**Draw Backs of JDBC:**

* In JDBC, if we open a database connection we need to write in try, and if any exceptions occurred catch block will take care about it, and finally used to close the connections.
* here as a programmer we must close the connection, Actually if we didn’t close the connection in the finally block, then jdbc doesn’t responsible to close that connection.
* In JDBC we need to write Sql commands in various places, after the program has created if the table structure is modified then the JDBC program doesn’t work, again we need to modify and compile and re-deploy required, which is tedious.
* JDBC used to generate database related error codes if an exception will occurs, but java programmers are unknown about this error codes right.
* In the Enterprise applications, the data flow with in an application from class to class will be in the form of objects, but while storing data finally in a database using JDBC then that object will be converted into text.  Because JDBC doesn’t transfer objects directly.

In order to overcome above problems,  Hibernate came into picture..!

**What is Hibernate?**

Hibernate is the ORM tool given to transfer the data between a java (object) application and a database (Relational) in the form of the objects.

Hibernate is the open source light weight tool given by **Gavin King**.

Hibernate can run with in or with out server, I mean it will suitable for all types of java applications (stand alone or desktop or any servlets bla bla.)

Hibernate is purely for persistence (to store/retrieve data from Database).

## ****Syntax Of Mapping xml:****

<hibernate-mapping>

<class name="POJO class name" table="table name in database">

<id name="variable name" column="column name in database" type="java/hibernate type" />

<property name="variable1 name" column="column name in database" type="java/hibernate type" />

<property name="variable2 name" column="column name in database" type="java/hibernate type" />

</class>

</hibernate-mapping>

**Configuration:**

Configuration is the file loaded into an hibernate application when working with hibernate, this configuration file contains 3 types of information.

* Connection Properties
* Hibernate Properties
* Mapping file name(s)

We must create one configuration file for each database we are going to use, suppose if we want to connect with 2 databases, like Oracle, MySql, then we must create 2 configuration files.

No. of databases we are using  = That many number of configuration files

**Syntax Of Configuration xml:**

<hibernate-configuration>

<session-factory>

<!-- Related to the connection START -->

<property name="connection.driver\_class">Driver Class Name </property>

<property name="connection.url">URL </property>

<property name="connection.user">user </property>

<property name="connection.password">password</property>

<!-- Related to the connection END -->

<!-- Related to hibernate properties START -->

<property name="show\_sql">true/false</property>

<property name="dialect">Database dialect class</property>

<property name="hbm2ddl.auto">create/update or what ever</property>

<!-- Related to hibernate properties END-->

<!-- Related to mapping START-->

<mapping resource="hbm file 1 name .xml" / >

<mapping resource="hbm file 2 name .xml" / >

<!-- Related to the mapping END -->

</session-factory>

</hibernate-configuration>

**Interacting With Multiple Databases in Hibernate :**

Step 1 : **Person.java**

package javakart;

import javax.persistence.Column;

import javax.persistence.Entity;

import javax.persistence.Id;

import javax.persistence.Table;

@Entity

@Table(name="PERSON")

public class Person{

@Id

@Column(name="person\_id")

private Integer personId;

@Column(name="name")

private String name;

// Getters & Setters

public Integer getPersonId(){

return personId;

}

public void setPersonId(Integer personId){

this.personId = personId;

}

public String getName(){

return name;

}

public void setName(String name){

this.name = name;

}

}

#### Step 2: Create HibernateUtil.java and Main.java for accessing data to & from database

**HibernateUtil.java**

**package** javakart;  
  
 **import** org.hibernate.Session;  
 **import** org.hibernate.SessionFactory;  
 **import** org.hibernate.cfg.Configuration;  
  
**public class** HibernateUtil{  
  
 @SuppressWarnings(**"deprecation"**)  
 **private static** SessionFactory buildSessionFactory(String configFile){  
 **try**{  
 **return new** Configuration().configure(configFile).buildSessionFactory();  
 }**catch**(Throwable ex){  
 System.***err***.println(**"Initial SessionFactory creation failed."** + ex);  
 **throw new** ExceptionInInitializerError(ex);  
 }  
 }  
  
 **public static** Session openSession(String configFile){  
 SessionFactory sf = *buildSessionFactory*(configFile);  
 **return** sf.openSession();  
 }  
}

**Main.java**  
  
 **package** javakart;  
  
 **import** org.hibernate.Session;  
 **import** org.hibernate.Transaction;  
  
**public class** Main{  
  
 **public static void** main(String args[]){  
 Person p1 = **new** Person();  
 p1.setPersonId(101);  
 p1.setName(**"John"**);  
  
 *storeIntoPostgres*(p1);  
 *storeIntoMySql*(p1);  
 }  
  
 **public static void** storeIntoPostgres(Person p1){  
 **try**{  
 Session s = HibernateUtil.openSession(**"postgres-hibernate.cfg.xml"**);  
 Transaction tx = s.beginTransaction();  
  
 s.save(p1);  
 tx.commit();  
 s.flush();  
 s.close();  
 }**catch**(Exception ex){  
 System.***out***.println(**"Error: "**+ex.getMessage());  
 }  
 }  
  
 **public static void** storeIntoMySql(Person p1){  
 **try**{  
 Session s = HibernateUtil.openSession(**"mysql-hibernate.cfg.xml"**);  
 Transaction tx = s.beginTransaction();  
  
 s.save(p1);  
 tx.commit();  
 s.flush();  
 s.close();  
 }**catch**(Exception ex){  
 System.***out***.println(**"Error: "**+ex.getMessage());  
 }  
 }  
  
}

**postgres-hibernate.cfg.xml**

**<?xml version='1.0' encoding='utf-8'?>  
<!DOCTYPE hibernate-configuration PUBLIC  
"-//Hibernate/Hibernate Configuration DTD 3.0//EN"  
"http**:**//www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">**

**<hibernate-configuration>  
<session-factory>  
<property name="connection.driver\_class">org.postgresql.Driver</property>  
<property name="connection.url">jdbc:postgresql://localhost/Hibernate\_Practice</property>  
<property name="connection.username">postgres</property>  
<property name="connection.password">mypassword</property>  
<property name="connection.pool\_size">10</property>  
<property name="dialect">org.hibernate.dialect.PostgreSQLDialect</property>  
<property name="current\_session\_context\_class">thread</property>  
<property name="cache.provider\_class">org.hibernate.cache.internal.NoCacheProvider</property>  
<property name="show\_sql">true</property>  
<property name="hbm2ddl.auto">update</property>  
  
<mapping class="javakart.Person"/>  
  
</session-factory>  
  
</hibernate-configuration>**

**mysql-hibernate.cfg.xml**

**<?xml version='1.0' encoding='utf-8'?>  
<!DOCTYPE hibernate-configuration PUBLIC  
"-//Hibernate/Hibernate Configuration DTD 3.0//EN"  
"http**:**//www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">  
  
<hibernate-configuration>  
<session-factory>  
<property name="connection.driver\_class">com.mysql.jdbc.Driver</property>  
<property name="connection.url">jdbc:mysql://localhost/Hibernate\_Practice</property>  
<property name="connection.username">postgres</property>  
<property name="connection.password">mypassword</property>  
<property name="connection.pool\_size">10</property>  
<property name="dialect">org.hibernate.dialect.MySQL5Dialect</property>  
<property name="current\_session\_context\_class">thread</property>  
<property name="cache.provider\_class">org.hibernate.cache.internal.NoCacheProvider</property>  
<property name="show\_sql">true</property>  
<property name="hbm2ddl.auto">update</property>  
  
<mapping class="javakart.Person"/>  
</session-factory>  
  
</hibernate-configuration>**

**Advantages of hibernates:**

* Hibernate supports Inheritance, Associations, Collections
* In hibernate if we save the derived class object,  then its base class object will also be stored into the database, it means hibernate supporting inheritance
* Hibernate supports relationships like One-To-Many, One-To-One, Many-To-Many, Many-To-One
* This will also supports collections like List, Set, Map (Only new collections)
* In jdbc all exceptions are checked exceptions, so we must write code in try, catch and throws, but in hibernate we only have Un-checked exceptions, so no need to write try, catch, or no need to write throws.  Actually in hibernate we have the translator which converts checked to Un-checked

-🡪 A history of exceptions — Exceptions and how they should be handled always end in heated debates between Java developers. It isn’t surprising that Hibernate has some noteworthy history as well. Until Hibernate 3.x, all exceptions thrown by Hibernate were checked exceptions, so every Hibernate API forced the developer to catch and handle exceptions. This strategy was influenced by JDBC , which also throws only checked exceptions. However, it soon became clear that this doesn’t make sense, because all exceptions thrown by Hibernate are fatal. In many cases, the best a developer can do in this situation is to clean up, display an error message, and exit the application. Therefore, starting with Hibernate 3.x, all exceptions thrown by Hibernate are subtypes of the unchecked Runtime Exception, which is usually handled in a single location in an application. This also makes any Hibernate template or wrapper API obsolete.

* Hibernate has capability to generate primary keys automatically while we are storing the records into database
* Hibernate has its own query language, i.e hibernate query language which is database independent
* So if we change the database, then also our application will works as HQL is database independent
* HQL contains database independent commands
* While we are inserting any record, if we don’t have any particular table in the database, JDBC will rises an error like “View not exist”, and throws exception, but in case of hibernate, if it not found any table in the database this will create the table for us
* Hibernate supports caching mechanism by this, the number of round trips between an application and the database will be reduced, by using this caching technique an application performance will be increased automatically.
* Hibernate supports annotations, apart from XML
* Hibernate provided Dialect classes, so we no need to write sql queries in hibernate, instead we use the methods provided by that API.

What is Dialect ?

Dialect is nothing but the ability of hibernate to create query based on the database provided in the configuration file. Normally if we create a query in oracle, then if we need to migrate to mysql then there are lots of changes need to be done to make it compatible with mysql. When we write it in hibernate, it will create the query as per the dialect provided.

* Getting pagination in hibernate is quite simple.

**Disadvantages of hibernates:**

* I don’t think there are disadvantages in hibernate
* You know some thing.., Its saying hibernate is little slower than pure JDBC, actually the reason being hibernate used to generate many SQL statements in run time, but i guess this is not the disadvantage 🙂
* But there is one major disadvantage, which was boilerplate code issue, actually we need to write same code in several files in the same application, but spring eliminated this

– higher learning curve than just using JDBC  
– longer time to implement simple DB solutions (setup, jars config files)  
– need to learn formats of config files and annotations above standard java  
– harder to debug and unhelpful error messages from frame work  
– SQL is standard teaching material in academy – you need to relearn how to perform SQL techniques using hibernate API (e.g., JOINs)

Hibernate Flow :

Final flow will be\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Configuration

SessionFactory

Session

Transaction

Close Statements

**Hibernate Insert Query :**

Configuration cfg = **new** Configuration();  
cfg.configure(**"hibernate.cfg.xml"**);  
SessionFactory factory = cfg.buildSessionFactory();  
Session session = factory.openSession();  
Product p=**new** Product();  
p.setProductId(101);  
p.setProName(**"iPhone"**);  
p.setPrice(25000);  
Transaction tx = session.beginTransaction();  
session.save(p);  
System.***out***.println(**"Object saved successfully.....!!"**);  
tx.commit();  
session.close();  
factory.close();

**Difference between save and persist method?**

* difference between hibernate save() and persist() methods is depends on generator class we are using. If our generator class is assigned, then there is no difference between save() and persist() methods. Because generator ‘assigned’ means, as  a programmer we need to give the primary key value to save in the database right
* In case of other than assigned generator class, suppose if our generator class name is Increment means hibernate it self will assign the primary key id value into the database right [ other than assigned generator, hibernate only used to take care the primary key id value remember ], so in this case if we call save() or persist() method then it will insert the record into the database normally

But here thing is,  save() method can return that primary key id value which is generated by hibernate and we can see it by

long s = session.save(k);

In this same case, persist() will never give any value back to the client

**What is Generator class in Hibernate?**

* Generator classes are used to generate the ‘**identifier or primary key value**‘ for a persistent object while saving an object in database.
* Hibernate provides different primary key generator algorithms.
* All hibernate generator classes implements **hibernate.id.IdentifierGenerator** interface, and overrides the **generate(SessionImplementor,Object)** method to generate the ‘**identifier or primary key value**‘.
* If we want our own user defined generator, then we should implement **IdentiferGenerator** interface and override the **generate()**
* **<generator />** tag (which is sub element of **<id />** tag) is used to configure generator class in mapping file.

Hibernate built-in generator classes

**1) Assigned**

* Assigned generator class is the default generator if there is no <generator> tag and supports in all the databases.
* Developer should assign the identifier value to entity object before saving into the database.
* <id name="empId" column="EMPNO">
* <generator class="assigned"/>
* </id>

**2) Increment**

* Increment generator supports in all databases and generates **identifier** value for new records by using below formula.

**Max of Id value in Database + 1**

* For first record it assigns 1 to the **identifier.** For second record it assigns based on above formula. i.e.( **Max of Id value in Database + 1) =( 1+1 ) = 2.**

**3) Sequence**

* Sequence generator does not support MySql database and it is database dependent. i.e. Before using this generator, We should know whether this generator supports in the database or not. If there is no sequence in database, it uses default sequence. For ex for oracle database it uses **HIBERNATE\_SEQUENCE**

**4)hilo**

**5) native**

* It uses internally **identity or sequence or hilo**generator classes.
* **native**picks up **identity or sequence or hilo**generator class depending upon the capabilities of the underlying database.

**6) identity**

* Identity columns are support by DB2, MYSQL, SQL SERVER  and SYBASE databases.

**7) Foreign**

* foreign uses the identifier of another associated object. Usually uses in conjunction with a <one-to-one> primary key association.

**Hibernate Select Query :**

Configuration **cfg** = **new** Configuration();  
cfg.configure(**"hibernate.cfg.xml"**);  
  
 SessionFactory factory = cfg.buildSessionFactory();  
 Session session = factory.openSession();  
 Object o=session.load(Product.**class**,**new** Integer(101));  
  
 Product s=(Product)o;  
  
 *// For loading Transaction scope is not necessary...* System.out.println(**"Loaded object product name is\_\_\_"**+s.getProName());  
  
 System.out.println(**"Object Loaded successfully.....!!"**);  
 session.close();  
 factory.close();

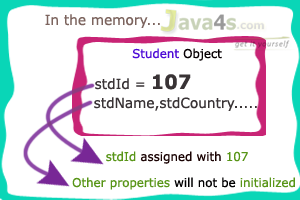
once we loaded the object for the database with load or get methods the object data will be loads into the Product.java(POJO) setter methods, so we are printing by using getter methods

## Difference between Hibernate get() & load() :

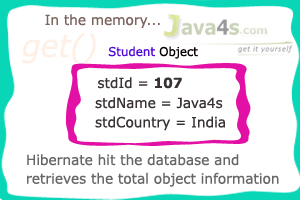
* Both are from Session interface, and we will call them as session.get() & session.load()
* Both will be use for retrieving the object (a row) from the database

Then what’s the difference them ? lets start with load() and then get() method.

Consider a Student class having 3 properties *stdId, stdName, stdCountry*

* 1. **session.load() :**
* When you call session.load() method, it will always return a “proxy” object,  whats the meaning of proxy object ?
* Proxy means, hibernate will prepare some fake object with given *identifier value* in the memory without hitting the database, for example if we call *session.load(Student.class,new Integer(107));*  > hibernate will create one fake Student object [row] in the memory with id 107, but remaining properties of Student class will not even be initialized, observe this graphical representation…  
  
* It will hit the database only when we try to retrieve the other properties of Student object i mean stdName, stdCountry.  If we call s2.getStdName() then hibernate will hit the database and search the row with student id 107 and retrieve the values, if object [row] not found in the database it will throws ObjectNotFoundException.,

## session.get() :

* When you call session.get() method, it will hit the database immediately and returns the original object  
  
* If the row is not available in the database, it returns **null** and if we try to fetch any other member in this case it will give **NullPointerException**

## Q-> which is the best method to use, hibernate load() or get() ?

## If u are not sure that data which is you are searching for is present in dB then u go for load() else get()

## Conclusion : load() is used for lazy loading and get() is used for early loading

## Note : load() may hit the database if lazy=false i.e in eagar loading so it not always return proxy object

**Hibernate delete Query :**

To deleting the object( 1 row) form the database we need to call delete method in the session.

Session **session** = factory.openSession();  
  
Object **o**=**session**.load(Product.**class**,**new** Integer(103));  
Product **p**=(Product)**o**;  
  
Transaction **tx** = **session**.beginTransaction();  
 session.delete(p);  
  
 System.out.println(**"Object Deleted successfully.....!!"**);  
 tx.commit();

**Can We Delete multiple records simultaneously ?**

Object **o1**=session.load(Sample.**class**,**new** Integer(9037,9035));

**if** you write **this**, it gives an Error like **"Remove argument to match Integer(int);"** we can delete multiple data like **this**…….

Object o=session.load(Sample.**class**,**new** Integer(9035));  
 Object o1=session.load(Sample.**class**,**new** Integer(9037));  
 Sample s=(Sample)o;  
 Sample s1=(Sample)o1;  
 Transaction tx = session.beginTransaction();  
 session.delete(s);  
 session.delete(s1);

**Hibernate update Query :**

to update an object (1 complete row) in the database, which is already persisted in the database, then we have the following two approaches…

## Approach 1

Load that object from the database, and modify its values, now hibernate automatically modifies the values on to database also, when ever the transaction is committed.

SessionFactory factory = cfg.buildSessionFactory();  
 Session session = factory.openSession();  
 Object o=session.load(Product.**class**,**new** Integer(105));  
 Product s=(Product)o;  
 Transaction tx = session.beginTransaction();  
*//s.setStno(105);should not update, because we loaded with that number right..?* s.setPrice(4000); *// implicitly update method will be called..*

tx.commit();  
 System.***out***.println(**"Object Updated successfully.....!!"**);  
 session.close();  
 factory.close();

once we call the commit(), automatically update method will be called by hibernate.

 When ever an object is loaded from the database then hibernate stores the loaded object in cache-memory maintained by session-interface

 Once an object is loaded, if we do any modifications on that object by calling its setter methods, then these modification are stored in the object maintained by cache-memory

 if we modify the loaded object for multiple times then also the modifications will be stored in object maintained by the cache-memory only.

 when ever we issue commit() operation then hibernate verify whether any changes are there between the object stored in the cache and object in the database, if changes exists then hibernate automatically updates the database by generating any update operation.

hibernate automatically maintains synchronization between cache-memory object and database table objects (rows)

## Approach 2:

If we want to modify object in the database, then create new object with same id and we must call update() given by session interface.

SessionFactory factory = cfg.buildSessionFactory();  
Session session = factory.openSession();  
Product p=**new** Product();  
p.setProductId(104); *// 104 must be in the DB*p.setProName(**"Someting"**);  
Transaction tx = session.beginTransaction();  
session.update(p);  
tx.commit();  
System.***out***.println(**"Object Updated successfully.....!!"**);  
session.close();  
factory.close();

* Here we no need to load an object from the database
* we will create a new object, and we will assign same id no’s to it and we will call update() explicitly in order to make the changes on the object that is stored in the database

That’s it, actually first approach is  recommended always.

**Difference between merge() and update() method:**

**B**oth update() and merge() methods in hibernate are used to convert the object which is in detached state into persistence state.  But there is little difference.

SessionFactory factory = cfg.buildSessionFactory();  
Session session1 = factory.openSession();  
Student s1 = (Student)session1.get(Student.**class**, **new** Integer(101));  
session1.close();  
s1.setMarks(97);  
Session session2 = factory.openSession();  
Student s2 = (Student)session1.get(Student.**class**, **new** Integer(101));  
Transaction tx=session2.beginTransaction();  
session2.merge(s1);

we just loaded one object s1 into session1 cache and closed session1, so now object s1 in the session1 cache will be destroyed as session1 cache will expires when ever we say session1.close()

 Now s1 object will be in some RAM location, not in the session1 cache here s1 is in detached state, we modified that detached object s1, now if we call update() method then hibernate will throws an error, because we can update the object in the session only. So we opened another session [session2] and again loaded the same student object from the database, so in this session2, we called **session2.merge(s1)**; now into s2 object s1 changes will be merged and saved into the database

Hope you are clear…, actually update and merge methods will come into picture when ever we loaded the same object again and again into the database, like above.

**Difference between primitive and wrapper type in Hibernate?**

**Let’s understand it by example :**

**Ex:** public class Product{

private int productId;

private String proName;

private double price;

Session session = factory.openSession();  
Product p=**new** Product();  
p.setProductId(105);  
p.setProName(**"watch"**);  
*//p.setPrice(35000); see am not setting any value to Price*Transaction tx = session.beginTransaction();  
session.save(p);  
System.***out***.println(**"Object saved successfully.....!!"**);

On using primitive data type if we don’t set the price ,once you execute this program in the database it will saves the price as 0(zero), so misunderstanding of data will happen like watch price is zero 🙂 [ free of cost hah ]

But if we use wrapper type :

public class Product{

Integer productId;

String proName;

Integer price;

}

in this case if we forget to write the setter for the price, in the database its not inserting any thing [ actually it has to insert NULL value, as of now leave it

**Hibernate Lifecycle of POJO:**

POJO class object having 3 states

## Transient:

## One newly created object,with out having any relation with the database, means never persistent, not associated with any Session object

## Persistent:

Having the relation with the database, associated with a unique Session object

## Detached:

previously having relation with the database [persistent ], now not associated with any Session

* When ever an object of a pojo class is created then it will be in the Transient state
* When the object is in a Transient state it doesn’t represent any row of the database, i mean not associated with any Session object, if we speak more we can say no relation with the database its just an normal object
* If we modify the data of a pojo class object, when it is in transient state then it doesn’t effect on the database table
* When the object is in persistent state, then it represent one row of the database, if the object is in persistent state then it is associated with the unique Session
* if we want to move an object from persistent to detached state, we need to do either closing that session or need to clear the cache of the session
* if we want to move an object from persistent state into transient state then we need to delete that object permanently from the database

*// Transient state\_\_\_\_\_start*Product p=**new** Product();  
p.setProductId(101);  
p.setProName(**"iPhone"**);  
p.setPrice(25000);  
*// Transient state\_\_\_\_\_end  
  
// Persistent state\_\_\_\_\_start*Transaction tx = session.beginTransaction();  
session.save(p);  
System.***out***.println(**"Object saved successfully.....!!"**);  
tx.commit();  
*// Persistent state\_\_\_\_\_end*

**if we want to convert the object from Transient state to Persistent state we can do in 2 ways**

* By saving that object like above
* By loading object from database

# Converting Object From Detached to Persistent state :

Session session1 = factory.openSession();  
Product p=**null**; *//Transient state..*Object o=session1.get(Product.**class**, **new** Integer(1001));  
p=(Product)o; *//now p is in Persistent state..*session1.close();  
p.setPrice(36000); *// p is in Detached state*Session session2=factory.openSession();  
Transaction tx=session2.beginTransaction();  
session2.update(p); *// now p reached to Persistent state*tx.commit();  
session2.close();

**Best Practices for ManyToOne and OneToMany Relationship:**

**Don’t use unidirectional one-to-many association**

**OneToOne :** there are 3 ways to create one-to-one relationships between two entities. Either way you have to use [**@OneToOne**](https://docs.oracle.com/javaee/5/api/javax/persistence/OneToOne.html) annotation.

# One-To-One Unidirectional with Shared Primary Key

In One-To-One Unidirectional Shared primary key mapping, two tables share the same primary key.

in One-To-One Unidirectional Shared primary key mapping, two tables share the same primary key.  
The Unidirectional relationship means only one side navigation is possible (STUDENT to ADDRESS in this example). Let’s get going.

|  |
| --- |
| OneToOneUniSharedPrimaryKey_img1 We are discussing an example of Student and Address relationship. A student lives on one Address. And one address can be occupied by only one student. |

#### Step 1: Create required Database Table

create table STUDENT (

   student\_id BIGINT NOT NULL AUTO\_INCREMENT,

   first\_name VARCHAR(30) NOT NULL,

   last\_name  VARCHAR(30) NOT NULL,

   section    VARCHAR(30) NOT NULL,

   PRIMARY KEY (student\_id)

);

create table ADDRESS (

   address\_id BIGINT NOT NULL,

   street VARCHAR(30) NOT NULL,

   city  VARCHAR(30) NOT NULL,

   country  VARCHAR(30) NOT NULL,

   PRIMARY KEY (address\_id),

   CONSTRAINT student\_address FOREIGN KEY (address\_id) REFERENCES STUDENT ( student\_id) ON DELETE CASCADE

);

# Here we have created Student and Address tables. Student table is pretty obvious. But while creating Address table, we have also specified a foreign key constraint from Address table primary key to Student table primary key.

Step 2 : Model class

**Student.java**

@Entity

@Table(name = "STUDENT")

public class Student {

    @Id

    @GeneratedValue

    @Column(name = "STUDENT\_ID")

    private long id;

    @Column(name = "FIRST\_NAME")

    private String firstName;

    @Column(name = "LAST\_NAME")

    private String lastName;

    @Column(name = "SECTION")

    private String section;

    @OneToOne(cascade = CascadeType.ALL)

    @PrimaryKeyJoinColumn

    private Address address;

**Address.java**

@Entity

@Table(name = "ADDRESS")

public class Address {

    @Id

    @Column(name = "ADDRESS\_ID")

    private long id;

    @Column(name = "STREET")

    private String street;

    @Column(name = "CITY")

    private String city;

    @Column(name = "COUNTRY")

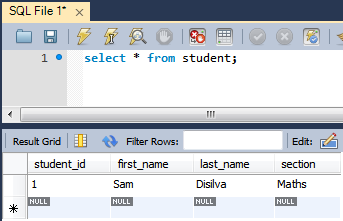
    private String country;

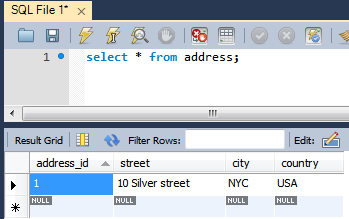
//getetrs setters and constructors

**@OneToOne** on address property of Student class indicates that there is a one-to-one association from Student to Address. **PrimaryKeyJoinColumn** indicates that the primary key of the Student entity is used as a foreign key to the Address entity. **Together these two annotation indicates that both the source and target share the same primary key values**.Also note the use of cascade = CascadeType.ALL. Since Address entity can not exist without Student entity, with this cascade setting, address entity will be updated/deleted on subsequent update/delete on student entity.Note that Address class does not contain any reference to Student, as it is a unidirectional relationship(From Student to Address).

**public class** HibernateStandAlone {  
  
 @SuppressWarnings(**"unchecked"**)  
 **public static void** main(String[] args) {  
 Student student = **new** Student(**"Sam"**,**"Disilva"**,**"Maths"**);  
 Address address = **new** Address(**"10 Silver street"**,**"NYC"**,**"USA"**);  
  
 Session session = HibernateUtil.getSessionFactory().openSession();  
 session.beginTransaction();  
 session.persist(student);  
  
 address.setId(student.getId());  
 student.setAddress(address);  
 session.save(student);  
  
 List<Student> students = (List<Student>)session.createQuery(**"from Student "**).list();  
 **for**(Student s: students){  
 System.***out***.println(**"Details : "**+s);  
 }  
 session.getTransaction().commit();  
 session.close();  
 }  
  
}

Look at how we have first persisted Student class , so that it’s id can be generated. Then we have set the address id with student id(so that foreign key constraint can be respected).Finally we have set the address property of Student and saved student.Thanks to Cascade attribute on address property of Student class, address will be saved automatically on student save.No need to save the address explicitly.



  
That’s it.

# One-To-One Unidirectional with Foreign Key Associations

In One-To-One Unidirectional with Foreign Key association mapping, one table has a foreign key column that references the primary key of associated table.

Step 1: Table creation

create table ADDRESS (

   address\_id BIGINT NOT NULL AUTO\_INCREMENT,

   street VARCHAR(30) NOT NULL,

   city  VARCHAR(30) NOT NULL,

   country  VARCHAR(30) NOT NULL,

   PRIMARY KEY (address\_id)

);

create table STUDENT (

   student\_id BIGINT NOT NULL AUTO\_INCREMENT,

   home\_address\_id BIGINT NOT NULL,

   first\_name VARCHAR(30) NOT NULL,

   last\_name  VARCHAR(30) NOT NULL,

   section    VARCHAR(30) NOT NULL,

   PRIMARY KEY (student\_id),

   CONSTRAINT student\_address FOREIGN KEY (home\_address\_id) REFERENCES ADDRESS ( address\_id)

);

Here we have first created Address table followed by student table as student table contains a foreign key referring to Address table.

Step 2: Create model classes

**Student.java**

@Entity

@Table(name = "STUDENT")

public class Student {

    @Id

    @GeneratedValue

    @Column(name = "STUDENT\_ID")

    private long id;

    @Column(name = "FIRST\_NAME")

    private String firstName;

    @Column(name = "LAST\_NAME")

    private String lastName;

    @Column(name = "SECTION")

    private String section;

    @OneToOne

    @JoinColumn(name="HOME\_ADDRESS\_ID")

    private Address address;

**Address.java**

@Entity

@Table(name = "ADDRESS")

public class Address {

    @Id

@GeneratedValue

    @Column(name = "ADDRESS\_ID")

    private long id;

    @Column(name = "STREET")

    private String street;

    @Column(name = "CITY")

    private String city;

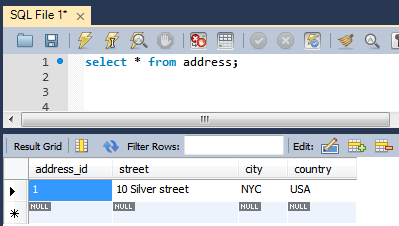
    @Column(name = "COUNTRY")

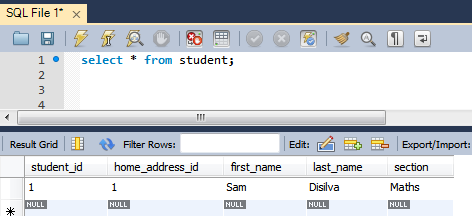
    private String country;

Difference between Student class define here and in the previous tutorial(shared primary key) is that we have replaced @PrimaryKeyJoinColumn with @joinColumn which maps on a seperate column in database but will still point to primary key of address table, thanks to the foreign key constrained we have declared during table creation time. Address class is completly independent of Student table.

**public class** HibernateStandAlone {  
  
 @SuppressWarnings(**"unchecked"**)  
 **public static void** main(String[] args) {  
 Student student = **new** Student(**"Sam"**,**"Disilva"**,**"Maths"**);  
 Address address = **new** Address(**"10 Silver street"**,**"NYC"**,**"USA"**);  
 Session session = HibernateUtil.getSessionFactory().openSession();  
 session.beginTransaction();  
 session.persist(address);  
 student.setAddress(address);  
 session.persist(student);  
 List<Student> students = (List<Student>)session.createQuery(**"from Student "**).list();  
 **for**(Student s: students){  
 System.***out***.println(**"Details : "**+s);  
 }  
 session.getTransaction().commit();  
 session.close();  
 }  
}

Here we have persisted Address class firstly in order to meet foreign key constraint(not null), then we have set student’s address property followed by persisting student.



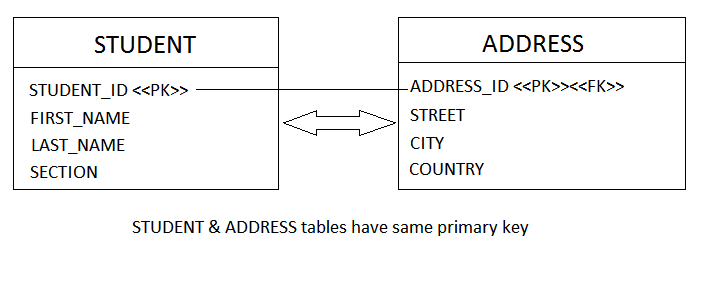


That’s it.

# One-To-One Bidirectional with Shared Primary Key

how to use Hibernate One-To-One Bidirectional Shared primary key mapping using annotation based configuration.

In One-To-One Bidirectional Shared primary key mapping, two tables share the same primary key.  
The Bidirectional relationship means navigation is possible in both direction.

EStep

Step 1 : Create database table ---like first one

Step 2: Create Model classes :

**Student.java**

@Entity

@Table(name = "STUDENT")

public class Student {

    @Id

    @GeneratedValue

    @Column(name = "STUDENT\_ID")

    private long id;

    @Column(name = "FIRST\_NAME")

    private String firstName;

    @Column(name = "LAST\_NAME")

    private String lastName;

    @Column(name = "SECTION")

    private String section;

    @OneToOne(mappedBy="student", cascade = CascadeType.ALL)

    private Address address;

**Address.java**

@Entity

@Table(name = "ADDRESS")

public class Address {

    @Id

    @Column(name="ADDRESS\_ID")

    @GeneratedValue(generator="gen")

    @GenericGenerator(name="gen", strategy="foreign",parameters=@Parameter(name="property", value="student"))

    private long id;

    @Column(name = "STREET")

    private String street;

    @Column(name = "CITY")

    private String city;

    @Column(name = "COUNTRY")

    private String country;

    @OneToOne

    @PrimaryKeyJoinColumn

    private Student student;

Note that now we have a student property in Address class and address property in Student class, which means we can now navigate in either direction.In hibernate, **for bidirectional relationships like this, we have a concept of ownership, means who is the owner of this relationship. Put simply, who is responsible for updating the column in DB on which this relationship depends on.** In our case it’s the **student\_id** of Student table which is driving the complete relationship truck. So we should tell hibernate that it’s the Student class which will manage the relationship.

To do that, we can use **mappedBy** attribute. **mappedBy attribute are always put(annotated) on the inverse side of relation ship and specifies with it’s attribute value, the owner of the relationship.**

|  |
| --- |
| @OneToOne(mappedBy="student", cascade = CascadeType.ALL)  private Address address; |

With this declaration, we ask hibernate to go and find the student property of Address class to know how to manage the relationship/perform some operation. Now in Address class , we have following declaration

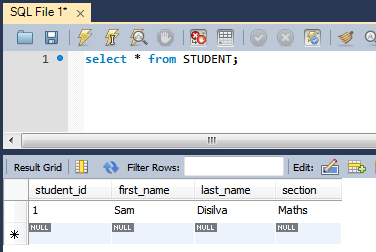
|  |
| --- |
| @OneToOne  @PrimaryKeyJoinColumn  private Student student; |

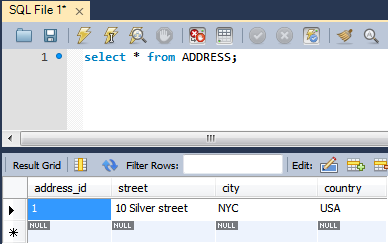
Which simply says that both the Address table and Student table share the same primary key.

But how would you make sure to that Address class use the same id value as used by Student? You have to make sure in your code that you set the id of Address with the id of saved Student before you save address(as we did in one-to-one unidirectional tutorial).If you don’t want to do that, another option is to use hibernate specific annotation @GenericGenerator (that’s what we are doing now in Address class). In Address class, **@GenericGenerator ensures that id value of Address property value will be taken from the id of Student table**.

**public class** HibernateStandAlone {  
  
 @SuppressWarnings(**"unchecked"**)  
 **public static void** main(String[] args) {  
 Student student = **new** Student(**"Sam"**,**"Disilva"**,**"Maths"**);  
 Address address = **new** Address(**"10 Silver street"**,**"NYC"**,**"USA"**);  
 Session session = HibernateUtil.getSessionFactory().openSession();  
 session.beginTransaction();  
 student.setAddress(address);  
 address.setStudent(student);  
 session.save(student);  
 List<Student> students = (List<Student>)session.createQuery(**"from Student "**).list();  
 **for**(Student s: students){  
 System.***out***.println(**"Details : "**+s);  
 }  
 session.getTransaction().commit();  
 session.close();  
 }  
  
}

we have set address property of student and student property of address.Then we have called save only on student, thanks to Cascade attribute on address property of Student class, address will be saved automatically.



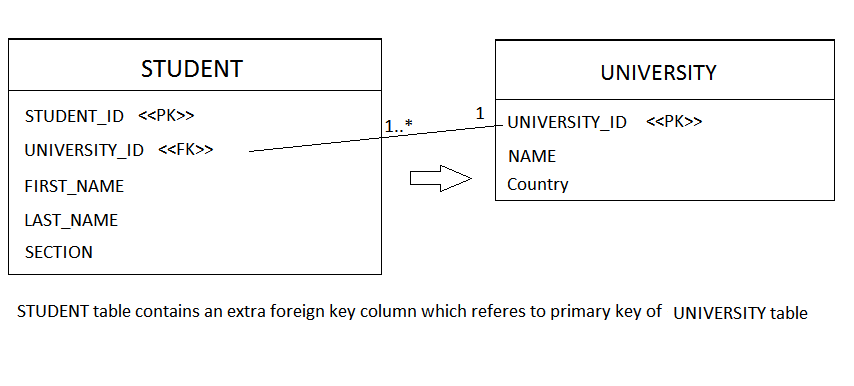


You can see that address was saved with the same id as student.

That’s it.

# Many-To-One Unidirectional

In Many-To-One Unidirectional mapping, one table has a foreign key column that references the primary key of associated table. By Unidirectional relationship means only one side navigation is possible (STUDENT to UNIVERSITY in this example).



#### Step 1: Create required Database Table

Open MySQL terminal / workbench terminal and execute following MySQL script :

**create table UNIVERSITY (**

**university\_id BIGINT NOT NULL AUTO\_INCREMENT,**

**name VARCHAR(30) NOT NULL,**

**country  VARCHAR(30) NOT NULL,**

**PRIMARY KEY (university\_id)**

**);**

**create table STUDENT (**

**student\_id BIGINT NOT NULL AUTO\_INCREMENT,**

**university\_id BIGINT NOT NULL,**

**first\_name VARCHAR(30) NOT NULL,**

**last\_name  VARCHAR(30) NOT NULL,**

**section    VARCHAR(30) NOT NULL,**

**PRIMARY KEY (student\_id),**

**CONSTRAINT student\_university FOREIGN KEY (university\_id) REFERENCES UNIVERSITY (university\_id) ON UPDATE CASCADE ON DELETE CASCADE**

**);**

Here we have first created University table followed by student table as student table contains a foreign key referring to University table.

Step 2 :

2.1.

**University.java**

@Entity

@Table(name = "UNIVERSITY")

public class University {

    @Id

    @GeneratedValue

    @Column(name = "UNIVERSITY\_ID")

    private long id;

    @Column(name = "NAME")

    private String name;

    @Column(name = "COUNTRY")

    private String country;

    public University() {

    }

    public University(String name, String country) {

        this.name = name;

        this.country = country;

    }

 2.2 Student.java

@Entity

@Table(name = "STUDENT")

public class Student {

    @Id

    @GeneratedValue

    @Column(name = "STUDENT\_ID")

    private long id;

    @Column(name = "FIRST\_NAME")

    private String firstName;

    @Column(name = "LAST\_NAME")

    private String lastName;

    @Column(name = "SECTION")

    private String section;

    @ManyToOne(optional = false)

    @JoinColumn(name="UNIVERSITY\_ID")

    private University university;

    public Student() {

    }

    public Student(String firstName, String lastName, String section) {

        this.firstName = firstName;

        this.lastName = lastName;

        this.section = section;

    }

University class is a simple pojo with no information of Student class. Student class on the other hand contains an ManyToOne association to University class.

|  |
| --- |
| @ManyToOne(optional = false)  @JoinColumn(name="UNIVERSITY\_ID")  private University university; |

@ManyToOne indicates that Many student tuples can refer to one University tuple. Also note that we have provided **optional=false** means this relationship becomes mandatory , no student row can be saved without a university tuple reference.@**JoinColumn** says that there is a column UNIVERSITY\_ID in Student table which will refer(foreign key) to primary key of the University table. In this example only Student to University entity navigation is possible. Not viceversa. In practice, however, you are free to use query language to find all the student for a given university.

3.Main.java

Here we have persisted University class firstly in order to meet foreign key constraint(not null), then we have set student’s address property followed by persisting student.

**public class** Main {  
 @SuppressWarnings(**"unchecked"**)  
 **public static void** main(String[] args) {  
 Student student1 = **new** Student(**"Sam"**,**"Disilva"**,**"Maths"**);  
 Student student2 = **new** Student(**"Joshua"**, **"Brill"**, **"Science"**);  
 Student student3 = **new** Student(**"Peter"**, **"Pan"**, **"Physics"**);

University university = **new** University(**"CAMBRIDGE"**, **"ENGLND"**);

student1.setUniversity(university);  
 student2.setUniversity(university);  
 student3.setUniversity(university);

Session session = HibernateUtil.getSessionFactory().openSession();  
 session.beginTransaction();

session.persist(university);  
 session.persist(student1);  
 session.persist(student2);  
 session.persist(student3);

List<Student> students = (List<Student>)session.createQuery(**"from Student "**).list();  
 **for**(Student s: students){  
 System.***out***.println(**"Details : "**+s);  
 System.***out***.println(**"Student University Details: "**+s.getUniversity());  
 }  
 session.getTransaction().commit();  
 session.close();  
 }  
  
}

Hibernate: insert into UNIVERSITY (COUNTRY, NAME) values (?, ?)

Hibernate: insert into STUDENT (FIRST\_NAME, LAST\_NAME, SECTION, UNIVERSITY\_ID) values (?, ?, ?, ?)

Hibernate: insert into STUDENT (FIRST\_NAME, LAST\_NAME, SECTION, UNIVERSITY\_ID) values (?, ?, ?, ?)

Hibernate: insert into STUDENT (FIRST\_NAME, LAST\_NAME, SECTION, UNIVERSITY\_ID) values (?, ?, ?, ?)

Hibernate: select student0\_.STUDENT\_ID as STUDENT\_1\_0\_, student0\_.FIRST\_NAME as FIRST\_NA2\_0\_, student0\_.LAST\_NAME as LAST\_NAM3\_0\_, student0\_.SECTION as SECTION4\_0\_, student0\_.UNIVERSITY\_ID as UNIVERSI5\_0\_ from STUDENT student0\_

Student Details : Student [id=1, firstName=Sam, lastName=Disilva, section=Maths]

Student University Details: University [id=1, name=CAMBRIDGE, country=ENGLAND]

Student Details : Student [id=2, firstName=Joshua, lastName=Brill, section=Science]

Student University Details: University [id=1, name=CAMBRIDGE, country=ENGLAND]

Student Details : Student [id=3, firstName=Peter, lastName=Pan, section=Physics]

Student University Details: University [id=1, name=CAMBRIDGE, country=ENGLAND]

# Many-To-One Bidirectional:

One table has a foreign key column that references the primary key of associated table.In Bidirectional relationship, both side navigation is possible.

@Entity

@Table(name = "UNIVERSITY")

public class University {

    @Id

    @GeneratedValue

    @Column(name = "UNIVERSITY\_ID")

    private long id;

    @Column(name = "NAME")

    private String name;

    @Column(name = "COUNTRY")

    private String country;

    @OneToMany(mappedBy = "university", cascade = CascadeType.ALL)

    private List<Student> students;

//getters setters and constructors

**Student.java**

@Entity

@Table(name = "STUDENT")

public class Student {

    @Id

    @GeneratedValue

    @Column(name = "STUDENT\_ID")

    private long id;

    @Column(name = "FIRST\_NAME")

    private String firstName;

    @Column(name = "LAST\_NAME")

    private String lastName;

    @Column(name = "SECTION")

    private String section;

    @ManyToOne(optional = false)

    @JoinColumn(name = "UNIVERSITY\_ID")

    private University university;

**Intersesting facts :**

|  |
| --- |
| **@OneToMany(mappedBy = "university", cascade = CascadeType.ALL)**  **private List<Student> students;** |

@OneToMany on list property here denotes that one University can have multiple students.With students property defined in University class, we can now navigate from University to students. **mappedBy** **says that it’s the inverse side of relationship**. **Also note the cascade attribute, which means the dependent object(Student) will be persisted/updated/deleted automatically on subsequent persist/update/delete on University object. No need to perform operation separately on Student.**

On the other hand, we have following in Student

|  |
| --- |
| **@ManyToOne(optional = false)**  **@JoinColumn(name = "UNIVERSITY\_ID")**  **private University university;** |

**@JoinColumn** says that Student table will contain a separate column UNIVERSITY\_ID which will eventually act as a foreign key reference to primary key of University table. **@ManyToOne** says that multiple Student tuples can refer to same University Tuples(Multiple students can register in same university).Additionally , **with optional=false we make sure that no Student tuple can exist without a University tuple.**

**public class** HibernateStandAlone {  
  
 @SuppressWarnings(**"unchecked"**)  
 **public static void** main(String[] args) {  
  
 Student student1 = **new** Student(**"Sam"**, **"Disilva"**, **"Maths"**);  
 Student student2 = **new** Student(**"Joshua"**, **"Brill"**, **"Science"**);  
 Student student3 = **new** Student(**"Peter"**, **"Pan"**, **"Physics"**);  
  
 University university = **new** University(**"CAMBRIDGE"**, **"ENGLAND"**);  
 List<Student> allStudents = **new** ArrayList<Student>();  
  
 student1.setUniversity(university);  
 student2.setUniversity(university);  
 student3.setUniversity(university);  
  
 allStudents.add(student1);  
 allStudents.add(student2);  
 allStudents.add(student3);  
  
 university.setStudents(allStudents);  
  
 Session session = HibernateUtil.getSessionFactory().openSession();  
 session.beginTransaction();  
  
 session.persist(university);*// Students will be presisted automatically, thanks to CASCADE.ALL defined on students  
 // property of University class.* List<Student> students = (List<Student>) session.createQuery(  
 **"from Student "**).list();  
 **for** (Student s : students) {  
 System.***out***.println(**"Student Details : "** + s);  
 System.***out***.println(**"Student University Details: "** + s.getUniversity());  
 }  
  
 *// Note that now you can also access the relationship from University to Student* session.getTransaction().commit();  
 session.close();  
 }

Result :

Hibernate: insert into UNIVERSITY (COUNTRY, NAME) values (?, ?)

Hibernate: insert into STUDENT (FIRST\_NAME, LAST\_NAME, SECTION, UNIVERSITY\_ID) values (?, ?, ?, ?)

Hibernate: insert into STUDENT (FIRST\_NAME, LAST\_NAME, SECTION, UNIVERSITY\_ID) values (?, ?, ?, ?)

Hibernate: insert into STUDENT (FIRST\_NAME, LAST\_NAME, SECTION, UNIVERSITY\_ID) values (?, ?, ?, ?)

Hibernate: select student0\_.STUDENT\_ID as STUDENT\_1\_0\_, student0\_.FIRST\_NAME as FIRST\_NA2\_0\_, student0\_.LAST\_NAME as LAST\_NAM3\_0\_, student0\_.SECTION as SECTION4\_0\_, student0\_.UNIVERSITY\_ID as UNIVERSI5\_0\_ from STUDENT student0\_

Student Details : Student [id=1, firstName=Sam, lastName=Disilva, section=Maths]

Student University Details: University [id=1, name=CAMBRIDGE, country=ENGLAND]

Student Details : Student [id=2, firstName=Joshua, lastName=Brill, section=Science]

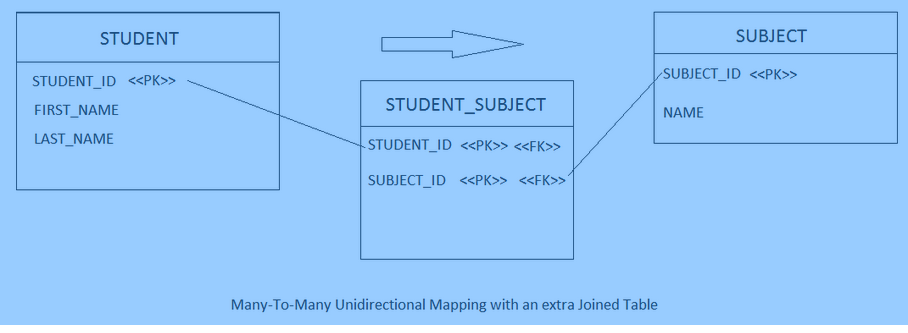
Student University Details: University [id=1, name=CAMBRIDGE, country=ENGLAND]

Student Details : Student [id=3, firstName=Peter, lastName=Pan, section=Physics]

Student University Details: University [id=1, name=CAMBRIDGE, country=ENGLAND]

# Many-To-Many Unidirectional

In Many-To-Many association, an extra table is used (known as Joined table) whose primary key is the combination of primary key of both the associated tables.In other words there is a foreign key association between the joined table and the associated tables.

****

We are discussing an example of Student and Subject relationship. A student can enroll for multiple subject. And a subject can have multiple students enrolled for it. We are considering Unidirectional mapping, means only Student to Subject entity navigation is possible.

#### Step 1: Create required Database Table

Open MySQL terminal / workbench terminal and execute following MySQL script :

|  |
| --- |
| create table STUDENT (     student\_id BIGINT NOT NULL AUTO\_INCREMENT,     first\_name VARCHAR(30) NOT NULL,     last\_name  VARCHAR(30) NOT NULL,     PRIMARY KEY (student\_id)  );  create table SUBJECT (     subject\_id BIGINT NOT NULL AUTO\_INCREMENT,     name VARCHAR(30) NOT NULL,     PRIMARY KEY (subject\_id)  );  CREATE TABLE STUDENT\_SUBJECT (      student\_id BIGINT NOT NULL,      subject\_id BIGINT NOT NULL,      PRIMARY KEY (student\_id, subject\_id),      CONSTRAINT FK\_STUDENT FOREIGN KEY (student\_id) REFERENCES STUDENT (student\_id),      CONSTRAINT FK\_SUBJECT FOREIGN KEY (subject\_id) REFERENCES SUBJECT (subject\_id)  ); |

Here we have first created the main tables STUDENT & SUBJECT. then we have created a joined table STUDENT\_SUBJECT whose primary key is the combination of primary keys of STUDENT & SUBJECT.

**Create Model classes:**

**Student.java**

@Entity

@Table(name = "STUDENT")

public class Student {

    @Id

    @GeneratedValue

    @Column(name = "STUDENT\_ID")

    private long id;

    @Column(name = "FIRST\_NAME")

    private String firstName;

    @Column(name = "LAST\_NAME")

    private String lastName;

    @ManyToMany(cascade = CascadeType.ALL)

    @JoinTable(name = "STUDENT\_SUBJECT",

             joinColumns = { @JoinColumn(name = "STUDENT\_ID") },

             inverseJoinColumns = { @JoinColumn(name = "SUBJECT\_ID") })

    private List<Subject> subjects = new ArrayList<Subject>();

**Subject.java:**

@Entity

@Table(name = "SUBJECT")

public class Subject {

    @Id

    @GeneratedValue

    @Column(name = "SUBJECT\_ID")

    private long id;

    @Column(name = "NAME")

    private String name;

Subject is a plain POJO, with no details of Student. On the other hand, in Student, we have defined following

|  |
| --- |
| **@ManyToMany(cascade = CascadeType.ALL)**  **@JoinTable(name = "STUDENT\_SUBJECT",**  **joinColumns = { @JoinColumn(name = "STUDENT\_ID") },**  **inverseJoinColumns = { @JoinColumn(name = "SUBJECT\_ID") })**  **private List<Subject> subjects = new ArrayList<Subject>();** |

@ManyToMany indicates that there is a Many-to-Many relationship between Student and subject. A Student can enroll for multiple subjects, and a subject can have multiple students enrolled. Notice **cascade = CascadeType.ALL**, with cascading while persisting (update/delete) Student tuples, subjects tuples will also be persisted (updated/deleted).

**@JoinTable** indicates that there is a link table which joins two tables via containing there keys. This annotation is mainly used on the owning side of the relationship. **joinColumns** refers to the column name of owning side(STUDENT\_ID of STUDENT), and **inverseJoinColumns** refers to the column of inverse side of relationship(SUBJECT\_ID of SUBJECT).Primary key of this joined table is combination of STUDENT\_ID & SUBJECT\_ID.

**One important remark : In case of \*Many\* association, always override hashcode and equals method which are looked by hibernate when holding entities into collections.**

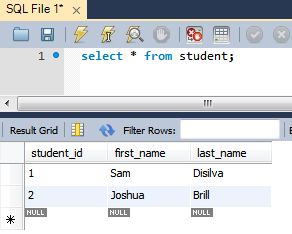
**Main file :**

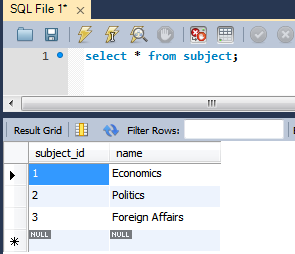
**public class** HibernateStandAlone {  
  
 **public static void** main(String[] args) {  
 Student student1 = **new** Student(**"Sam"**,**"Disilva"**);  
 Student student2 = **new** Student(**"Joshua"**, **"Brill"**);  
 Subject subject1 = **new** Subject(**"Economics"**);  
 Subject subject2 = **new** Subject(**"Politics"**);  
 Subject subject3 = **new** Subject(**"Foreign Affairs"**);  
   
 *//Student1 have 3 subjects* student1.getSubjects().add(subject1);  
 student1.getSubjects().add(subject2);  
 student1.getSubjects().add(subject3);  
 *//Student2 have 2 subjects* student2.getSubjects().add(subject1);  
 student2.getSubjects().add(subject2);

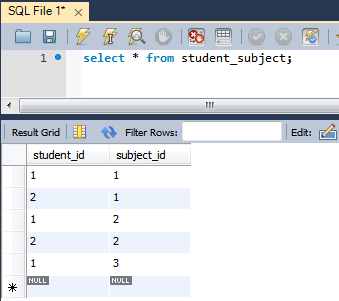
Session session = HibernateUtil.getSessionFactory().openSession();  
 session.beginTransaction();  
 session.persist(student1);  
 session.persist(student2);  
 session.getTransaction().commit();  
 session.close();  
 }

Here you can see that we have set the subjects property of Student class, and just persisted Student objects.Thanks to Cascade attribute, Subject tubles will be persisted automatically. And due to the annotation mapping we have done above in student class, a new tuple will be created in joined table (STUDENT\_SUBJECT) for each combination of student & subject item we dealt with in main program.

Below is the snapshot of MySQL database after execution of above program.







That’s it.

# Many-To-Many Bidirectional

**Same as Unidirectional only changes in**

**Student.java :**

@Entity  
@Table(name = **"STUDENT"**)  
**public class** Student {  
  
 @Id  
 @GeneratedValue  
 @Column(name = **"STUDENT\_ID"**)  
 **private long id**;  
  
 @Column(name = **"FIRST\_NAME"**)  
 **private** String **firstName**;  
  
 @Column(name = **"LAST\_NAME"**)  
 **private** String **lastName**;  
  
 @ManyToMany(cascade = CascadeType.ALL)  
 @JoinTable(name = **"STUDENT\_SUBJECT"**,  
 joinColumns = { @JoinColumn(name = **"STUDENT\_ID"**) },  
 inverseJoinColumns = { @JoinColumn(name = **"SUBJECT\_ID"**) })  
 **private** List<Subject> **subjects** = **new** ArrayList<Subject>();  
  
  
}

In Subject.java

@Entity  
@Table(name = **"SUBJECT"**)  
**public class** Subject {  
  
 @Id  
 @GeneratedValue  
 @Column(name = **"SUBJECT\_ID"**)  
 **private long id**;  
  
 @Column(name = **"NAME"**)  
 **private** String **name**;  
  
  
 @ManyToMany(mappedBy=**"subjects"**)  
 **private** List<Student> **students** = **new** ArrayList<Student>();

Only change in this relationship( ManyToMany Bidirectional) and [ManyToMany Unidirectional](http://websystique.com/hibernate/hibernate-many-to-many-unidirectional-annotation-example/) is that, in the Subject class we have added following property.

|  |
| --- |
| @ManyToMany(mappedBy="subjects")  private List<Student> students = new ArrayList<Student>(); |

Nothing else changes. We added this property in Subject class to make the relationship bidirectional. You can now navigate from Subject to Student. **mappedBy** attribute tells that this is the inverse side of relationship which is managed by “subjects” property of Student annotated with **@JoinColumn**.