out.println("Citizen id is " + cz.getCid());

                System.out.println("Citizen name is " + cz.getCname());

                System.out.println("Citizen passport number is " + cz.getPassport().getPnumber());



                // PASSPORT CAN ACCESS CITIZEN BECAUSE IT IS BIDIRECTIONAL

                Passport pt = em.find(Passport.class,1);

                System.out.println("Passport id is " + pt.getPid());

                System.out.println("Passport number is " + pt.getPnumber());

                System.out.println("Citizen name is " + pt.getCitizen().getCname());



                em.getTransaction().commit();
```

```
        }
}
```

Many-to-Many Bidirectional Association Mapping:

Many-To-Many mapping is an association between one many persistence object and many other related persistence object. In Many-To-Many mapping, relationship will be **always bidirectional**.

Let us consider persistence object A holding persistence objects X, Y, Z. persistence object Z can hold persistence objects M, N, O. Now these persistence objects M, N, O are related to persistence object Z. But not related to persistence object A. But persistence object Z is related to persistence object A. In this way it becomes conflict in keeping the relationship. To handle this situation in Many-To-Many Bidirectional Association Mapping, we take the support of third table called relationship table. This table maintains the relationship between records.

As an example, the association between Student and Course is a bidirectional many-to-many mapping. One Student can subscribe for many Course. One Course can be attended by many students. Student needs Course information and as well as Course needs Student information.

Example:

Now let us see how this mapping is done in persistence class by going through an example. The following code shows how to do Many-To-Many bidirectional association mapping. We created two entities, Student and Course. In the Student class we annotate the collection of Course field by @ManyToMany annotation and also in Course class we annotate the collection Student field by @ ManyToMany annotation. Now either in Student or in Course we have annotate the collection of other persistence object with the annotation @JoinTable by specifying the relationship table. Here the relationship table is STUDENTCOURSE table, this maintains the relationship between records.

Database script (MySQL):

```
CREATE TABLE STUDENT(

SID INT(5) PRIMARY KEY AUTO_INCREMENT,

SNAME VARCHAR(30)

);


CREATE TABLE COURSE(

CID INT(5) PRIMARY KEY AUTO_INCREMENT,

CNAME VARCHAR(30)

);


CREATE TABLE STUDENTCOURSE(

SID INT(5),

CID INT(5),

CONSTRAINT FOREIGN KEY (SID) REFERENCES STUDENT (SID),
```

```
CONSTRAINT FOREIGN KEY (CID) REFERENCES COURSE (CID)

);
```

Course.java:

```
@Entity
@Table(name = "COURSE")
public class Course {

	@Id
	@Column(name = "CID")
	@GeneratedValue(strategy=GenerationType.AUTO)
	private int cid;

	@Column(name = "CNAME")
	private String cname;

	@ManyToMany(mappedBy = "courses",cascade=CascadeType.ALL)
	private Set<Student> students;

	//getters and setters

}
```

Student.java:

```
@Entity
@Table(name = "STUDENT")
public class Student {

	@Id
	@Column(name = "SID")
	@GeneratedValue(strategy = GenerationType.AUTO)
	private int sid;

	@Column(name = "SNAME")
	private String sname;

	@ManyToMany(cascade = CascadeType.ALL)
	@JoinTable(name = "STUDENTCOURSE", joinColumns = { @JoinColumn(name = "SID") }, inverseJoinColumns = { @JoinColumn(name = "CID") })
	private Set<Course> courses;

	//getters and setters
}
```

Demo.java:

```
public class Demo{

	public static void main(String[] args) {

		EntityManager em = EMUtil.provideEntityManager();



		em.getTransaction().begin();

		Course c1 = new Course();
		c1.setCname("JAVA");

		Course c2 = new Course();
		c2.setCname("SERVLET");

		Course c3 = new Course();
		c3.setCname("JSP");
```

```
		Student s1 = new Student();
		s1.setSname("Manu Manjunatha");

		Student s2 = new Student();
		s2.setSname("Advith Tyagraj");


		Set<Course> sc = new HashSet<>();
		sc.add(c1);
		sc.add(c2);
		sc.add(c3);

		Set<Student> ss = new HashSet<>();
		ss.add(s1);
		ss.add(s2);

		// ASSOCIATING STUDENT WITH COURSE

		c1.setStudents(ss);
		c2.setStudents(ss);
		c3.setStudents(ss);

		// ASSOCIATING COURSE WITH STUDENT
		s1.setCourses(sc);
		s2.setCourses(sc);

		em.persist(s1);
		em.persist(s2);

		em.getTransaction().commit();



		em.getTransaction().begin();

		// STUDENT CAN ACCESS COURSE

		Student s = em.find(Student.class, 1);

		System.out.println("Student id is " + s.getSid());

		System.out.println("Student name is " + s.getSname());

		Set<Course> z = s.getCourses();

		for (Course course : z) {

			System.out.println("This student has joined for following courses " + course.getCname());

		}


		// COURSE CAN ACCESS STUDENT

		Course c = em.find(Course.class, 1);

		System.out.println("Course id is " + c.getCid());

		System.out.println("Course name is " + c.getCname());

		Set<Student> students = c.getStudents();

		for (Student stu : students) {

			System.out.println("Following students have joined for this course " + stu.getSname());

		}

		em.getTransaction().commit();

	}

}
```