Hibernate Inheritance Mapping

We can map the inheritance hierarchy classes with the table of the database. There are three inheritance mapping strategies defined in the hibernate:

1. Table Per Hierarchy
2. Table Per Concrete class
3. Table Per Subclass

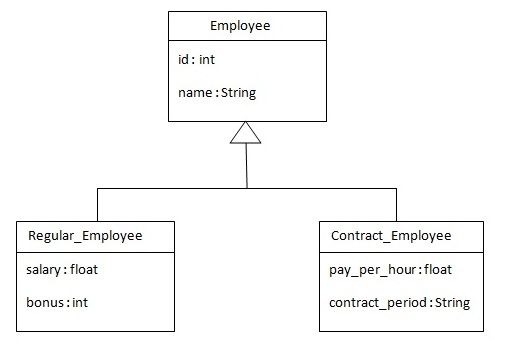
# Hibernate Table Per Hierarchy using Annotation

1. [Table Per Hierarchy](https://www.javatpoint.com/hibernate-table-per-hierarchy-using-annotation-tutorial-example)
2. [Example of Table Per Hierarchy](https://www.javatpoint.com/hibernate-table-per-hierarchy-using-annotation-tutorial-example#ex)

In the previous page, we have mapped the inheritance hierarchy with one table only using xml file. Here, we are going to perform this task using annotation. You need to use @Inheritance(strategy=InheritanceType.SINGLE\_TABLE), @DiscriminatorColumn and @DiscriminatorValue annotations for mapping table per hierarchy strategy.

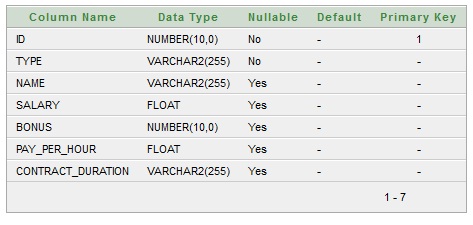
In case of table per hierarchy, only one table is required to map the inheritance hierarchy. Here, an extra column (also known as **discriminator column**) is created in the table to identify the class.

Let's see the inheritance hierarchy:



There are three classes in this hierarchy. Employee is the super class for Regular\_Employee and Contract\_Employee classes.

|  |
| --- |
| The table structure for this hierarchy is as shown below: |



### **Example of Hibernate Table Per Hierarchy using Annotation**

You need to follow following steps to create simple example:

* Create the persistent classes
* Create the configuration file
* Create the class to store the fetch the data

### **1) Create the Persistent classes**

You need to create the persistent classes representing the inheritance. Let's create the three classes for the above hierarchy:

*File: Employee.java*

1. **package** com.javatpoint.mypackage;
2. **import** javax.persistence.\*;
4. @Entity
5. @Table(name = "employee101")
6. @Inheritance(strategy=InheritanceType.SINGLE\_TABLE)
7. @DiscriminatorColumn(name="type",discriminatorType=DiscriminatorType.STRING)
8. @DiscriminatorValue(value="employee")
10. **public** **class** Employee {
11. @Id
12. @GeneratedValue(strategy=GenerationType.AUTO)
14. @Column(name = "id")
15. **private** **int** id;
17. @Column(name = "name")
18. **private** String name;
20. //setters and getters
21. }

*File: Regular\_Employee.java*

1. **package** com.javatpoint.mypackage;
3. **import** javax.persistence.\*;
5. @Entity
6. @DiscriminatorValue("regularemployee")
7. **public** **class** Regular\_Employee **extends** Employee{
9. @Column(name="salary")
10. **private** **float** salary;
12. @Column(name="bonus")
13. **private** **int** bonus;
15. //setters and getters
16. }

*File: Contract\_Employee.java*

1. **package** com.javatpoint.mypackage;
3. **import** javax.persistence.Column;
4. **import** javax.persistence.DiscriminatorValue;
5. **import** javax.persistence.Entity;
7. @Entity
8. @DiscriminatorValue("contractemployee")
9. **public** **class** Contract\_Employee **extends** Employee{
11. @Column(name="pay\_per\_hour")
12. **private** **float** pay\_per\_hour;
14. @Column(name="contract\_duration")
15. **private** String contract\_duration;
17. //setters and getters
18. }

### **2) Add the persistent classes in configuration file**

|  |
| --- |
| Open the hibernate.cgf.xml file, and add entries of entity classes like this: |

1. <mapping **class**="com.javatpoint.mypackage.Employee"/>
2. <mapping **class**="com.javatpoint.mypackage.Contract\_Employee"/>
3. <mapping **class**="com.javatpoint.mypackage.Regular\_Employee"/>
4. </pre></div>
5. <table >
6. <tr><td>Now the configuration file will look like **this**:
7. </td></tr>
8. </table>
9. <span id="filename">File: hibernate.cfg.xml</span>
10. <div **class**="codeblock"><pre name="code" **class**="java" >
11. <?xml version='1.0' encoding='UTF-8'?>
12. <!DOCTYPE hibernate-configuration PUBLIC
13. "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
14. "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
16. <!-- Generated by MyEclipse Hibernate Tools.                   -->
17. <hibernate-configuration>
19. <session-factory>
20. <property name="hbm2ddl.auto">update</property>
21. <property name="dialect">org.hibernate.dialect.Oracle9Dialect</property>
22. <property name="connection.url">jdbc:oracle:thin:@localhost:1521:xe</property>
23. <property name="connection.username">system</property>
24. <property name="connection.password">oracle</property>
25. <property name="connection.driver\_class">oracle.jdbc.driver.OracleDriver</property>
27. <mapping **class**="com.javatpoint.mypackage.Employee"/>
28. <mapping **class**="com.javatpoint.mypackage.Contract\_Employee"/>
29. <mapping **class**="com.javatpoint.mypackage.Regular\_Employee"/>
30. </session-factory>
32. </hibernate-configuration>

The hbm2ddl.auto property is defined for creating automatic table in the database.

### **3) Create the class that stores the persistent object**

In this class, we are simply storing the employee objects in the database.

*File: StoreTest.java*

1. **package** com.javatpoint.mypackage;
3. **import** org.hibernate.\*;
4. **import** org.hibernate.cfg.\*;
6. **public** **class** StoreData {
7. **public** **static** **void** main(String[] args) {
8. AnnotationConfiguration cfg=**new** AnnotationConfiguration();
9. Session session=cfg.configure("hibernate.cfg.xml").buildSessionFactory().openSession();
11. Transaction t=session.beginTransaction();
13. Employee e1=**new** Employee();
14. e1.setName("sonoo");
16. Regular\_Employee e2=**new** Regular\_Employee();
17. e2.setName("Vivek Kumar");
18. e2.setSalary(50000);
19. e2.setBonus(5);
21. Contract\_Employee e3=**new** Contract\_Employee();
22. e3.setName("Arjun Kumar");
23. e3.setPay\_per\_hour(1000);
24. e3.setContract\_duration("15 hours");
26. session.persist(e1);
27. session.persist(e2);
28. session.persist(e3);
30. t.commit();
31. session.close();
32. System.out.println("success");
33. }
34. }

#### **Output:**

# output of the table per class hierarchyTable Per Concrete class using Annotation

1. [Table Per Concrete class using annotation](https://www.javatpoint.com/hibernate-table-per-concrete-class-using-annotation-tutorial-example)

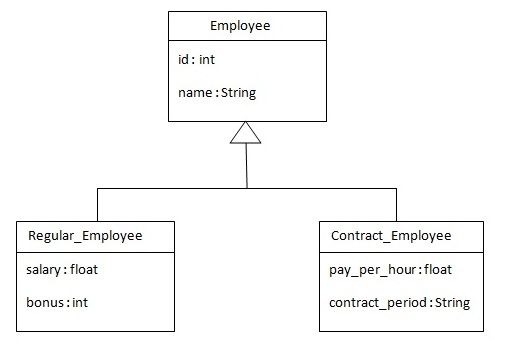
In case of Table Per Concrete class, tables are created per class. So there are no nullable values in the table. Disadvantage of this approach is that duplicate columns are created in the subclass tables.

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.

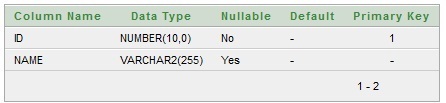
**@AttributeOverrides** defines that parent class attributes will be overriden in this class. In table structure, parent class table columns will be added in the subclass table.

The class hierarchy is given below:

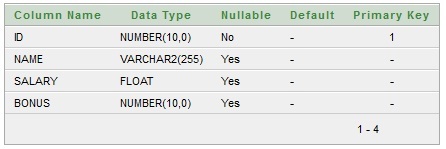


|  |
| --- |
| The table structure for each table will be as follows: |

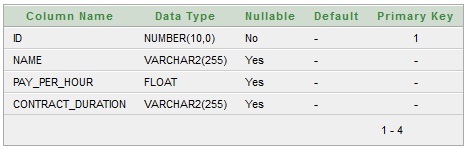
#### **Table structure for Employee class**

#### **Table structure for Regular\_Employee class**

#### **Table structure for Contract\_Employee class**



### **Example of Table per concrete class**

In this example we are creating the three classes and provide mapping of these classes in the employee.hbm.xml file.

### **1) Create the Persistent classes**

You need to create the persistent classes representing the inheritance. Let's create the three classes for the above hierarchy:

*File: Employee.java*

1. **package** com.javatpoint.mypackage;
2. **import** javax.persistence.\*;
4. @Entity
5. @Table(name = "employee102")
6. @Inheritance(strategy = InheritanceType.TABLE\_PER\_CLASS)
8. **public** **class** Employee {
9. @Id
10. @GeneratedValue(strategy=GenerationType.AUTO)
12. @Column(name = "id")
13. **private** **int** id;
15. @Column(name = "name")
16. **private** String name;
18. //setters and getters
19. }

*File: Regular\_Employee.java*

1. **package** com.javatpoint.mypackage;
2. **import** javax.persistence.\*;
4. @Entity
5. @Table(name="regularemployee102")
6. @AttributeOverrides({
7. @AttributeOverride(name="id", column=@Column(name="id")),
8. @AttributeOverride(name="name", column=@Column(name="name"))
9. })
10. **public** **class** Regular\_Employee **extends** Employee{
12. @Column(name="salary")
13. **private** **float** salary;
15. @Column(name="bonus")
16. **private** **int** bonus;
18. //setters and getters
19. }

*File: Contract\_Employee.java*

1. **package** com.javatpoint.mypackage;
2. **import** javax.persistence.\*;
3. @Entity
4. @Table(name="contractemployee102")
5. @AttributeOverrides({
6. @AttributeOverride(name="id", column=@Column(name="id")),
7. @AttributeOverride(name="name", column=@Column(name="name"))
8. })
9. **public** **class** Contract\_Employee **extends** Employee{
11. @Column(name="pay\_per\_hour")
12. **private** **float** pay\_per\_hour;
14. @Column(name="contract\_duration")
15. **private** String contract\_duration;
17. **public** **float** getPay\_per\_hour() {
18. **return** pay\_per\_hour;
19. }
20. **public** **void** setPay\_per\_hour(**float** payPerHour) {
21. pay\_per\_hour = payPerHour;
22. }
23. **public** String getContract\_duration() {
24. **return** contract\_duration;
25. }
26. **public** **void** setContract\_duration(String contractDuration) {
27. contract\_duration = contractDuration;
28. }
29. }

### **2) Add mapping of hbm file in configuration file**

*File: hibernate.cfg.xml*

1. <?xml version='1.0' encoding='UTF-8'?>
2. <!DOCTYPE hibernate-configuration PUBLIC
3. "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
4. "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
6. <!-- Generated by MyEclipse Hibernate Tools.                   -->
7. <hibernate-configuration>
8. <session-factory>
9. <property name="hbm2ddl.auto">update</property>
10. <property name="dialect">org.hibernate.dialect.Oracle9Dialect</property>
11. <property name="connection.url">jdbc:oracle:thin:@localhost:1521:xe</property>
12. <property name="connection.username">system</property>
13. <property name="connection.password">oracle</property>
14. <property name="connection.driver\_class">oracle.jdbc.driver.OracleDriver</property>
16. <mapping **class**="com.javatpoint.mypackage.Employee"/>
17. <mapping **class**="com.javatpoint.mypackage.Contract\_Employee"/>
18. <mapping **class**="com.javatpoint.mypackage.Regular\_Employee"/>
19. </session-factory>
20. </hibernate-configuration>

The hbm2ddl.auto property is defined for creating automatic table in the database.

### **3) Create the class that stores the persistent object**

In this class, we are simply storing the employee objects in the database.

*File: StoreData.java*

1. **package** com.javatpoint.mypackage;
3. **import** org.hibernate.\*;
4. **import** org.hibernate.cfg.\*;
6. **public** **class** StoreData {
7. **public** **static** **void** main(String[] args) {
8. AnnotationConfiguration cfg=**new** AnnotationConfiguration();
9. Session session=cfg.configure("hibernate.cfg.xml").buildSessionFactory().openSession();
11. Transaction t=session.beginTransaction();
13. Employee e1=**new** Employee();
14. e1.setName("sonoo");
16. Regular\_Employee e2=**new** Regular\_Employee();
17. e2.setName("Vivek Kumar");
18. e2.setSalary(50000);
19. e2.setBonus(5);
21. Contract\_Employee e3=**new** Contract\_Employee();
22. e3.setName("Arjun Kumar");
23. e3.setPay\_per\_hour(1000);
24. e3.setContract\_duration("15 hours");
26. session.persist(e1);
27. session.persist(e2);
28. session.persist(e3);
30. t.commit();
31. session.close();
32. System.out.println("success");
33. }
34. }

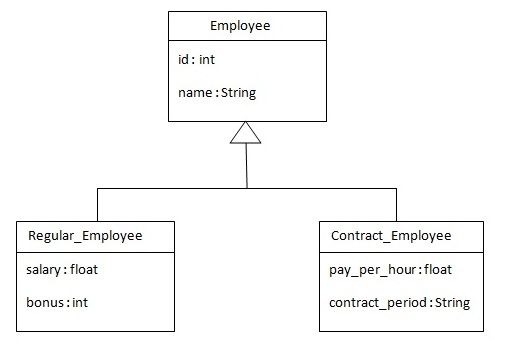
# Table Per Subclass using Annotation

1. [Table Per Subclass class](https://www.javatpoint.com/hibernate-table-per-subclass-using-annotation-tutorial-example)
2. [Example of Table Per Subclass](https://www.javatpoint.com/hibernate-table-per-subclass-using-annotation-tutorial-example#ex)

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

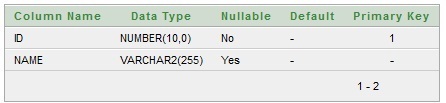
We need to specify **@Inheritance(strategy=InheritanceType.JOINED)** in the parent class and **@PrimaryKeyJoinColumn**annotation in the subclasses.

Let's see the hierarchy of classes that we are going to map.

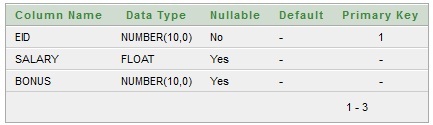
 

The table structure for each table will be as follows:

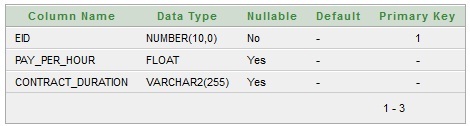
#### **Table structure for Employee class**

#### **Table structure for Regular\_Employee class**

#### **Table structure for Contract\_Employee class**



### **Example of Table per subclass class using Annotation**

In this example we are creating the three classes and provide mapping of these classes in the employee.hbm.xml file.

### **1) Create the Persistent classes**

You need to create the persistent classes representing the inheritance. Let's create the three classes for the above hierarchy:

*File: Employee.java*

1. **package** com.javatpoint.mypackage;
2. **import** javax.persistence.\*;
4. @Entity
5. @Table(name = "employee103")
6. @Inheritance(strategy=InheritanceType.JOINED)
8. **public** **class** Employee {
9. @Id
10. @GeneratedValue(strategy=GenerationType.AUTO)
12. @Column(name = "id")
13. **private** **int** id;
15. @Column(name = "name")
16. **private** String name;
18. //setters and getters
19. }

*File: Regular\_Employee.java*

1. **package** com.javatpoint.mypackage;
3. **import** javax.persistence.\*;
5. @Entity
6. @Table(name="regularemployee103")
7. @PrimaryKeyJoinColumn(name="ID")
8. **public** **class** Regular\_Employee **extends** Employee{
10. @Column(name="salary")
11. **private** **float** salary;
13. @Column(name="bonus")
14. **private** **int** bonus;
16. //setters and getters
17. }

*File: Contract\_Employee.java*

1. **package** com.javatpoint.mypackage;
3. **import** javax.persistence.\*;
5. @Entity
6. @Table(name="contractemployee103")
7. @PrimaryKeyJoinColumn(name="ID")
8. **public** **class** Contract\_Employee **extends** Employee{
10. @Column(name="pay\_per\_hour")
11. **private** **float** pay\_per\_hour;
13. @Column(name="contract\_duration")
14. **private** String contract\_duration;
16. //setters and getters
17. }

### **2) create configuration file**

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

1. <mapping **class**="com.javatpoint.mypackage.Employee"/>
2. <mapping **class**="com.javatpoint.mypackage.Contract\_Employee"/>
3. <mapping **class**="com.javatpoint.mypackage.Regular\_Employee"/>

Now the configuration file will look like this:

*File: hibernate.cfg.xml*

1. <?xml version='1.0' encoding='UTF-8'?>
2. <!DOCTYPE hibernate-configuration PUBLIC
3. "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
4. "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
6. <!-- Generated by MyEclipse Hibernate Tools.                   -->
7. <hibernate-configuration>
9. <session-factory>
10. <property name="hbm2ddl.auto">update</property>
11. <property name="dialect">org.hibernate.dialect.Oracle9Dialect</property>
12. <property name="connection.url">jdbc:oracle:thin:@localhost:1521:xe</property>
13. <property name="connection.username">system</property>
14. <property name="connection.password">oracle</property>
15. <property name="connection.driver\_class">oracle.jdbc.driver.OracleDriver</property>
17. <mapping **class**="com.javatpoint.mypackage.Employee"/>
18. <mapping **class**="com.javatpoint.mypackage.Contract\_Employee"/>
19. <mapping **class**="com.javatpoint.mypackage.Regular\_Employee"/>
20. </session-factory>
22. </hibernate-configuration>

The hbm2ddl.auto property is defined for creating automatic table in the database.

### **3) Create the class that stores the persistent object**

In this class, we are simply storing the employee objects in the database.

*File: StoreData.java*

1. **package** com.javatpoint.mypackage;
2. **import** org.hibernate.\*;
3. **import** org.hibernate.cfg.\*;
5. **public** **class** StoreData {
6. **public** **static** **void** main(String[] args) {
7. AnnotationConfiguration cfg=**new** AnnotationConfiguration();
8. Session session=cfg.configure("hibernate.cfg.xml").buildSessionFactory().openSession();
10. Transaction t=session.beginTransaction();
12. Employee e1=**new** Employee();
13. e1.setName("sonoo");
15. Regular\_Employee e2=**new** Regular\_Employee();
16. e2.setName("Vivek Kumar");
17. e2.setSalary(50000);
18. e2.setBonus(5);
20. Contract\_Employee e3=**new** Contract\_Employee();
21. e3.setName("Arjun Kumar");
22. e3.setPay\_per\_hour(1000);
23. e3.setContract\_duration("15 hours");
25. session.persist(e1);
26. session.persist(e2);
27. session.persist(e3);
29. t.commit();
30. session.close();
31. System.out.println("success");
32. }
33. }

Q= What is GenerationType in hibernate?

Ans=Generation type is used to generate the unique identifier for the objects of persistent class. There are many generator classes defined in the Hibernate Framework.

All the generator classes implements the **org.hibernate.id.IdentifierGenerator**[**interface**](https://www.javatpoint.com/interface-in-java).

1. assigned
2. increment
3. sequence
4. hilo
5. native
6. identity
7. seqhilo
8. uuid
9. guid
10. select
11. foreign
12. sequence-identity

example:

@Id

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

@Basic(optional = **false**)

@Column(name = "id")

**private** Long id;

Q=what is cascade type in hibernate?

Ans= The “Cascade” keyword is often appear on the collection mapping to manage the state of the collection automatically.

The cascade types supported by the Java Persistence Architecture are as below:

1. **CascadeType.PERSIST** : means that save() or persist() operations cascade to related entities.
2. **CascadeType.MERGE** : means that related entities are merged when the owning entity is merged.
3. **CascadeType.REFRESH** : does the same thing for the refresh() operation.
4. **CascadeType.REMOVE** : removes all related entities association with this setting when the owning entity is deleted.
5. **CascadeType.DETACH** : detaches all related entities if a “manual detach” occurs.
6. **CascadeType.ALL** : is shorthand for all of the above cascade operations.

Q= how to delete object from second level cache?

Ans=using evict() of sessionFActory

|  |  |
| --- | --- |
| void | [**evict**](https://docs.jboss.org/hibernate/orm/3.5/javadocs/org/hibernate/SessionFactory.html#evict(java.lang.Class))([Class](http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Class.html?is-external=true) persistentClass)            **Deprecated.** *Use [Cache.evictEntityRegion(Class)](https://docs.jboss.org/hibernate/orm/3.5/javadocs/org/hibernate/Cache.html" \l "evictEntityRegion(java.lang.Class)) accessed through [getCache()](https://docs.jboss.org/hibernate/orm/3.5/javadocs/org/hibernate/SessionFactory.html" \l "getCache()) instead.* |
| void | [**evict**](https://docs.jboss.org/hibernate/orm/3.5/javadocs/org/hibernate/SessionFactory.html#evict(java.lang.Class, java.io.Serializable))([Class](http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Class.html?is-external=true) persistentClass, [Serializable](http://java.sun.com/j2se/1.5.0/docs/api/java/io/Serializable.html?is-external=true) id)            **Deprecated.** *Use [Cache.containsEntity(Class, Serializable)](https://docs.jboss.org/hibernate/orm/3.5/javadocs/org/hibernate/Cache.html" \l "containsEntity(java.lang.Class, java.io.Serializable)) accessed through [getCache()](https://docs.jboss.org/hibernate/orm/3.5/javadocs/org/hibernate/SessionFactory.html" \l "getCache()) instead.* |
| void | [**evictCollection**](https://docs.jboss.org/hibernate/orm/3.5/javadocs/org/hibernate/SessionFactory.html#evictCollection(java.lang.String))([String](http://java.sun.com/j2se/1.5.0/docs/api/java/lang/String.html?is-external=true) roleName)            **Deprecated.** *Use*[*Cache.evictCollectionRegion(String)*](https://docs.jboss.org/hibernate/orm/3.5/javadocs/org/hibernate/Cache.html#evictCollectionRegion(java.lang.String))*accessed through [getCache()](https://docs.jboss.org/hibernate/orm/3.5/javadocs/org/hibernate/SessionFactory.html" \l "getCache()) instead.* |
| void | [**evictCollection**](https://docs.jboss.org/hibernate/orm/3.5/javadocs/org/hibernate/SessionFactory.html#evictCollection(java.lang.String, java.io.Serializable))([String](http://java.sun.com/j2se/1.5.0/docs/api/java/lang/String.html?is-external=true) roleName, [Serializable](http://java.sun.com/j2se/1.5.0/docs/api/java/io/Serializable.html?is-external=true) id)            **Deprecated.** *Use*[*Cache.evictCollection(String,Serializable)*](https://docs.jboss.org/hibernate/orm/3.5/javadocs/org/hibernate/Cache.html#evictCollection(java.lang.String, java.io.Serializable))*accessed through [getCache()](https://docs.jboss.org/hibernate/orm/3.5/javadocs/org/hibernate/SessionFactory.html" \l "getCache()) instead.* |
| void | [**evictEntity**](https://docs.jboss.org/hibernate/orm/3.5/javadocs/org/hibernate/SessionFactory.html#evictEntity(java.lang.String))([String](http://java.sun.com/j2se/1.5.0/docs/api/java/lang/String.html?is-external=true) entityName)            **Deprecated.** *Use*[*Cache.evictEntityRegion(String)*](https://docs.jboss.org/hibernate/orm/3.5/javadocs/org/hibernate/Cache.html#evictEntityRegion(java.lang.String))*accessed through [getCache()](https://docs.jboss.org/hibernate/orm/3.5/javadocs/org/hibernate/SessionFactory.html" \l "getCache()) instead.* |
| void | [**evictEntity**](https://docs.jboss.org/hibernate/orm/3.5/javadocs/org/hibernate/SessionFactory.html#evictEntity(java.lang.String, java.io.Serializable))([String](http://java.sun.com/j2se/1.5.0/docs/api/java/lang/String.html?is-external=true) entityName, [Serializable](http://java.sun.com/j2se/1.5.0/docs/api/java/io/Serializable.html?is-external=true) id)            **Deprecated.** *Use*[*Cache.evictEntity(String,Serializable)*](https://docs.jboss.org/hibernate/orm/3.5/javadocs/org/hibernate/Cache.html#evictEntity(java.lang.String, java.io.Serializable))*accessed through [getCache()](https://docs.jboss.org/hibernate/orm/3.5/javadocs/org/hibernate/SessionFactory.html" \l "getCache()) instead.* |
| void | [**evictQueries**](https://docs.jboss.org/hibernate/orm/3.5/javadocs/org/hibernate/SessionFactory.html#evictQueries())()            **Deprecated.** *Use [Cache.evictQueryRegions()](https://docs.jboss.org/hibernate/orm/3.5/javadocs/org/hibernate/Cache.html" \l "evictQueryRegions()) accessed through [getCache()](https://docs.jboss.org/hibernate/orm/3.5/javadocs/org/hibernate/SessionFactory.html" \l "getCache()) instead.* |
| void | [**evictQueries**](https://docs.jboss.org/hibernate/orm/3.5/javadocs/org/hibernate/SessionFactory.html#evictQueries(java.lang.String))([String](http://java.sun.com/j2se/1.5.0/docs/api/java/lang/String.html?is-external=true) cacheRegion)            **Deprecated.** *Use*[*Cache.evictQueryRegion(String)*](https://docs.jboss.org/hibernate/orm/3.5/javadocs/org/hibernate/Cache.html#evictQueryRegion(java.lang.String))*accessed through [getCache()](https://docs.jboss.org/hibernate/orm/3.5/javadocs/org/hibernate/SessionFactory.html" \l "getCache()) instead.* |

Q=how to remove object of first level cache?

Ans=evict(Object) and clear() to remove object.

Evict(Object):- remove sp[ecific object.

Clear:- Completely clear the session. Evict all loaded instances and cancel all pending

saves, updates and deletions. Do not close open iterators or instances of scrollableResults

Q=caching in hibernate?

Ans= Hibernate comes with different types of Cache:

1. **First Level Cache**: Hibernate first level cache is associated with the Session object. Hibernate first level cache is enabled by default and there is no way to disable it. However hibernate provides methods through which we can delete selected objects from the cache or clear the cache completely.

Any object cached in a session will not be visible to other sessions and when the session is closed, all the cached objects will also be lost.

1. **Second Level Cache**: Hibernate Second Level cache is disabled by default but we can enable it through configuration. Currently EHCache and Infinispan provides implementation for Hibernate Second level cache and we can use them. We will look into this in the next tutorial for hibernate caching.
2. **Query Cache**: Hibernate can also cache result set of a query. Hibernate Query Cache doesn’t cache the state of the actual entities in the cache; it caches only identifier values and results of value type. So it should always be used in conjunction with the second-level cache.

**First Level Cache:**

1. Hibernate First Level cache is enabled by default, there are no configurations needed for this.
2. Hibernate first level cache is session specific, that’s why when we are getting the same data in same session there is no query fired whereas in other session query is fired to load the data.
3. Hibernate first level cache can have old values, as you can see above that I have put my program to sleep for 10 seconds and in that time I updated the value (name from Pankaj to PankajK) in database but it didn’t get reflected in the same session. But in other session, we got the updated value.
4. We can use session evict() method to remove a single object from the hibernate first level cache.
5. We can use session clear() method to clear the cache i.e delete all the objects from the cache.
6. We can use session contains() method to check if an object is present in the hibernate cache or not, if the object is found in cache, it returns true or else it returns false.
7. Since hibernate cache all the objects into session first level cache, while running bulk queries or batch updates it’s necessary to clear the cache at certain intervals to avoid memory issues.

# Q= [How to apply unique constraints using annotation](https://stackoverflow.com/questions/7497095/how-to-apply-unique-constraints-using-annotation)

@Column(name = "userId",unique=true)

OR

@Entity

@Table(name="tbl\_sky",uniqueConstraints = {@UniqueConstraint(columnNames={"month", "day"})})

public class Sky implements Serializable {

...

}

# Q=Hibernate EhCache Configuration Tutorial?

Ans=[Home](https://howtodoinjava.com/)  >  [Hibernate](https://howtodoinjava.com/category/hibernate/)  >  Hibernate EhCache Configuration Tutorial

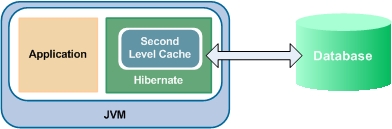
**Hibernate EhCache Configuration Tutorial**

July 4, 2013 by Lokesh Gupta

Caching is facility provided by ORM frameworks which help users to get fast running web application, while help framework itself to reduce number of queries made to database in a single transaction. [**Hibernate**](https://howtodoinjava.com/hibernate-tutorials/) also provide this caching functionality, in two layers.

* **Fist level cache**: This is enabled by default and works in session scope. Read more about [**hibernate first level cache**](https://howtodoinjava.com/hibernate/understanding-hibernate-first-level-cache-with-example/).
* **Second level cache**: This is apart from first level cache which is available to be used globally in session factory scope.

In this tutorial, I am giving an example using ehcache configuration as [**second level cache in hibernate**](https://howtodoinjava.com/hibernate/how-hibernate-second-level-cache-works/).

Hibernate with EhCache

**Sections in this post:**

How second level cache works

About EhCache

Configuring EhCache

Configuring entity objects

Query caching

Example application

Sourcecode download

**How second level cache works**

Lets write all the facts point by point:

1. Whenever hibernate session try to load an entity, the very first place it look for cached copy of entity in first level cache (associated with particular hibernate session).
2. If cached copy of entity is present in first level cache, it is returned as result of load method.
3. If there is no cached entity in first level cache, then second level cache is looked up for cached entity.
4. If second level cache has cached entity, it is returned as result of load method. But, before returning the entity, it is stored in first level cache also so that next invocation to load method for entity will return the entity from first level cache itself, and there will not be need to go to second level cache again.
5. If entity is not found in first level cache and second level cache also, then database query is executed and entity is stored in both cache levels, before returning as response of load() method.
6. Second level cache validate itself for modified entities, if modification has been done through hibernate session APIs.
7. If some user or process make changes directly in database, the there is no way that second level cache update itself until “timeToLiveSeconds” duration has passed for that cache region. In this case, it is good idea to invalidate whole cache and let hibernate build its cache once again. You can use below code snippet to invalidate whole hibernate second level cache.

The **maven dependency** is for Ehcache 2.0 and any upgrades is:

|  |
| --- |
| <dependency>      <groupId>net.sf.ehcache</groupId>      <artifactId>ehcache</artifactId>      <version>[2.0.0]</version>      <type>pom</type>  </dependency> |

**Configuring EhCache**

To configure ehcache, you need to do two steps:

1. configure Hibernate for second level caching
2. specify the second level cache provider

**Hibernate 4.x and above**

|  |
| --- |
| <property key="hibernate.cache.use\_second\_level\_cache">true</property>  <propertyname="hibernate.cache.region.factory\_class">org.hibernate.cache.ehcache.EhCacheRegionFactory</property> |

**Hibernate 3.3 and above**

|  |
| --- |
| <property key="hibernate.cache.use\_second\_level\_cache">true</property>  <propertyname="hibernate.cache.region.factory\_class">net.sf.ehcache.hibernate.EhCacheRegionFactory</property> |

**Hibernate 3.2 and below**

|  |
| --- |
| <property key="hibernate.cache.use\_second\_level\_cache">true</property>  <propertyname="hibernate.cache.region.provider\_class">net.sf.ehcache.hibernate.EhCacheProvider</property> |

**Configuring entity objects**

This may done in two ways.

1) If you are using **hbm.xml files** then use below configuration:

|  |
| --- |
| <class name="com.application.entity.DepartmentEntity" table="...">      <cache usage="read-write"/>  </class> |

2) Otherwise, if you are using annotations, use these **annotations**:

|  |
| --- |
| @Entity  @Cache(usage=CacheConcurrencyStrategy.READ\_ONLY,  region="department")  public class DepartmentEntity implements Serializable  {      //code  } |

For both options, **caching strategy** can be of following types:

* **none** : No caching will happen.
* **read-only** : If your application needs to read, but not modify, instances of a persistent class, a read-only cache can be used.
* **read-write** : If the application needs to update data, a read-write cache might be appropriate.
* **nonstrict-read-write** : If the application only occasionally needs to update data (i.e. if it is extremely unlikely that two transactions would try to update the same item simultaneously), and strict transaction isolation is not required, a nonstrict-read-write cache might be appropriate.
* **transactional** : The transactional cache strategy provides support for fully transactional cache providers such as JBoss TreeCache. Such a cache can only be used in a JTA environment and you must specify hibernate.transaction.manager\_lookup\_class.
* **Query caching**
* You can also enable query caching. To do so configure it in your hbm.xml:

|  |
| --- |
| <property key="hibernate.cache.use\_query\_cache">true</property> |

* and where queries are defined in your code, add the method call **setCacheable(true)** to the queries that should be cached:

|  |
| --- |
| sessionFactory.getCurrentSession().createQuery("...").setCacheable(true).list(); |

* By default, Ehcache will create separate cache regions for each entity that you configure for caching. You can change the defaults for these regions by adding the configuration to your ehcache.xml. To provide this configuration file, use this property in hibernate configuration:

|  |
| --- |
| <property name="net.sf.ehcache.configurationResourceName">/ehcache.xml</property> |

* And use below configuration to override the default configuration:

|  |
| --- |
| <cache      name="com.somecompany.someproject.domain.Country"      maxElementsInMemory="10000"      eternal="false"      timeToIdleSeconds="300"      timeToLiveSeconds="600"      overflowToDisk="true"  /> |

* Please note that in ehcache.xml,**if eternal=”true” then we should not write timeToIdealSeconds, timeToLiveSeconds**, hibernate will take care about those values  
  So if you want to give values manually better use eternal=”false” always, so that we can assign values into timeToIdealSeconds, timeToLiveSeconds manually.
* **timeToIdealSeconds=”seconds”** means, if the object in the global cache is ideal, means not using by any other class or object then it will be waited for some time we specified and deleted from the global cache if time is exceeds more than timeToIdealSeconds value.
* **timeToLiveSeconds=”seconds”** means, the other Session or class using this object or not, i mean may be it is using by other sessions or may not, what ever the situation might be, once it competed the time specified timeToLiveSeconds, then it will be removed from the global cache by hibernate.

Hibernate : Session Lock

This tutorial contains how Session.lock() method works in Hibernate.

**Hibernate : Session Lock :-**

The lock() method obtains the specified lock level upon the given object. It allows your application   
to reattach the object to the session. It doesn't check or update the database as it assumes that the  
database is in sync with the detached object.

**Syntax:**   lock(Object entity,LockMode lockMode)  
**Specified by:**  lock in interface HibernateOperations  
**Parameters:**  
entity - the persistent instance to lock  
lockMode - the lock mode to obtain  
**Throws:**  
DataAccessException - in case of Hibernate errors

There are five kind of locks that can be obtained -

**LockMode.WRITE :** It is obtained when Hibernate updates an object or saves new object.

**LockMode.UPGRADE :**Hibernate obtains this lock when using the SELECT <String> FOR UPDATE SQL command.

**LockMode.UPGRADE\_NOWAIT :** Hibernate obtains this lock when using the SELECT <String> FOR UPDATE SQL command under the Oracle database server.

**LockMode.READ :** Hibernate obtains this lock either at the user's request or when neede for reading an object.

**LockMode.NONE :** Hibernate obtains this lock when a transaction finishes or during the start of a call to update() or saveOrUpdate().

**Example :**In this example we are updating record of student and using lock method to update record.

**Here is main class code -**

package net.roseindia.main;

import net.roseindia.table.Student;

import net.roseindia.util.HibernateUtil;

import org.hibernate.HibernateException;

import org.hibernate.LockMode;

import org.hibernate.Session;

import org.hibernate.Transaction;

public class HibernateUpdate {

public static void main(String args[]) {

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

Transaction transaction = session.beginTransaction();

int roll = 5;

Student student = (Student) session.get(Student.class, roll);

try {

session.lock(student, LockMode.UPGRADE); // Using lock()

student.setCourse("Hibernate");

//session.update(student);

//session.saveOrUpdate(student);

session.merge(student);

transaction.commit();

System.out.println("Update Successfully");

session.close();

} catch (HibernateException e) {

e.printStackTrace();

}

}

}

**Output :**

Hibernate: select student0\_.roll\_no as roll1\_0\_0\_, student0\_.course as course0\_0\_, student0\_.name as name0\_0\_ from student student0\_ where student0\_.roll\_no=?

Hibernate: select roll\_no from student where roll\_no =? for update

Hibernate: update student set course=?, name=? where roll\_no=?

Update Successfully

**Q =** [how to make two column as a primary key in hibernate annotation class](https://stackoverflow.com/questions/21284175/how-to-make-two-column-as-a-primary-key-in-hibernate-annotation-class)**?**

**Ans=**You should create a new @Embeddable class containing the PK fields:

@Embeddable

public class user\_groupId implements Serializable {

@Column(name="userId")

private String userId;

@Column(name="groupId")

private String group;

}

And use it in the @Entity as an @EmbeddedId:

@Entity

public class user\_group {

@EmbeddedId

user\_groupId id;

...

}

# Q=Hibernate One-to-One Mapping Using Annotations

In hibernate there are 3 ways to create one-to-one relationships between two entities. Either way you have to use [**@OneToOne**](http://docs.oracle.com/javaee/5/api/javax/persistence/OneToOne.html) annotation. ***First technique*** is most widely used and uses a foreign key column in one to table. ***Second technique*** uses a rather known solution of having a third table to store mapping between first two tables. ***Third technique*** is something new which uses a common primary key value in both the tables.

**Using foreign key association**

@OneToOne

@JoinColumn(name="ACCOUNT\_ID")

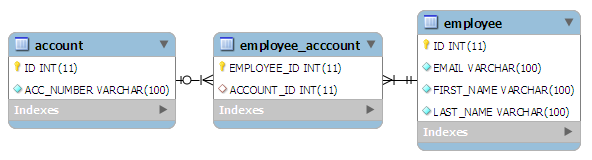
private AccountEntity account;

@OneToOne(mappedBy="account")

private EmployeeEntity employee; //bidirectional

**Using a common join table**

This approach is not new to all of us. Lets start with targeted DB structure in this technique.

[](https://howtodoinjava.files.wordpress.com/2012/11/join-table-one-to-one-mapping.png)

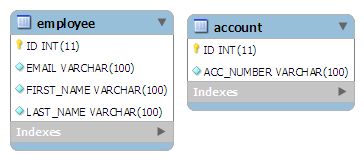
In this technique, main annotation to be used is [**@JoinTable**](http://docs.oracle.com/javaee/5/api/javax/persistence/JoinTable.html). **This annotation is used to define the new table name (mandatory) and foreign keys from both of the tables**. Lets see how it is used:

|  |
| --- |
| @OneToOne(cascade = CascadeType.ALL)  @JoinTable(name="EMPLOYEE\_ACCCOUNT", joinColumns = @JoinColumn(name="EMPLOYEE\_ID"),  inverseJoinColumns = @JoinColumn(name="ACCOUNT\_ID"))  private AccountEntity account; |

**Using shared primary key**

In this technique, hibernate will ensure that **it will use a common primary key value in both the tables**. This way primary key of EmployeeEntity can safely be assumed the primary key of AccountEntity also.

Table structure will be like this:

[](https://howtodoinjava.files.wordpress.com/2012/11/shared-primary-key-one-to-one.png)

In this approach, [**@PrimaryKeyJoinColumn**](http://docs.oracle.com/javaee/5/api/javax/persistence/PrimaryKeyJoinColumn.html) is the main annotation to be used.Let see how to use it.

|  |
| --- |
| @OneToOne(cascade = CascadeType.ALL)  @PrimaryKeyJoinColumn  private AccountEntity account; |

**Q= How to Enable Lazy Loading in Hibernate?**

**Ans=** @OneToMany( mappedBy = "category", fetch = FetchType.LAZY )

private Set<ProductEntity> products;

Bydefault all mapping is lazy loading.

As we know that hibernate can only access the database via a session, So If an entity is detached from the session and when we try to access an association (via a proxy or collection wrapper) that has not yet been loaded, **Hibernate throws a LazyInitializationException**.

# Q=Hibernate @NaturalId Example Tutorial?

Ans= [**Hibernate**](https://howtodoinjava.com/hibernate-tutorials/) 4 has bring lots of improvements and [**@NaturalId**](http://docs.jboss.org/hibernate/orm/4.0/javadocs/org/hibernate/annotations/NaturalId.html) is one of such nice improvements. As you know @Id annotation is used as meta data for specifying the primary key of an entity. But sometimes, entity is usually used in DAO layer code with id which not not primary key but its logical or natural id. In such cases, @NaturalId annotation will prove good replacement of named queries in hibernate.

For example, in any application there can be an employee entity. In this case, primary key will definitely be “employee id” but in cases such as login by email, user will provide email and password. In this case, in stead of writing named query, you can directly use @NaturalId annotation on “email” field.

@Entity

@Table(name = "Employee", uniqueConstraints = {

        @UniqueConstraint(columnNames = "ID"),

        @UniqueConstraint(columnNames = "EMAIL") })

public class EmployeeEntity implements Serializable {

    private static final long serialVersionUID = -1798070786993154676L;

    @Id

    @GeneratedValue(strategy = GenerationType.IDENTITY)

    @Column(name = "ID", unique = true, nullable = false)

    private Integer employeeId;

    //Use the natural id annotation here

    @NaturalId (mutable = false)

    @Column(name = "EMAIL", unique = true, nullable = false, length = 100)

    private String email;

    @Column(name = "FIRST\_NAME", unique = false, nullable = false, length = 100)

    private String firstName;

    @Column(name = "LAST\_NAME", unique = false, nullable = false, length = 100)

    private String lastName;

    //Setters and Getters

}

public class TestHibernate

{

    public static void main(String[] args)

    {

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

        session.beginTransaction();

        //Add new Employee object

        EmployeeEntity emp = new EmployeeEntity();

        emp.setEmail("demo-user@mail.com");

        emp.setFirstName("demo");

        emp.setLastName("user");

        //Save entity

        session.save(emp);

        EmployeeEntity empGet = (EmployeeEntity) session.bySimpleNaturalId( EmployeeEntity.class).load( "demo-user@mail.com" );

        System.out.println(empGet.getFirstName());

        System.out.println(empGet.getLastName());

        session.getTransaction().commit();

        HibernateUtil.shutdown();

    }

}