# Hibernate with Annotation

The hibernate application can be created with annotation. There are many annotations that can be used to create hibernate application such as **@Entity, @Id, @Table** etc.

The core advantage of using hibernate annotation is that you don't need to create mapping (hbm) file. Here, hibernate annotations are used to provide the meta data.

### Create the Persistent class

Here, we are creating the same persistent class which we have created in the previous topic. But here, we are using annotation.

**@Entity** annotation marks this class as an entity.

**@Transient** If we don’t any variable in DB

**@Table** annotation specifies the table name where data of this entity is to be persisted. If you don't use @Table annotation, hibernate will use the class name as the table name by default.

**@Id** annotation marks the identifier for this entity.

**@Column** annotation specifies the details of the column for this property or field. If **@Column** annotation is not specified, property name will be used as the column name by default.

### *Student.java*

@Entity

@Table(name = "stu")

**public** **class** Student {

@Id

@GeneratedValue(strategy = GenerationType.***AUTO***)

**int** sid;

@Column(name = "Name")

String sName;

@Column

**int** fee;

@Transient **// If we dont want to store any value in DB**

String hobby;

}

**@Entity(name = "*emp*")**

When we pass **name = “*emp*”** it will consider your className will be emp So ClassName Employee will be emp So in HQL we need to pass query = “from emp” and no need to define **@Table** also

Add mapping of Persistent class in configuration file

**open the hibernate.cgf.xml file, and add an entry of mapping resource like this:**

<mapping class="p1.Student"></mapping>

### *Client.java*

Insert, Update, Delete will be same as previous

# Component Mapping with Annotation

### *Employee.java*

@Id

@GeneratedValue(strategy = GenerationType.***AUTO***)

**int** eid;

@Column

String eName;

Address add; //Getter and Setter

### *Address.java*

**@Embeddable** // For Component Mapping....

**public** **class** Address {We can add **@Embedded** also above Address reference in Employee class

**int** pinCode;

String city;

**Caching in Hibernate**

Hibernate caching improves the performance of the application by pooling the object in the cache. There are mainly two types of caching: first level cache and second level cache.

### *First Level Cache*

Session object holds the first level cache data. It is enabled by default. The first level cache data will not be available to entire application. An application can use many sessions object.

**Ex.... Fetch any Record from DB two time by using load or get method**

Employee e = (Employee)s.get(Employee.**class**, 1);

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

Employee e1 = (Employee)s.get(Employee.**class**, 1); System.***out***.println(e1);

**So for this when we see console only one time select query is coming and two Records are there....**

**But what if we have 2 different session object like......**

Employee e = (Employee)**s1**.get(Employee.**class**, 1);

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

Employee e1 = (Employee)**s2**.get(Employee.**class**, 1); System.***out***.println(e1);

**Then it will again hit to database two time. So, to resolve this issue we can use Second level Cache...**

#### Second Level Cache

#### SessionFactory object holds the second level cache data. The data stored in the second level cache will be available to entire application. But we need to enable it explicitly.

So, its mean it does not matter how many sessions object you have all can share Second Level Cache... So now for this it will hit DB only once

Employee e = (Employee)**s1**.get(Employee.**class**, 1);

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

Employee e1 = (Employee)**s2**.get(Employee.**class**, 1); System.***out***.println(e1);

***Let see how it is.........***

So, we have to configured SLC for that we have to use some third-party provider. we have like

**EHCache** OR **OSCache** OR **Swarm Cache** ETC........But in those **EHCache** is More getting use NowAday.

**Now we required some jar files..**

* 1. ehcache-2.10.6.jar (For Features.......)
  2. hibernate-ehcache-5.3.4.Final.jar (For Integrations.......)

**NOTE For this example our Hibernate Version and hibernate-ehcache jar version Must SAME.**

**NOTE In Hibernate 3.6, “org.hibernate.cfg.AnnotationConfiguration” is deprecated, and all its functionality has been moved to “org.hibernate.cfg.Configuration“.**

***Employee.class***

@Entity

@Table(name = "Emp")

**/\*To Enable Cache in Current class\*/**

@Cache(usage = CacheConcurrencyStrategy.***READ\_ONLY***)

**public** **class** Employee {

@Id

@GeneratedValue(strategy = GenerationType.***IDENTITY***)

**int** eid;

@Column

String eName; // Getter and Setter

***Hibernate.cfg.xml***

<property name = *"hibernate.cache.use\_second\_level\_cache"*>true</property>

<property name=*"hibernate.cache.provider\_class"*>

org.hibernate.cache.EhCacheProvider</property>

<property name = *"hibernate.cache.region.factory\_class"*>

org.hibernate.cache.ehcache.internal.EhcacheRegionFactory</property>

***Client.java***

Session s = sf.openSession();

Employee e = (Employee)s.get(Employee.**class**, 1);

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

s.close();

Session s1 = sf.openSession();

Employee e1 = (Employee)s1.get(Employee.**class**, 1);

System.***out***.println(e1);

s1.close();

**We will get Select Query only once means second time its fetching record from SLC**

***So now what if we are using Query interface in get or load method to fetch the Records***

Session s = sf.openSession();

Query q1 = s.createQuery("from Employee where eid = 1");

Employee e1 = (Employee)q1.uniqueResult();

System.***out***.println(e1);

s.close();

Session s2 = sf.openSession();

Query q2 = s2.createQuery("from Employee where eid = 1");

Employee e2 = (Employee)q2.uniqueResult();// If only 1 Records Coming from DB

System.***out***.println(e2);

s2.close();

**Again it wil hit the DB 2 time so to resolve this issue So byDefault SLC is available to load or get method only we have to make it enable for Query Interface also..**

***Hibernate.cfg.xml***

<property name = *"hibernate.cache.use\_query\_cache"*>true</property>

***Add this After that 3 Property***

***Client.java***

q1.setCacheable(**true**);

For **Q1** and **Q2** both we have to set, If we are getting value via passing parameter so that: n which we pass instead of ? that should be same, means name :n also same

**Hikari Connection Pool**

1. **HikariCP-2.6.1.jar**
2. **hibernate-hikaricp-5.3.4.Final.jar**

***Hibernate.cfg.xml***

<property name=*"hibernate.hikari.connectionTimeout"*>300000</property>

<property name=*"hibernate.hikari.minimumIdle"*>10</property>

<property name=*"hibernate.hikari.maximumPoolSize"*>6</property>

<property name=*"hibernate.hikari.idleTimeout"*>300000</property>

**Named query Example**

In Hibernate, a named query is a JPQL or SQL expression with predefined unchangeable query string. You can define a named query either in hibernate mapping file or in an entity class.

Annotations **@NamedQuery** and **@NamedQueries** are used for **HQL** or **JPQL** expression whereas **@NamedNativeQuery** and **@NamedNativeQueries** are used for native **SQL** expression.

**1 )** **@NamedQuery Example**

@NamedQuery(name = "**getAll**",query = "**from Student**")

Add this on top of pojo class below **@Table Annotation**

**FetchData.java**

StandardServiceRegistry ssr = **new** StandardServiceRegistryBuilder().configure("Hibernate.cfg.xml").build();

MetadataSources mds = **new** MetadataSources(ssr);

Metadata md = mds.getMetadataBuilder().build();

SessionFactory sf = md.getSessionFactoryBuilder().build();

Session s = sf.openSession();

Query<Student> q = s.createNamedQuery("getAll",Student.**class**);

List<Student> li = q.list();

System.***out***.println(li);

with Where condition

@NamedQuery(name = "**getAll**",query = "**from Student where sid = :x and roll = :y**")

**FetchData.java**

Query<Student> q =

s.createNamedQuery("getAll",Student.**class**).setParameter("x", 1).setParameter("y", 30);

List<Student> li = q.list();

System.***out***.println(li);

**2 ) @NamedQueries Example**

If we want to define multiple **@NamedQuery** for single pojo class so for that we can use **@NamedQueries**

**@NamedQueries**({ @NamedQuery(name = "**getAll**", query = "**from Student**"),

@NamedQuery(name = "**getIdCount**", query = "**select count(sid) from Student**") })

Add this on top of pojo class below **@Table Annotation**

**FetchData.java**

Will be same as above we need to call **s.createNamedQuery** seperate for all **@NamedQuery**

**3 ) @NamedNativeQuery Example**

So till now we have create query which can be understand by java HBM like for **select \* from stu** we have use **from Student** etc.

So **Select \* from stu** is **MySQL** code and its **native** for **java** and if we want to use this Native code in java we can go with **@NamedNativeQuery**

**@NamedNativeQuery**(name = "**getAll**",query = "**Select \* from s**",resultClass = Student.**class**)

**FetchData.java**

Query<Student> q = s.createNamedQuery("getAll",Student.**class**);

List<Student> li = q.list();

System.*out*.println("GetAll " + li);

**4 ) @NamedNativeQueries Example**

If we want to define multiple **@NamedNativeQuery** for single pojo class so for that we can use **@NamedNativeQueries**

**@NamedNativeQueries({**

@**NamedNativeQuery**(name = "**getAll**",query = "**Select \* from s**",resultClass = Student.**class**),

@**NamedNativeQuery**(name = "**getIdCount**",query = "**Select count(1) from s**") **})**

Add this on top of pojo class below **@Table Annotation**

**FetchData.java**

Query<Student> q = s.createNamedQuery("getAll",Student.class);

List<Student> li = q.list();

System.***out***.println("GetAll " + li);

Query<Object> q1 = s.createNamedQuery("getIdCount");

List<Object> li1 = q1.list();

System.*out*.println(li1);

**5 ) @NamedStoredProcedureQuery Example**

mysql> delimiter //

mysql> create procedure getData(id int)

-> begin

-> select \* from s where sid = id;

-> end;

-> //

Add this on top of pojo class below **@Table Annotation**

**@NamedStoredProcedureQuery(**

name = "**getAll**",

procedureName = "**getData**",

parameters = {

**@StoredProcedureParameter(**name = "**id**",mode = ParameterMode.***IN***,type = Integer.**class**)

**},**

resultClasses = Student.**class**

)

**FetchData.java**

StoredProcedureQuery sp = s.createNamedStoredProcedureQuery("getAll");

sp.setParameter("id", 1);

System.***out***.println(sp.getSingleResult());

**6 ) @NamedStoredProcedureQueries Example**

Add this on top of pojo class below **@Table Annotation**

**@NamedStoredProcedureQueries**({

@NamedStoredProcedureQuery(

name = "**getA**",

procedureName = "**getData**",

parameters = {

@StoredProcedureParameter(name = "**id**",mode = ParameterMode.***IN***,type =Integer.**class**)

},

resultClasses = Student.**class**

),

@NamedStoredProcedureQuery(

name = "**getB**",

procedureName = "**getData**",

parameters = {

@StoredProcedureParameter(name = "**id**",mode = ParameterMode.***IN***,type = Integer.**class**)

},

resultClasses = Student.**class**

)

})

To getOutPut Parameter if any

Long data = (Long)sp.getOutputParameterValue("pco");

**FetchData.java**

**Calling will be same as Above**

6) @Immutable Example

**@Immutable** annotation tells Hibernate is that any updates to an immutable entity should not be passed on to the database without giving any error. **@Immutable** can be placed on a **collection too**; in this case, changes to the collection (additions, or removals) will cause a **HibernateException** to be thrown.

Add this on top of pojo class above **@Entity** Annotation

**1)** First add one record

**2)** Fetch that same record by using get, load or byId method and update some column and call **s.update()** on it and commit.

**3)** You will see changes not reflect in DB

7) @NaturalId Example

We learned so much things in past tutorials about **@Id** annotation with **@GeneratedValue** to create primary keys for records in database. In most real-life applications, these primary keys are **“artificial primary keys”** and referred only inside application runtime. However, there’s also the concept of a **“natural ID “**, which provides another convenient and logical way to refer to an entity, apart from an artificial or **composite primary key**.

**An example of natural id might be a Social Security number or a Tax Identification Number in the United States, and PAN number in India.** An entity (being a person or a corporation) might have an artificial primary key generated by Hibernate, but it also might have a unique tax identifier. Hibernate allows you to search and load entities based on these natural ids as well.

**For natural IDs,** there are two forms of load mechanisms; one uses the simple natural ID **(where the natural ID is one and only one field),** and the other uses **named attributes as part of a composite natural ID.**

Add this inside pojo class above int roll;

Its internally works as a Unique key in MySQL.

**FetchData.java**

Student s2 = ss.byNaturalId(Student.**class**).using("roll", 40).load();

8) **@MappedSuperclass Example**

@MappedSuperclass

**public** **class** BaseEntity **implements** Serializable {

@Id

@GeneratedValue(strategy = GenerationType.***IDENTITY***)

Integer id;

}

@Entity

@Table(name = "e")

**public** **class** Employee **extends** BaseEntity{

String eName;

}

We no need to define **@Id** now because it inherits from there whatever column is common amongst Entity, we can define in BaseEntity

**Hibernate Relation with Annotation**

**One to Many with String class**

Question.java

@Entity

@Table(name = "one")

**public** **class** Question {

@Id

@GeneratedValue(strategy = GenerationType.***IDENTITY***)

**int** qid;

@Column

String qname;

@ElementCollection

@CollectionTable(name = ***"ans***")

List<String> ans; Getter and Setter

**One to Many with Entity class**

Question.java

@Entity

@Table(name = "one")

**public** **class** Question {

@Id

@GeneratedValue(strategy = GenerationType.***IDENTITY***)

**int** qid;

@Column

String qname;

@OneToMany(cascade = CascadeType.***PERSIST***)

List<Answer> ans; Getter and Setter

Answer.java

@Entity

@Table(name = "two")

**public** **class** Answer {

@Id

@GeneratedValue(strategy=GenerationType.***IDENTITY***)

**int** aid;

@Column

String answer;

@Column

String postedby; Getter and Setter

Client Answer a = **new** Answer();

a.setAnswer("Language");

a.setPostedby("Pam");

Answer a1 = **new** Answer();

a1.setAnswer("Program");

a1.setPostedby("Harry");

List<Answer> li = **new** ArrayList<Answer>();

li.add(a1);

li.add(a);

Question q = **new** Question();

q.setQid(1);

q.setQname("Java");

q.setAns(li);

**s.save(q);**

It will create seperate table by combining both table name and show the linked on that table

**Now Suppose if we want to change the table and column name which its gonna created byDefault so add this below @OneToMany Annotation.**

@**JoinTable**(**name** = "ot",**joinColumns** = **@JoinColumn**(name = "qid"),

**inverseJoinColumns** = **@JoinColumn**(name = "aid"))

**ot is Table name , qid and aid is new column name**

**OR Else**

**If we dont want seperate table we want combination like we are getting XML way we need to use**

@OneToMany(cascade = CascadeType.***ALL***,fetch = FetchType.***LAZY***)

@JoinColumn(name = "j" ,referencedColumnName = "qid")

List<Answer> ans;

Now **FetchType** can be ***LAZY*** and ***EAGER***, if we don’t specify any it will be by default ***LAZY*** means if we have **OneToMany** like one **Question** and **List<String> ans** inside that **Question** So for that it will create **two table** 1st for Question and 2nd List<String>

So suppose I am fetching record like **q.getQName()** only So it will not fetch the ans related to that Question from 2nd table if we set **FetchType**.***LAZY*** **HBM** think we don’t required ans Once we use that ans like Printing then only it will fetch like **get()** and **load()** diff. If we want both at same time then we can use ***EAGER***

**Many to One with Annotation ( UniDirectional )**

Consider **One to Many with Annotation** on that we have One Question and Multiple Answer for that Question so it will be One to Many from Question side and Many to One from Answer side. But in this it will be from Answer side only not from Question side.

Question.java

@Entity

@Table(name = "one")

**public** **class** Question {

@Id

@GeneratedValue(strategy = GenerationType.***IDENTITY***)

**int** qid;

@Column

String qname;

Answer.java

@Entity

@Table(name = "two")

**public** **class** Answer {

@Id

@GeneratedValue(strategy=GenerationType.***IDENTITY***)

**int** aid;

@Column

String answer;

@Column

String postedby;

@ManyToOne(cascade = CascadeType.***ALL***)

@JoinColumn(name = "j",referencedColumnName = "qid")

Question q; **Getter and Setter**

Client Question q = **new** Question();

q.setQid(1);

q.setQname("Java");

Answer a = **new** Answer();

a.setAnswer("Language");

a.setPostedby("Pam");

a.setQ(q);

Answer a1 = **new** Answer();

a1.setAnswer("Program");

a1.setPostedby("Harry");

a1.setQ(q);

**s.save(a); s.save(a1); s.save(q);**

**if we dont want to save Question we can use cascade in Answer class**

**Many to One with Annotation ( BiDirectional )**

Consider **One to Many with Annotation** on that we have One Question and Multiple Answer for that Question so it will be One to Many from Question side and Many to One from Answer side. So this time **Question is giving One to Many RelationShip** and **Answer is Giving Many to One RelationShip**

Question.java

@Entity

@Table(name = "one")

**public** **class** Question {

@Id

@GeneratedValue(strategy = GenerationType.***IDENTITY***)

**int** qid;

@Column

String qname;

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

List<Answer> ans; Getter and Setter

Answer.java

@Entity

@Table(name = "two")

**public** **class** Answer {

@Id

@GeneratedValue(strategy=GenerationType.***IDENTITY***)

**int** aid;

@Column

String answer;

@Column

String postedby;

@ManyToOne(optional = **false**)

@JoinColumn(name = "qis")

Question q; [ optional = **false** ]  we make sure that no Answer tuple can exist without a Quesion tuple. }

Client Question q = **new** Question();

q.setqName("What is Java");

Answer a1 = **new** Answer();

a1.setaName("Pro");

a1.setQ(q);

Answer a2 = **new** Answer();

a2.setaName("Lang");

**a2.setQ(q);**

q.setAns(Arrays.*asList*(a1,a2));

**s.save(q);**

s.beginTransaction().commit();

**One to One with Annotation ( UniDirectional )**

Now Suppose we have Question and Answer related to that Question So UniDirectional we can get data from Question Side only not from Answer side means Association is from Question side not from Answer

Question.java

@Entity

@Table(name = "one")

**public** **class** Question {

@Id

@GeneratedValue(strategy = GenerationType.***IDENTITY***)

**int** qid;

@Column

String qname;

@OneToOne

@Cascade(value = CascadeType.***ALL***)

@PrimaryKeyJoinColumn

Answer ans; Getter and Setter

Answer.java

@Entity

@Table(name = "two")

**public** **class** Answer {

@Id

@GeneratedValue(strategy = GenerationType.**IDENTITY**)

**int** aid;

@Column

String answer;

@Column

String postedby;

}

Client Answer a = **new** Answer();

a.setAnswer("Language");

a.setPostedby("Pam");

Answer a1 = **new** Answer();

a1.setAnswer("Program");

a1.setPostedby("Harry");

Question q = **new** Question();

q.setQid(1);

q.setQname("Java");

q.setAns(a);

q.setAns(a1);

**s.save(q);**

FetchData.java

ServiceRegistry ssr = **new** StandardServiceRegistryBuilder().configure().build();

MetadataSources mds = **new** MetadataSources(ssr);

Metadata md = mds.getMetadataBuilder().build();

SessionFactory sf = md.getSessionFactoryBuilder().build();

Session s = sf.openSession();

Question q = s.load(Question.**class**, 1);

System.***out***.println(q.getQ().getQid());

System.***out***.println(q.getQ().getQname());

**System.*out*.println(q.getAid());**

**System.*out*.println(q.getAnswer());**

**We cannot call Answer directly we need to call Answer through Question class.**

**One to One with Annotation ( BiDirectional )**

Now Suppose we have One Question and One Answer for that Question means One Answer for one Question and one Question for One Answer So if i call Answer Hbm should tell me that Answer is related to which Question and if i call Question Hbm should tell me associate answer of it

Question.java

@Entity

@Table(name = "one")

**public** **class** Question {

@Id

@GeneratedValue(strategy = GenerationType.***IDENTITY***)

**int** qid;

@Column

String qname;

@OneToOne(mappedBy = "q")

@Cascade(value = CascadeType.***ALL***)

Answer ans; **Getter and Setter**

Answer.java

@Entity

@Table(name = "two")

**public** **class** Answer {

@Id

@Column(name="a\_id", unique=**true**, nullable=**false**)

@GeneratedValue(generator="gen")

@GenericGenerator(name = "gen", strategy = "foreign",

parameters = @Parameter(name="property", value="q"))

**int** aid;

@Column

String answer;

@Column

String postedby;

@OneToOne

@PrimaryKeyJoinColumn

Question q;  **Getter and Setter**

Client Answer a = **new** Answer();

a.setAnswer("Language");

a.setPostedby("Pam");

Answer a1 = **new** Answer();

a1.setAnswer("Program");

a1.setPostedby("Harry");

List<Answer> li = **new** ArrayList<Answer>();

li.add(a1);

li.add(a);

Question q = **new** Question();

q.setQid(1);

q.setQname("Java");

q.setAns(li);

**s.save(a); s.save(a1); s.save(q);**

FetchData.java

ServiceRegistry ssr = **new** StandardServiceRegistryBuilder().configure().build();

MetadataSources mds = **new** MetadataSources(ssr);

Metadata md = mds.getMetadataBuilder().build();

SessionFactory sf = md.getSessionFactoryBuilder().build();

Session s = sf.openSession();

Answer a = s.load(Answer.**class**, 1);

System.***out***.println(a.getQ().getQid());

System.***out***.println(a.getQ().getQname());

System.***out***.println(a.getAid());

System.***out***.println(a.getAnswer());

**So we are calling Answer so we are getting Question Also and if we call Question then we get Answer also that we call BiDirectional One-To-One**

# Hibernate Inheritance Mapping with Annotations

# 1) Hibernate Table Per Hierarchy using Annotation

 You need to use **@Inheritance(strategy=InheritanceType.SINGLE\_TABLE),**

**@DiscriminatorColumn** and **@DiscriminatorValue** annotations for mapping table per hierarchy strategy.In case of table per hierarchy, only one table is required to map the inheritance hierarchy. Here, an extra column (also known as discriminator column) is created in the table to identify the class.

***Employee.java***

@Entity

@Table(name = "employee101")

@Inheritance(strategy=InheritanceType.SINGLE\_TABLE)

@DiscriminatorColumn(name="type",discriminatorType=DiscriminatorType.STRING)

@DiscriminatorValue(value="employee")

**public** **class** Employee {

@Id

@GeneratedValue(strategy=GenerationType.AUTO)

@Column(name = "id")

**private** **int** id;

@Column(name = "name")

**private** String name;

//setters and getters

*Regular\_Employee.java*

@Entity

@DiscriminatorValue("regularemployee")

**public** **class** Regular\_Employee **extends** Employee{

@Column(name="salary")

**private** **float** salary;

@Column(name="bonus")

**private** **int** bonus;

//setters and getters

**Add this file like this**

        <mapping **class**="com.javatpoint.mypackage.Employee"/>

        <mapping **class**="com.javatpoint.mypackage.Contract\_Employee"/> Inside Session-Factory Tag

Table Per Concrete class using Annotation

In case of Table Per Concrete class, tables are created per class. So, there are no nullable values in the table. Disadvantage of this approach is that duplicate columns are created in the subclass tables.

Here, we need to use @Inheritance(strategy = InheritanceType.TABLE\_PER\_CLASS) annotation in the parent class and @AttributeOverrides annotation in the subclasses.

**@Inheritance(strategy = InheritanceType.TABLE\_PER\_CLASS)** specifies that we are using table per concrete class strategy. It should be specified in the parent class only.

**@AttributeOverrides** defines that parent class attributes will be overriden in this class. In table structure, parent class table columns will be added in the subclass table.

*Employee.java*

**package** com.javatpoint.mypackage;

**import** javax.persistence.\*;

@Entity

@Table(name = "employee102")

@Inheritance(strategy = InheritanceType.TABLE\_PER\_CLASS)

**public** **class** Employee {

@Id

@GeneratedValue(strategy=GenerationType.AUTO)

@Column(name = "id")

**private** **int** id;

@Column(name = "name")

**private** String name;

//setters and getters

}

*Regular\_Employee.java*

**package** com.javatpoint.mypackage;

**import** javax.persistence.\*;

@Entity

@Table(name="regularemployee102")

@AttributeOverrides({

    @AttributeOverride(name="id", column=@Column(name="id")),

    @AttributeOverride(name="name", column=@Column(name="name"))

})

**public** **class** Regular\_Employee **extends** Employee{

@Column(name="salary")

**private** **float** salary;

@Column(name="bonus")

**private** **int** bonus;

//setters and getters

}

# Table Per Subclass using Annotation

As we have specified earlier, in case of table per subclass strategy, tables are created as per persistent classes but they are related using primary and foreign key. So, there will not be duplicate columns in the relation.

We need to specify **@Inheritance (strategy=InheritanceType.JOINED)** in the parent class and **@PrimaryKeyJoinColumn** annotation in the subclasses.

*Employee.java*

**package** com.javatpoint.mypackage;

**import** javax.persistence.\*;

@Entity

@Table(name = "employee103")

@Inheritance(strategy=InheritanceType.JOINED)

**public** **class** Employee {

@Id

@GeneratedValue(strategy=GenerationType.AUTO)

@Column(name = "id")

**private** **int** id;

@Column(name = "name")

**private** String name;

//setters and getters

}

*Regular\_Employee.java*

**package** com.javatpoint.mypackage;

**import** javax.persistence.\*;

@Entity

@Table(name="regularemployee103")

@PrimaryKeyJoinColumn(name="ID")

**public** **class** Regular\_Employee **extends** Employee{

@Column(name="salary")

**private** **float** salary;

@Column(name="bonus")

**private** **int** bonus;