**Hibernate**: Hibernate is one of the most widely used ORM tool for Java applications. It’s used a lot in enterprise applications for database operations. Hibernate is java based ORM tool that provides framework for mapping application domain objects to the relational database tables and vice versa.

Hibernate provides reference implementation of Java Persistence API, that makes it a great choice as ORM tool with benefits of loose coupling.

Hibernate framework provide option to map plain old java objects(POJO) to traditional database tables with the use of JPA annotations as well as XML based configuration.

**ORM:** ORM (Object Relational Mapping) is the fundamental concept of Hibernate framework which maps database tables with Java Objects and then provides various API’s to perform different types of operations on the data tables.

**What is Java Persistence API (JPA)?**

Java Persistence API (JPA) provides specification for managing the relational data in applications. Current JPA version 2.1 was started in July 2011 as JSR 338. JPA 2.1 was approved as final on 22 May 2013.

JPA specifications is defined with annotations in javax.persistence package. Using JPA annotation helps us in writing implementation independent code.

**What are the important benefits of using Hibernate Framework?**

1. Hibernate eliminates all the boiler-plate code that comes with JDBC and takes care of managing resources, so we can focus on business logic.
2. Hibernate framework provides support for XML as well as JPA annotations, that makes our code implementation independent.
3. Hibernate provides a powerful query language (HQL) that is similar to SQL.
4. Hibernate is an open source project from Red Hat Community and used worldwide.
5. Hibernate is easy to integrate with other Java EE frameworks, it’s so popular that Spring Framework provides built-in support for integrating hibernate with Spring applications.
6. Hibernate supports lazy initialization using proxy objects and perform actual database queries only when it’s required.
7. Hibernate cache helps us in getting better performance.

**What are the advantages of Hibernate over JDBC?**

1. Hibernate removes a lot of boiler-plate code that comes with JDBC API, the code looks more cleaner and readable.
2. Hibernate supports inheritance, associations and collections. These features are not present with JDBC API.
3. Hibernate implicitly provides transaction management, in fact most of the queries can’t be executed outside transaction. In JDBC API, we need to write code for transaction management using commit and rollback.
4. JDBC API throws SQLException that is a checked exception, so we need to write a lot of try-catch block code. Most of the times it’s redundant in every JDBC call and used for transaction management. Hibernate wraps JDBC exceptions and throw JDBCException or HibernateException un-checked exception, so we don’t need to write code to handle it. Hibernate built-in transaction management removes the usage of try-catch blocks.
5. Hibernate Query Language (HQL) is more object oriented and close to java programming language. For JDBC, we need to write native sql queries.
6. Hibernate supports caching that is better for performance, JDBC queries are not cached hence performance is low.
7. Hibernate provide option through which we can create database tables too, for JDBC tables must exist in the database.
8. Hibernate configuration helps us in using JDBC like connection as well as JNDI DataSource for connection pool. This is very important feature in enterprise application and completely missing in JDBC API.
9. Hibernate supports JPA annotations, so code is independent of implementation and easily replaceable with other ORM tools. JDBC code is very tightly coupled with the application.

**Pojo:** POJOs( Plain Old Java Objects) are java beans with proper getter and setter methods for each and every properties.

Use of POJOs instead of simple java classes results in an efficient and well constructed code.

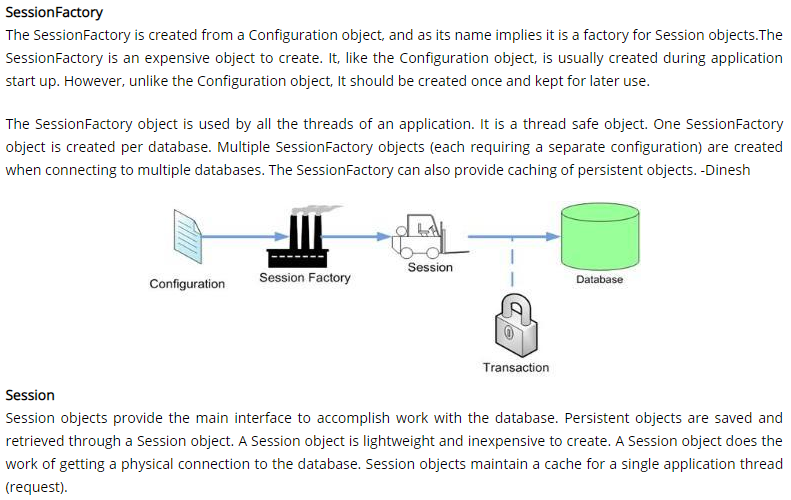
**What’s the usage of Configuration Interface in hibernate?**

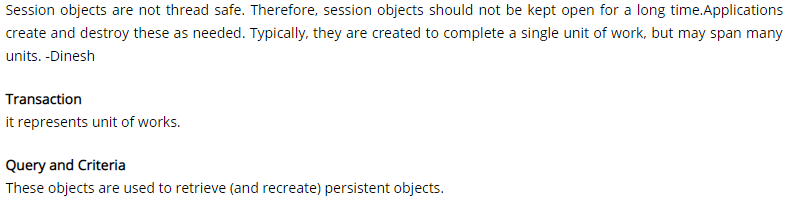
Configuration interface of hibernate framework is used to configure hibernate. It’s also used to bootstrap hibernate. Mapping documents of hibernate are located using this interface.

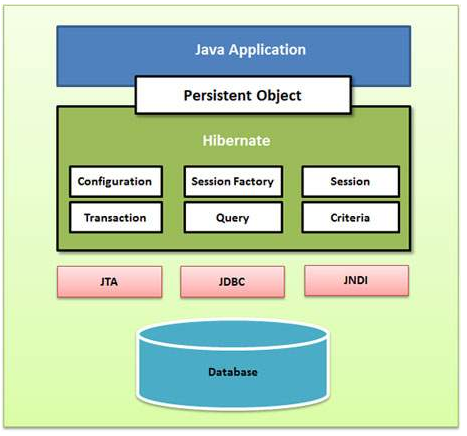
**Name some important interfaces of Hibernate framework?**

1. **SessionFactory (org.hibernate.SessionFactory):** SessionFactory is an immutable thread-safe cache of compiled mappings for a single database. We need to initialize SessionFactory once and then we can cache and reuse it. SessionFactory instance is used to get the Session objects for database operations.
2. **Session (org.hibernate.Session):** Session is a single-threaded, short-lived object representing a conversation between the application and the persistent store. It wraps JDBC java.sql.Connection and works as a factory for org.hibernate.Transaction. We should open session only when it’s required and close it as soon as we are done using it. Session object is the interface between java application code and hibernate framework and provide methods for CRUD operations.
3. **Transaction (org.hibernate.Transaction):** Transaction is a single-threaded, short-lived object used by the application to specify atomic units of work. It abstracts the application from the underlying JDBC or JTA transaction. A org.hibernate.Session might span multiple org.hibernate.Transaction in some cases.

**Hibernate Architecture:**







**What is SessionFactory in Hibernate?**

SessionFactory is an interface. SessionFactory can be created by providing Configuration object, which will contain all DB related property details pulled from either hibernate.cfg.xml file or hibernate.properties file. SessionFactory is a factory for Session objects.

SessionFactory is the factory class used to get the Session objects. SessionFactory is responsible to read the hibernate configuration parameters and connect to the database and provide Session objects. Usually an application has a single SessionFactory instance and threads servicing client requests obtain Session instances from this factory.

We can create one SessionFactory implementation per database in any application. If your application is referring to multiple databases, then you need to create one SessionFactory per database.

Internal state of SessionFactory is immutable, so it’s thread safe. Multiple threads can access it simultaneously to get Session instances.

**Hibernate SessionFactory provides three methods through which we can get Session object – getCurrentSession(), openSession() and openStatelessSession()**.

**Hibernate SessionFactory getCurrentSession()** method returns the session bound to the context. But for this to work, we need to configure it in hibernate configuration file like below:

<property name="hibernate.current\_session\_context\_class">thread</property>

Since this session object belongs to the hibernate context, we don’t need to close it. Once the session factory is closed, this session object gets closed.

Hibernate Session objects are not thread safe, so we should not use it in multi-threaded environment. We can use it in single threaded environment because it’s relatively faster than opening a new session.

**Hibernate SessionFactory openSession()** method always opens a new session. We should close this session object once we are done with all the database operations.

We should open a new session for each request in multi-threaded environment. For web application frameworks, we can choose to open a new session for each request or for each session based on the requirement.

**Hibernate SessionFactory openStatelessSession()** method returns instance of StatelessSession. There is another overloaded method where we can pass java.sql.Connection object to get a stateless session object from hibernate.

StatelessSession in Hibernate does not implement first-level cache and it doesn’t interact with any second-level cache. Since it’s stateless, it doesn’t implement transactional write-behind or automatic dirty checking or do cascading operations to associated entities.

However, stateless session can be a good fit in certain situations. For example where we are loading bulk data into database and we don’t want hibernate session to hold huge data in first-level cache memory.

**What is hibernate configuration file?**

Hibernate configuration file contains database specific configurations and used to initialize SessionFactory. We provide database credentials or JNDI resource information in the hibernate configuration xml file. Some other important parts of hibernate configuration file is Dialect information, so that hibernate knows the database type and mapping file or class details.

**What is hibernate mapping file?**

Hibernate mapping file is used to define the entity bean fields and database table column mappings. We know that JPA annotations can be used for mapping but sometimes XML mapping file comes handy when we are using third party classes and we can’t use annotations.

**Name some important annotations used for Hibernate mapping?**

Hibernate supports JPA annotations and it has some other annotations in **org.hibernate.annotations** package.

1. **javax.persistence.Entity**: Used with model classes to specify that they are entity beans.
2. **javax.persistence.Table**: Used with entity beans to define the corresponding table name in database.
3. **javax.persistence.Access:** Used to define the access type, either field or property. Default value is field and if you want hibernate to use getter/setter methods then you need to set it to property.
4. **javax.persistence.Id:** Used to define the primary key in the entity bean.
5. **javax.persistence.EmbeddedId:** Used to define composite primary key in the entity bean.
6. **javax.persistence.Column:** Used to define the column name in database table.
7. **javax.persistence.GeneratedValue:** Used to define the strategy to be used for generation of primary key. Used in conjunction with javax.persistence.GenerationType enum.
8. **javax.persistence.OneToOne:** Used to define the one-to-one mapping between two entity beans. We have other similar annotations as OneToMany, ManyToOne and ManyToMany
9. **org.hibernate.annotations.Cascade:** Used to define the cascading between two entity beans, used with mappings. It works in conjunction with org.hibernate.annotations.CascadeType
10. **javax.persistence.PrimaryKeyJoinColumn:** Used to define the property for foreign key. Used with org.hibernate.annotations.GenericGenerator and org.hibernate.annotations.Parameter

Example:

import javax.persistence.Access;

import javax.persistence.AccessType;

import javax.persistence.Column;

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.GenerationType;

import javax.persistence.Id;

import javax.persistence.OneToOne;

import javax.persistence.Table;

import org.hibernate.annotations.Cascade;

@Entity

@Table(name = "EMPLOYEE")

@Access(value=AccessType.FIELD)

public class Employee {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

@Column(name = "emp\_id")

private long id;

@Column(name = "emp\_name")

private String name;

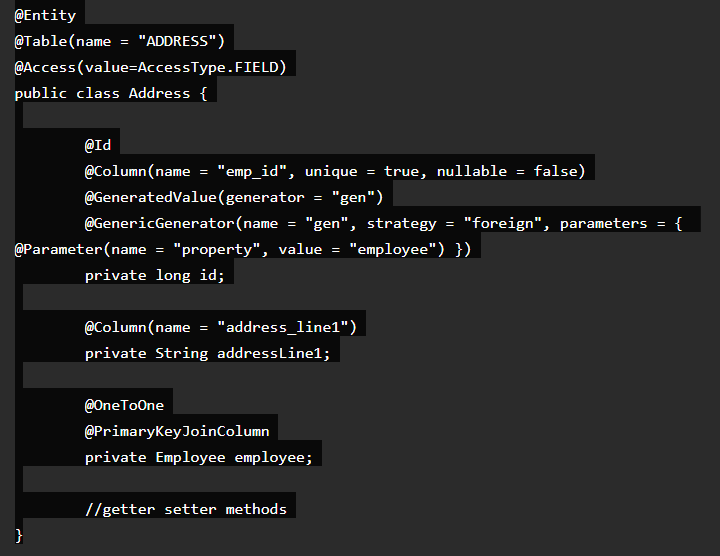
@OneToOne(mappedBy = "employee")

@Cascade(value = org.hibernate.annotations.CascadeType.ALL)

private Address address;

//getter setter method

}



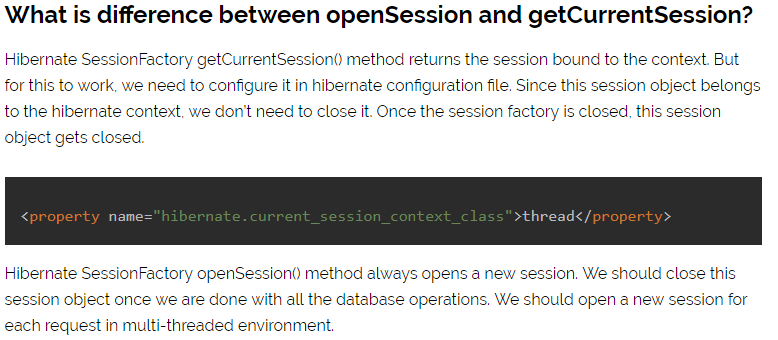
**What is Hibernate Session and how to get it?**

Hibernate Session is the interface between java application layer and hibernate. This is the core interface used to perform database operations. Lifecycle of a session is bound by the beginning and end of a transaction.

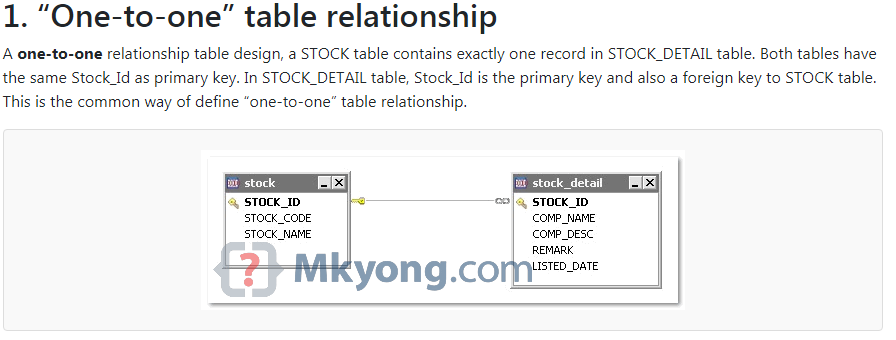
Session provide methods to perform create, read, update and delete operations for a persistent object. We can execute HQL queries, SQL native queries and create criteria using Session object.

Hibernate Session object is not thread safe, every thread should get it’s own session instance and close it after it’s work is finished.

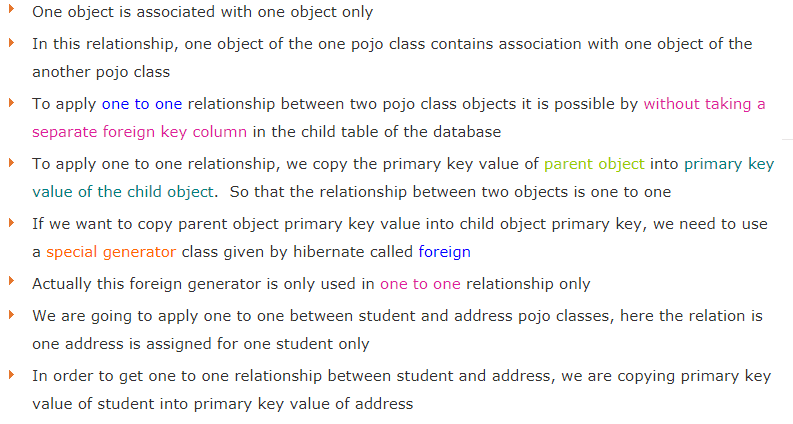
Unlike SessionFactory, the Session object will be created on demand. Session is a lightweight object. Session provides a physical connectivity between your application and database. The Session will be established each time your application wants do something with database.

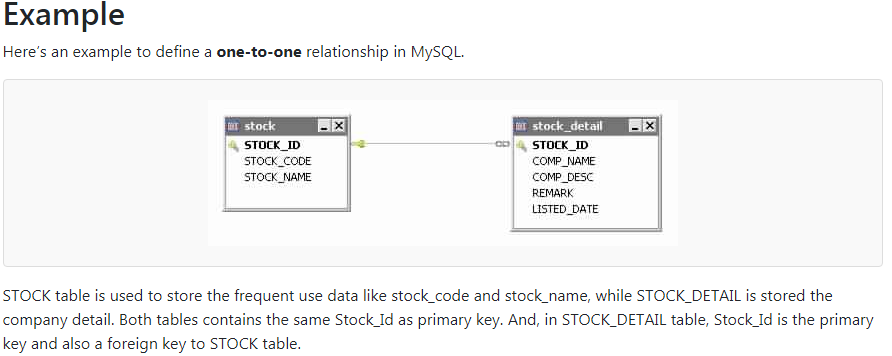


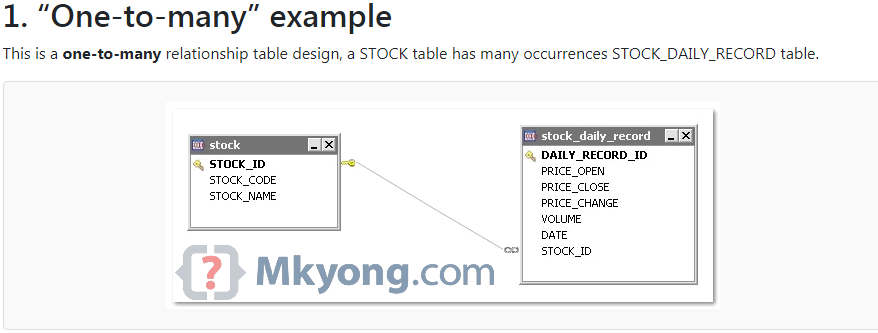
**Hibernate Relationship Mapping(Association Mapping):**





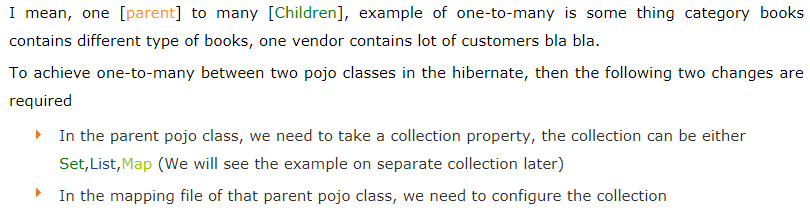


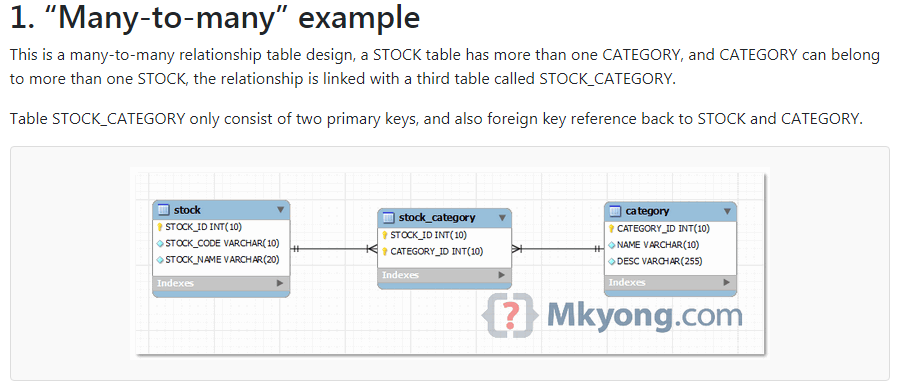




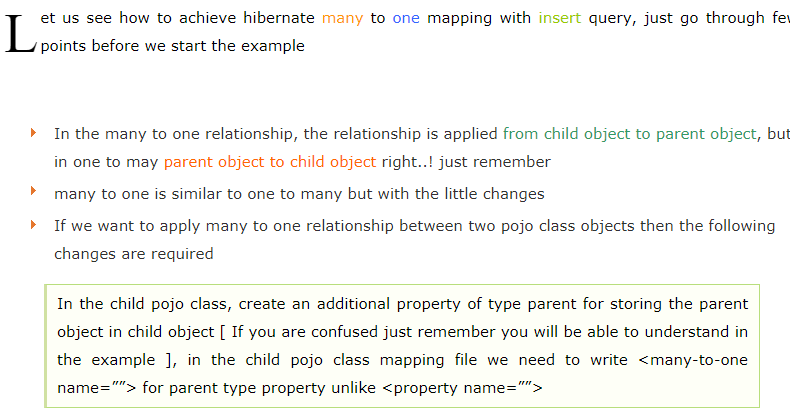


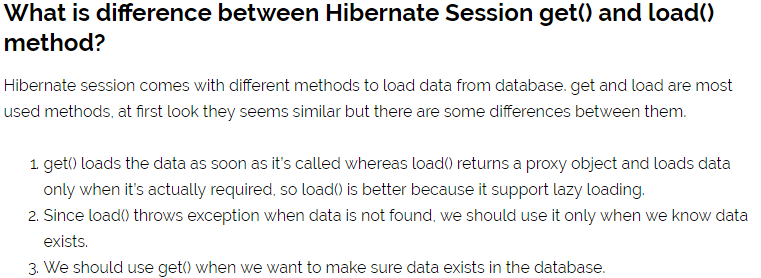




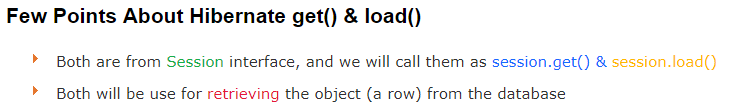


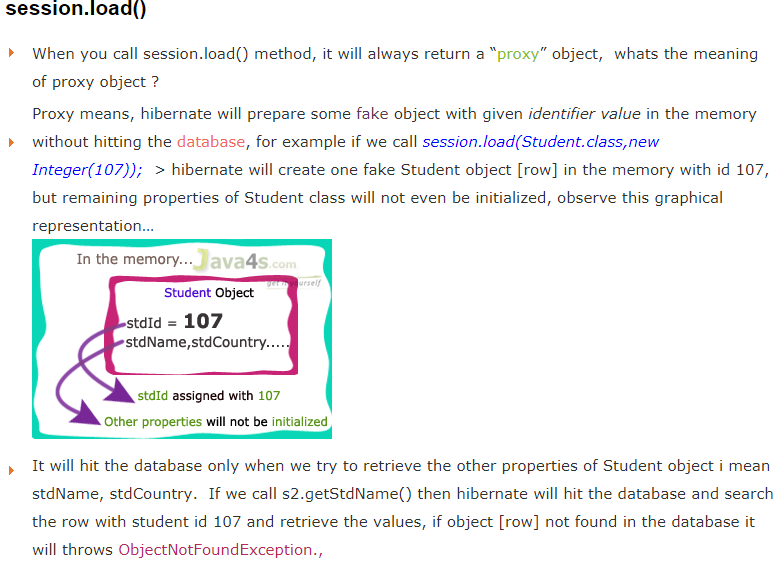
Many-to-one:

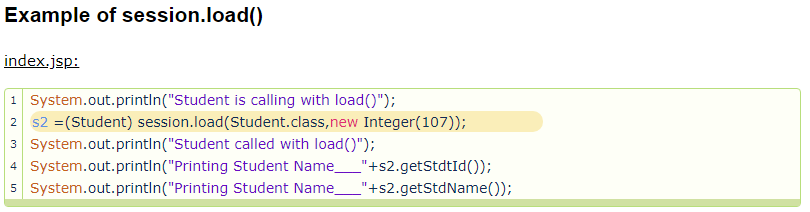


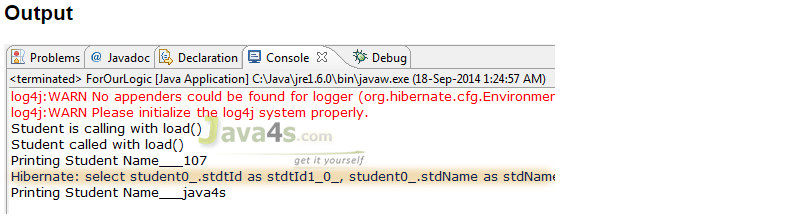


Example:

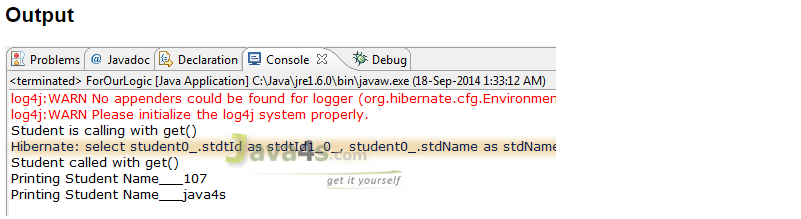


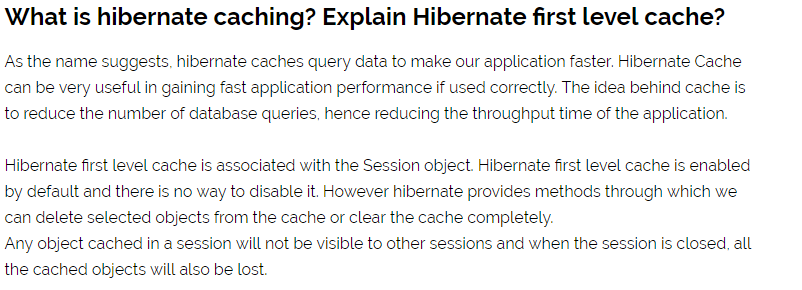


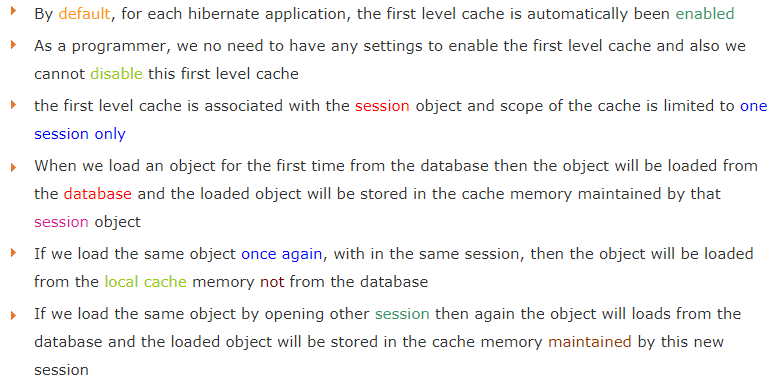


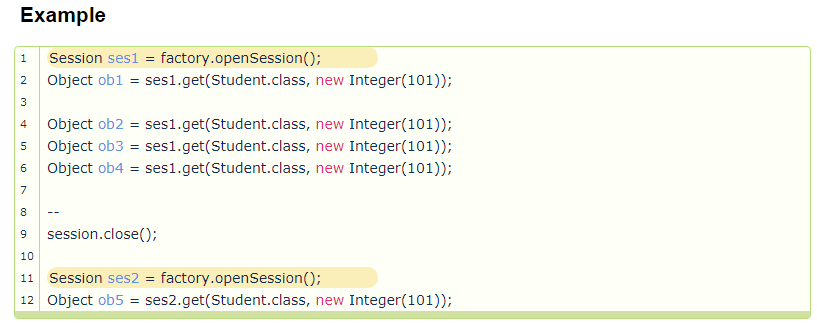


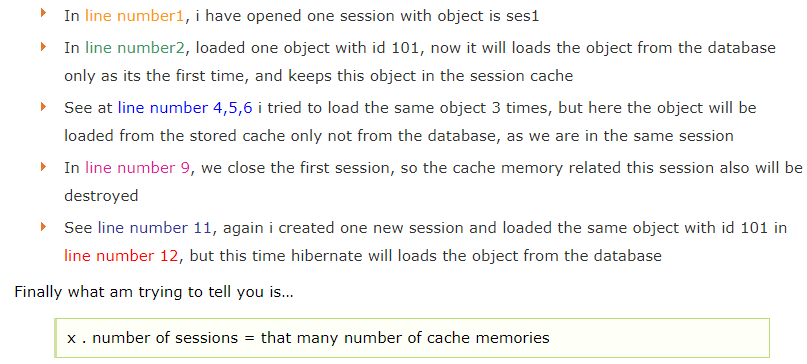


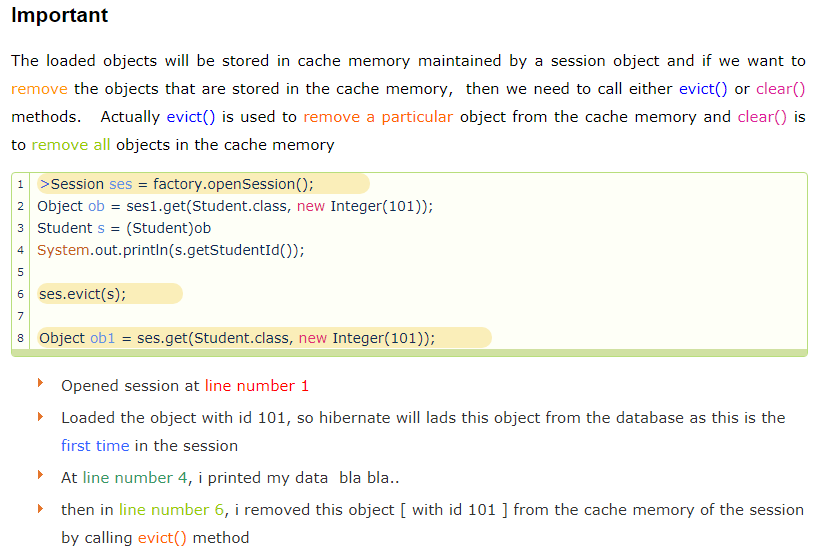


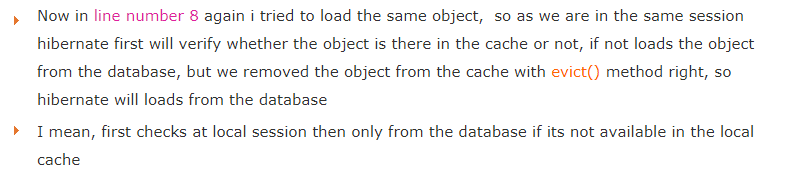






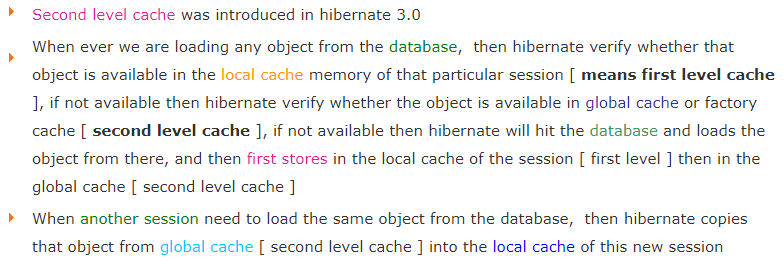


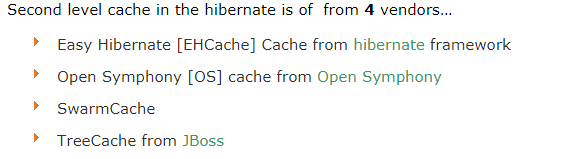


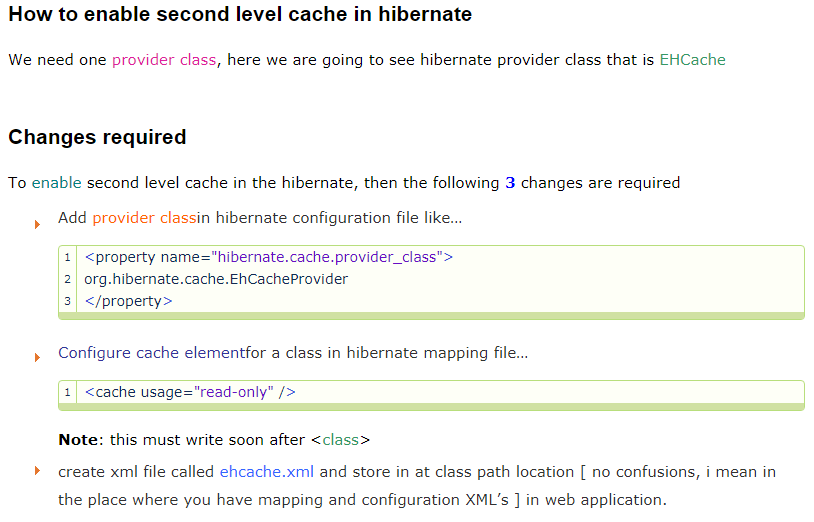


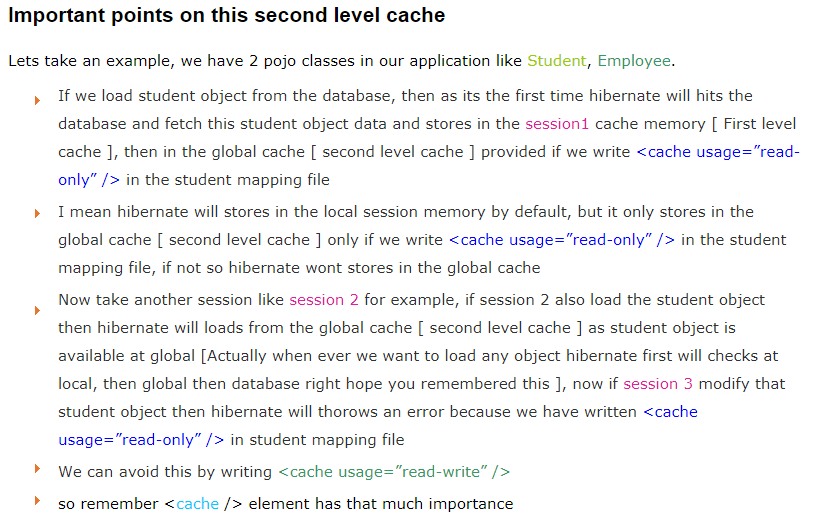
How To Enable Second Level Caching In Hibernate:



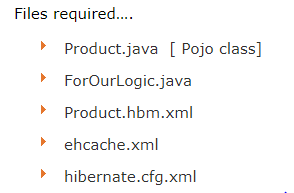








**Example Second Level Cache:**



Product.java->

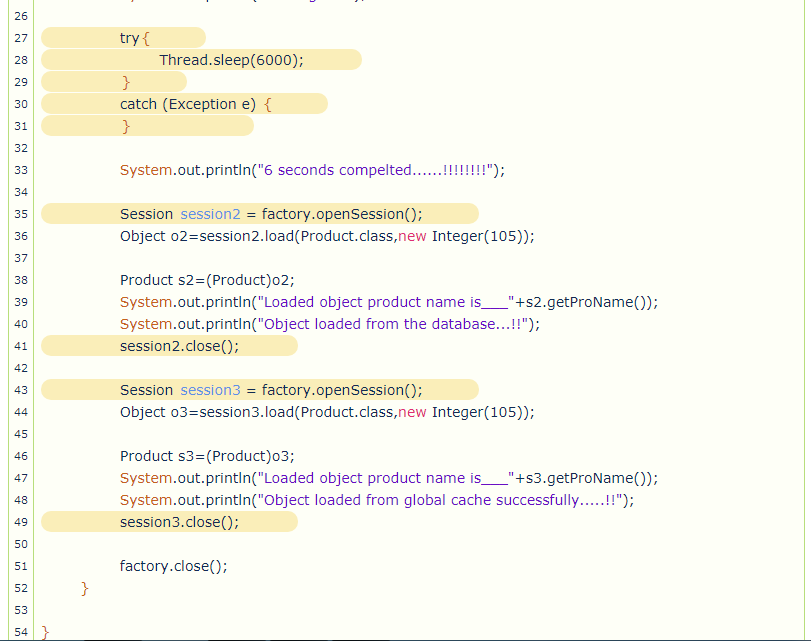
private int productId;

private String proName;

private double price

//Setter and Getter.

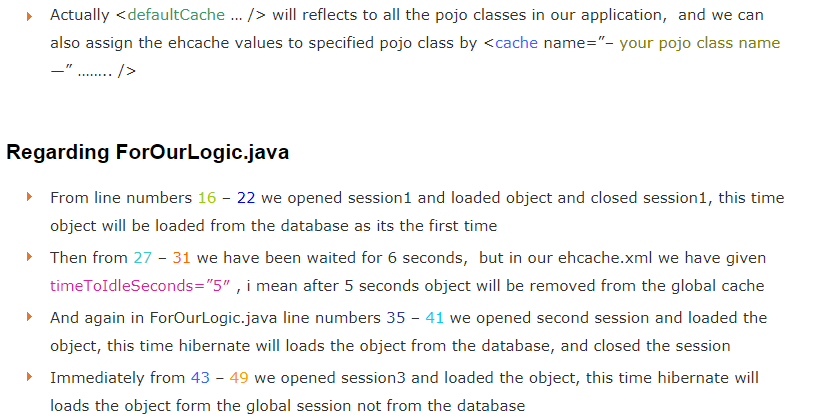


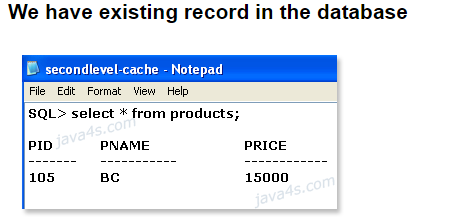


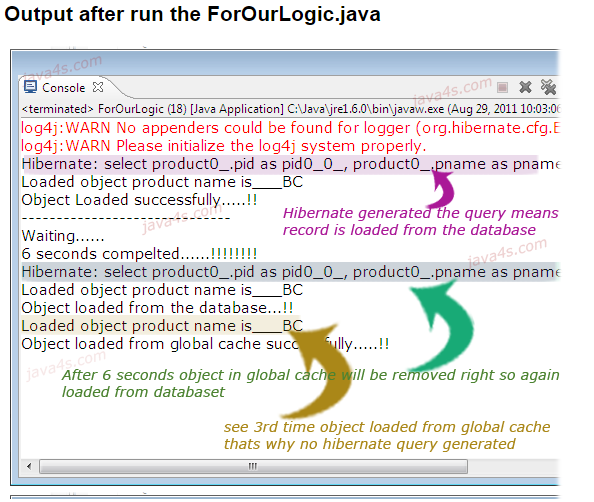


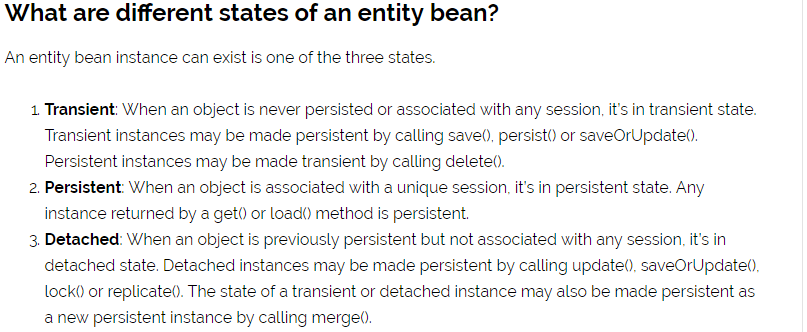


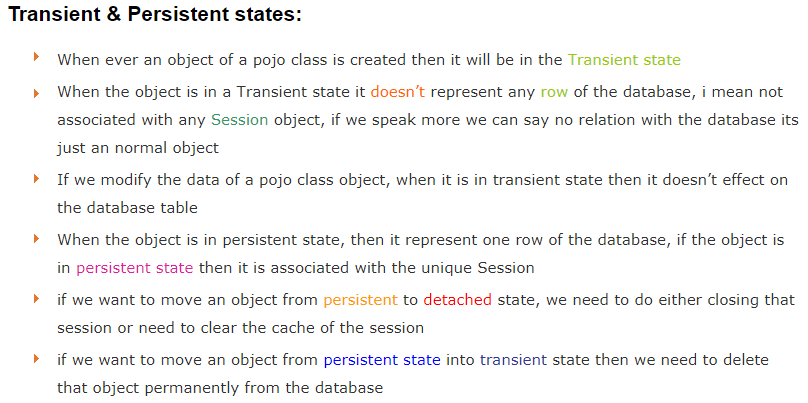




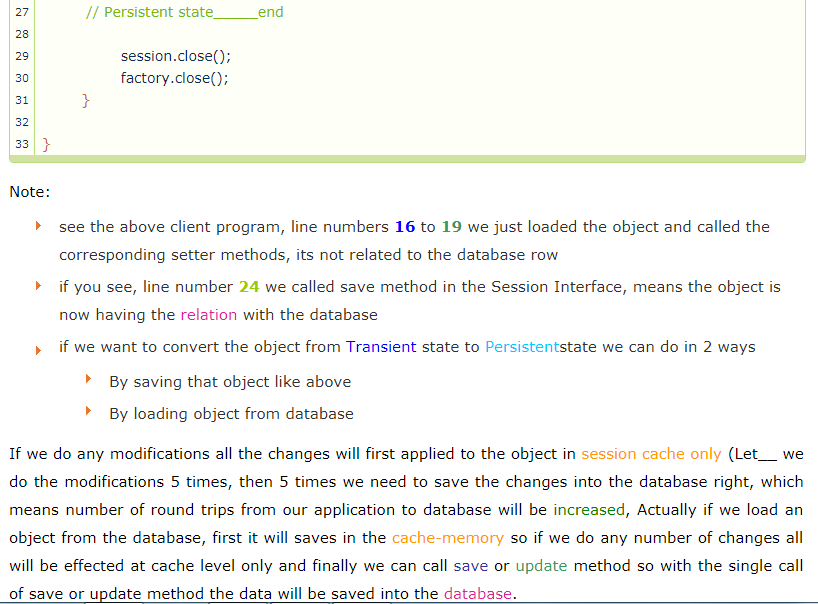


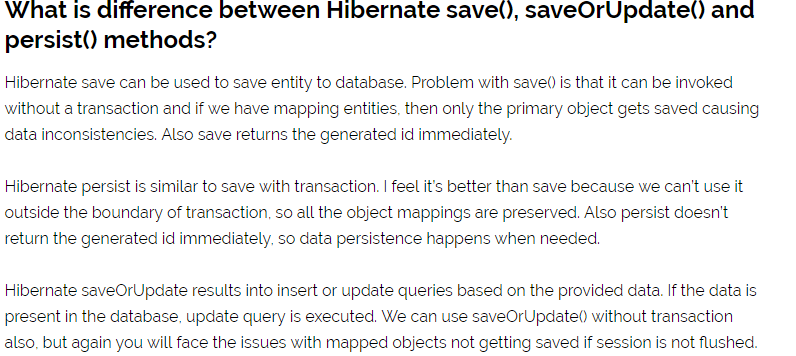


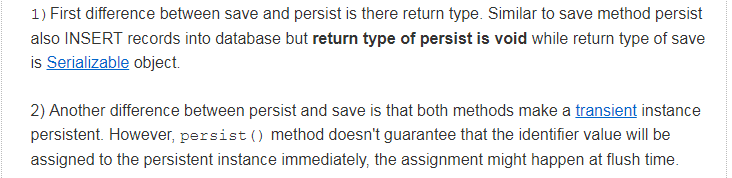


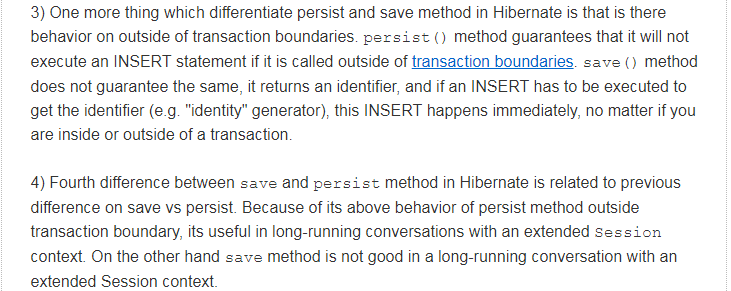


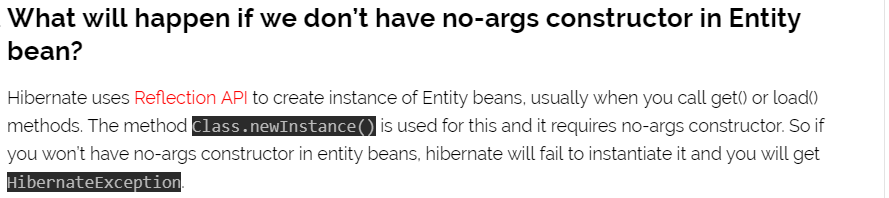


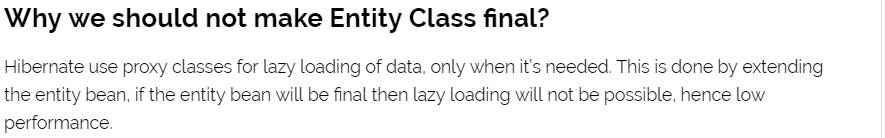


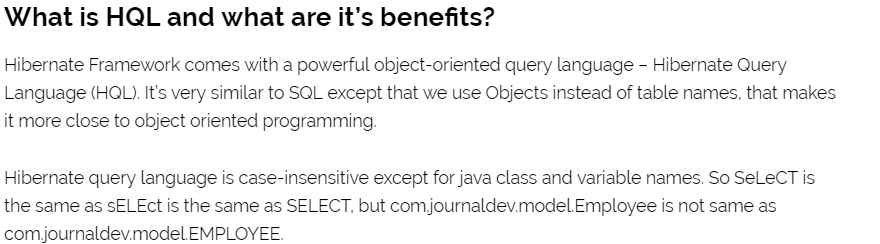


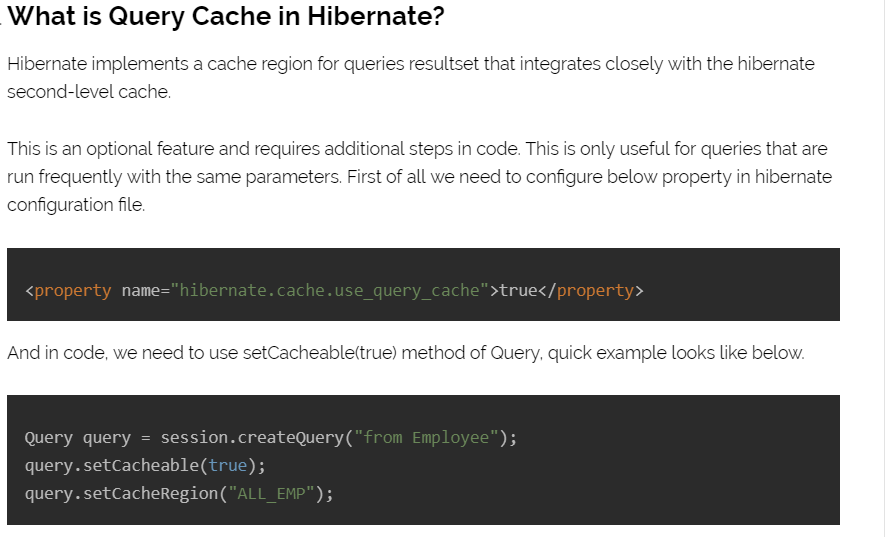


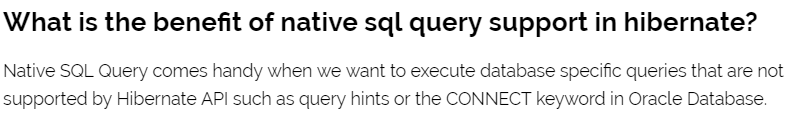


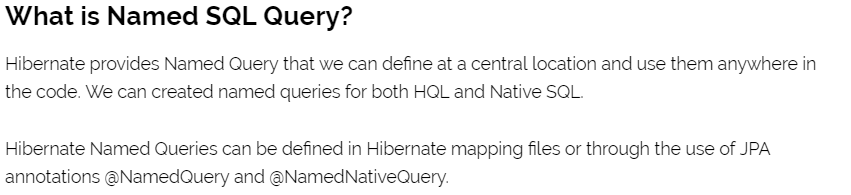


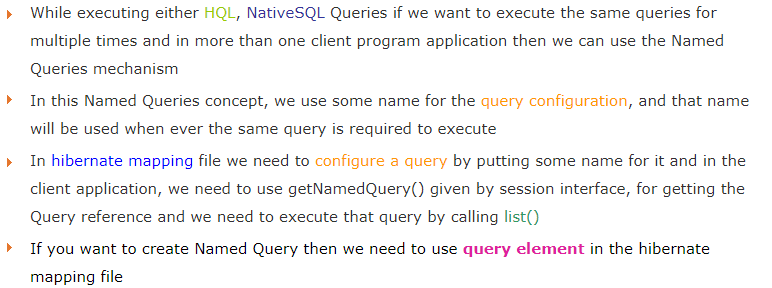


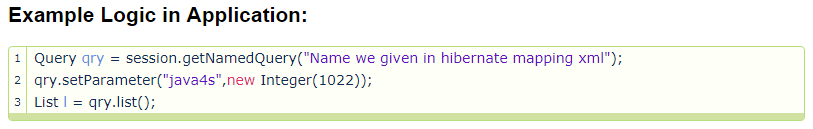


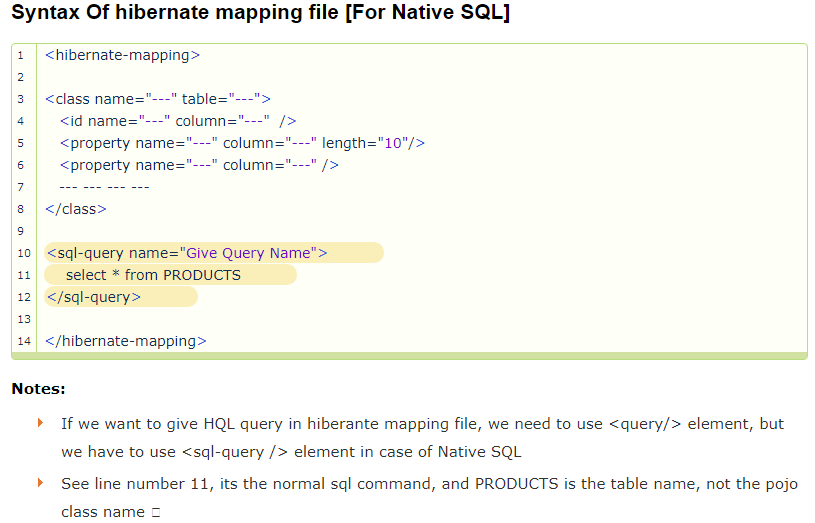


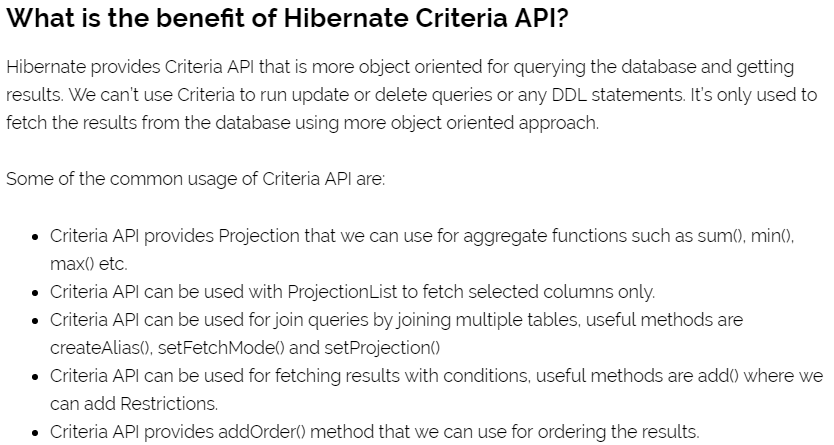




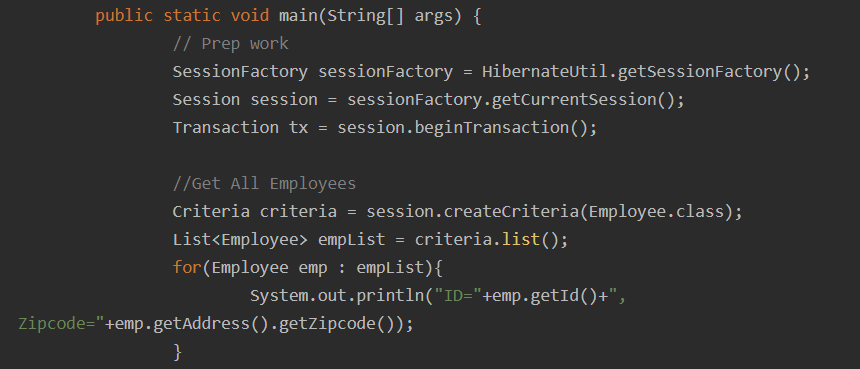


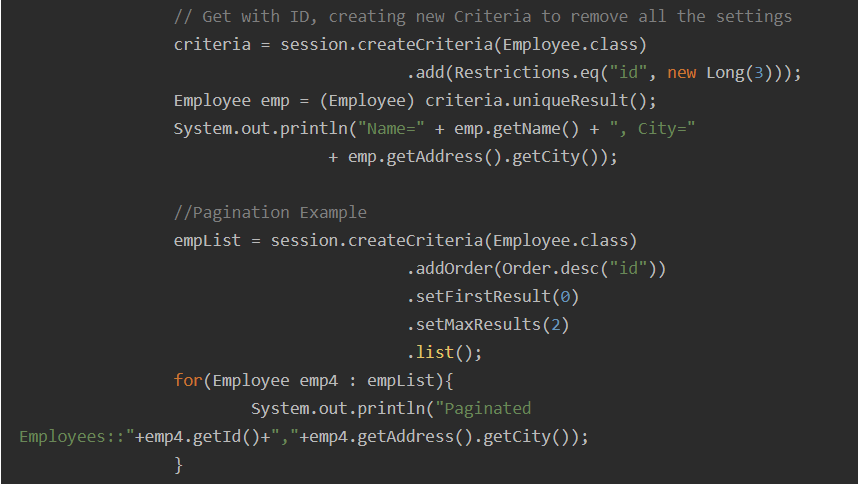


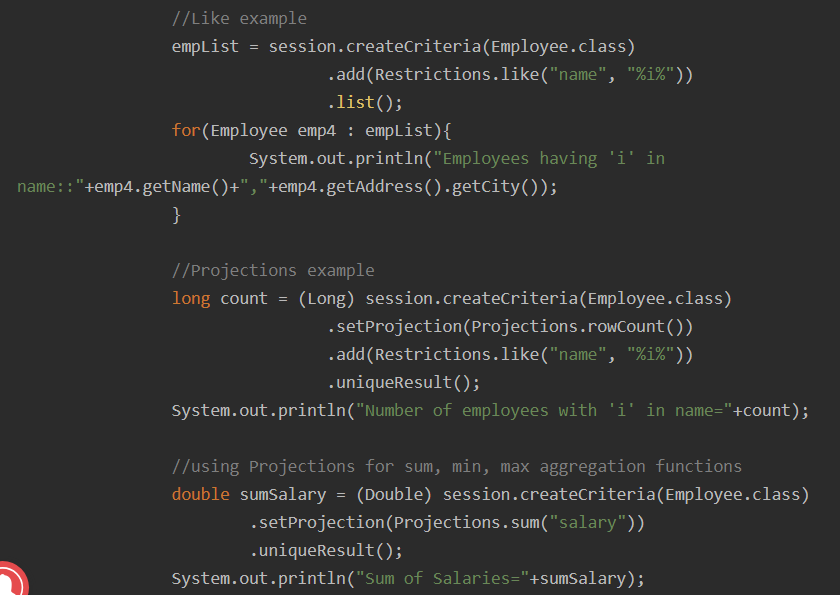




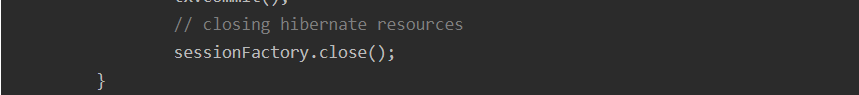
**Example:**

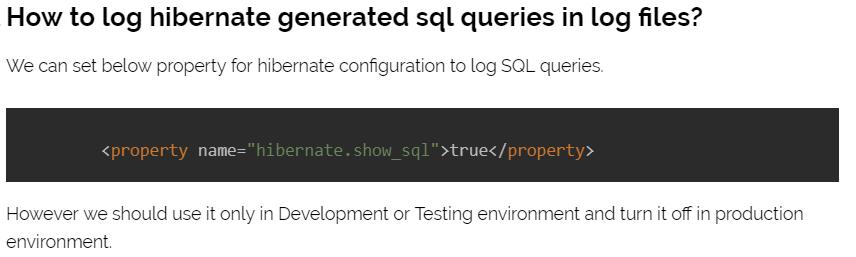


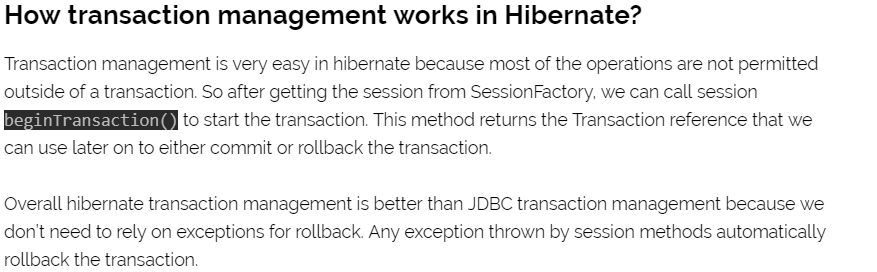


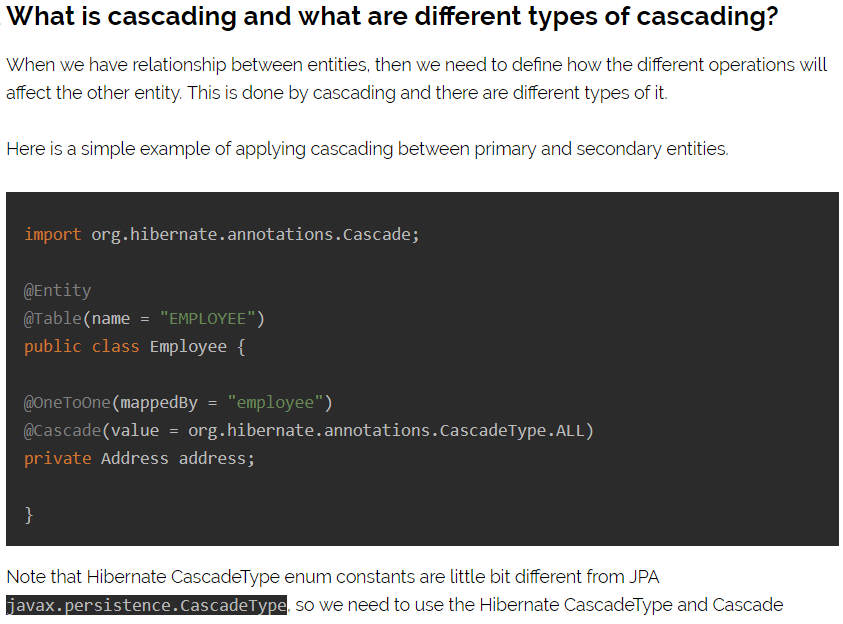




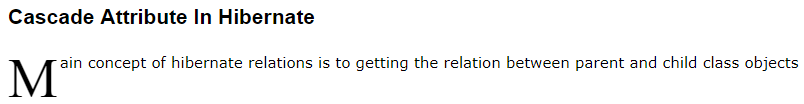


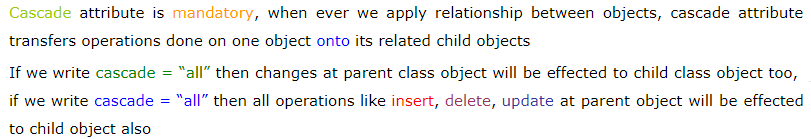


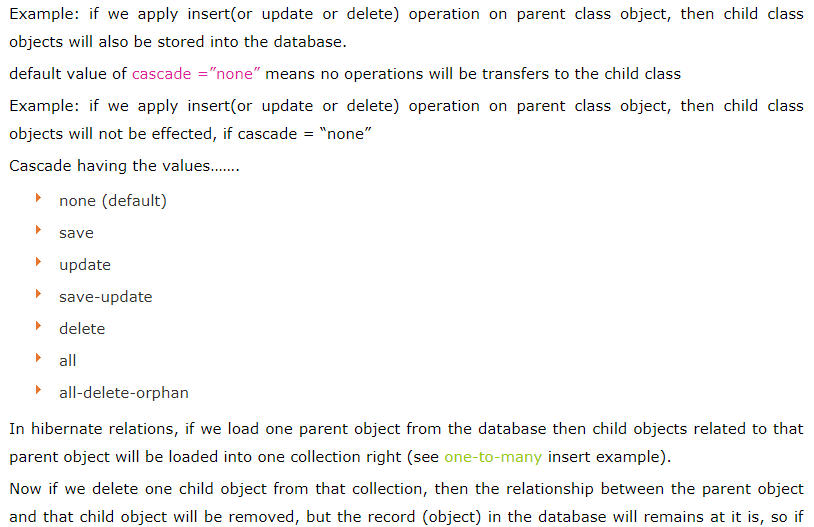


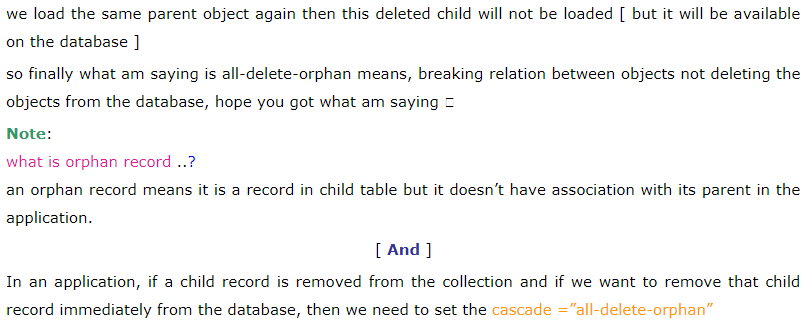


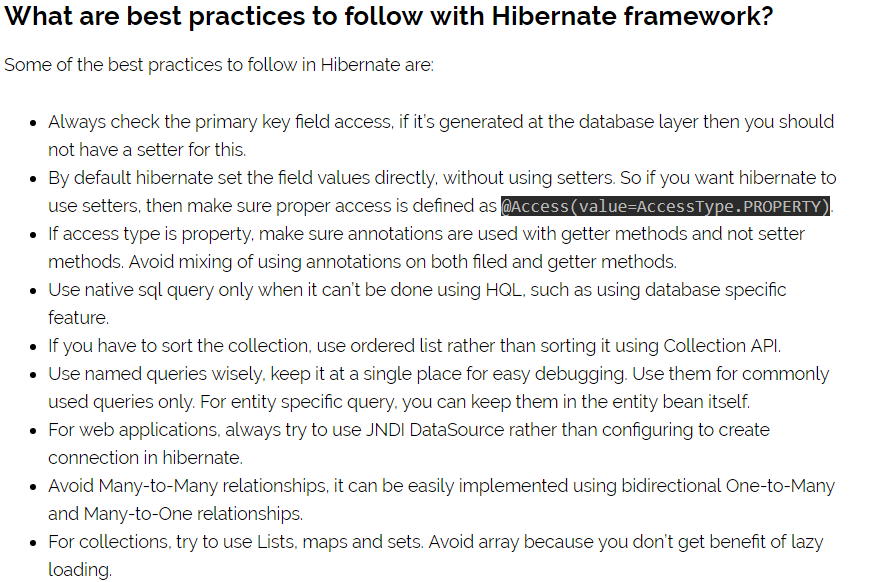


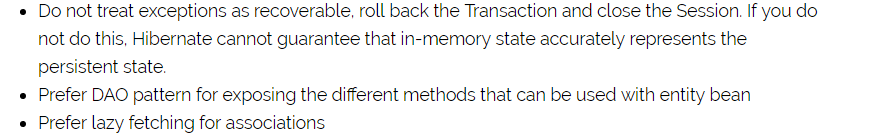


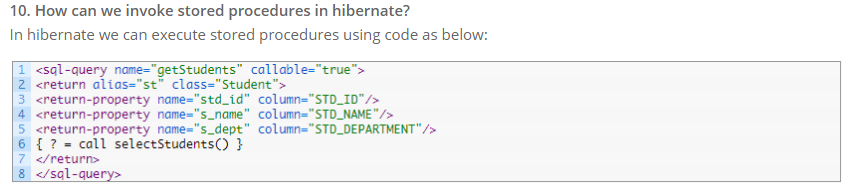


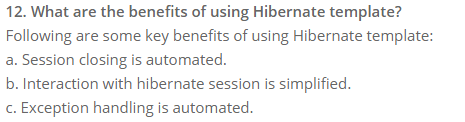


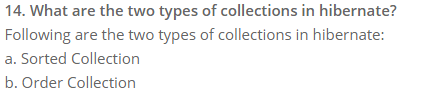


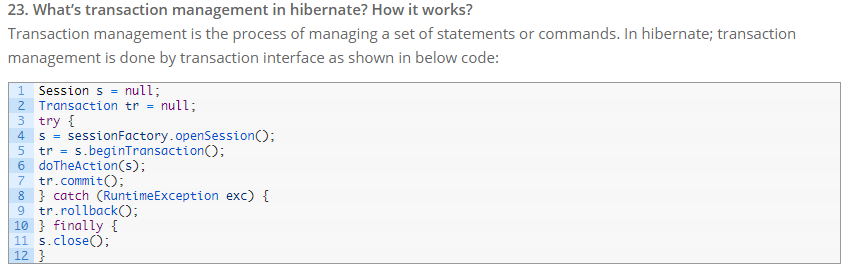


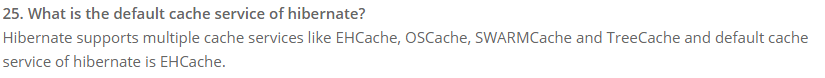


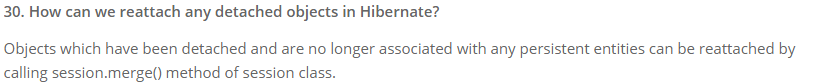


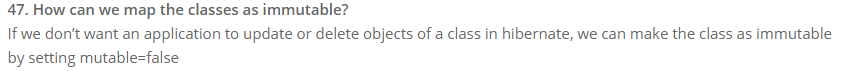




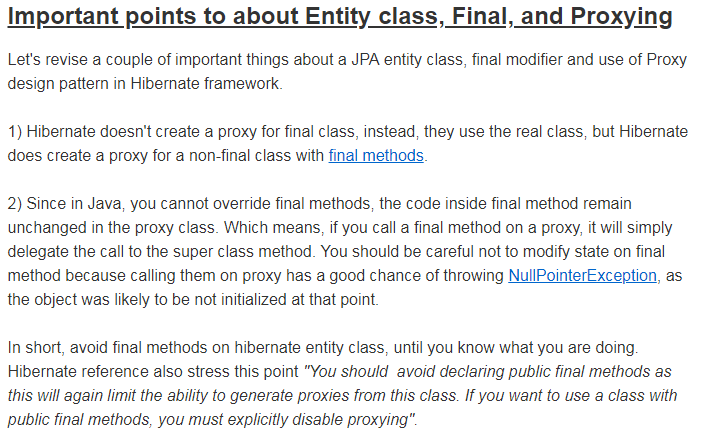


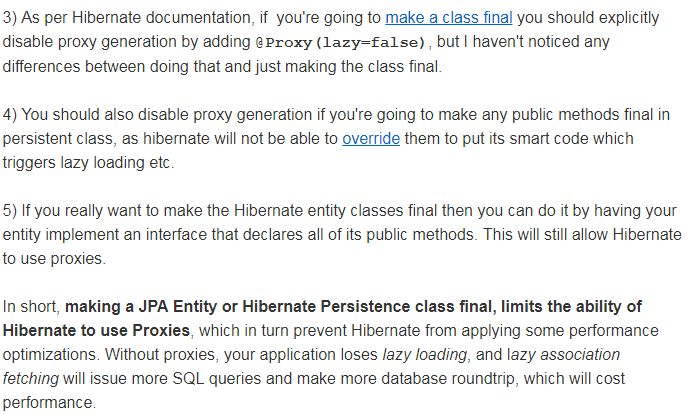


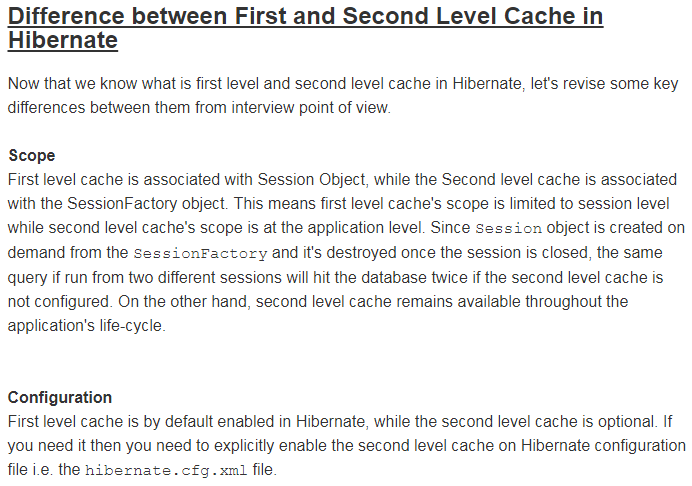


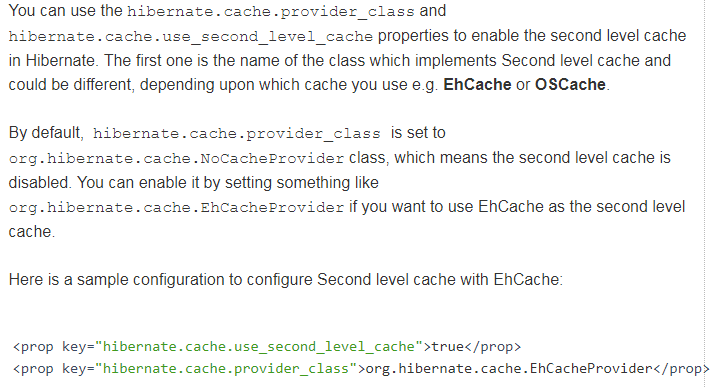


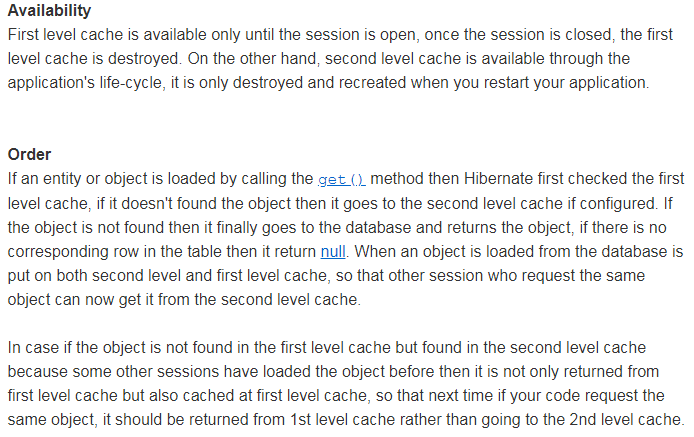


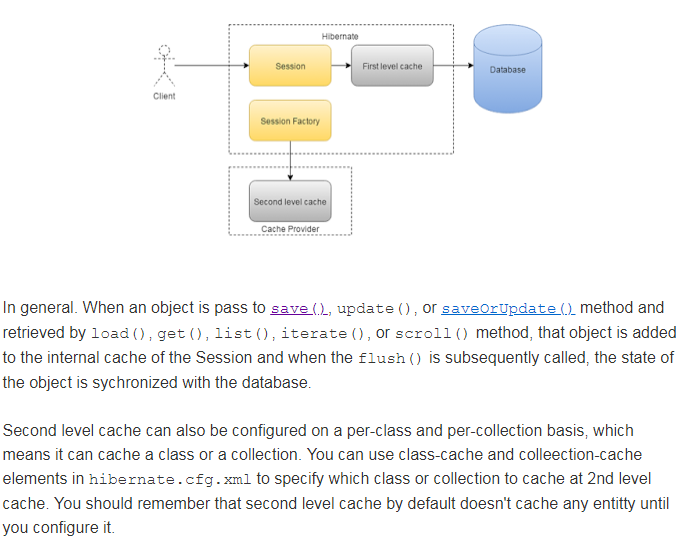


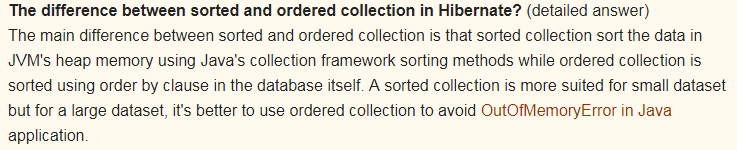




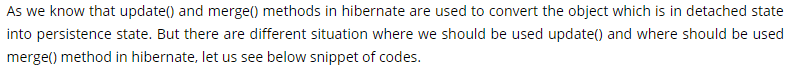


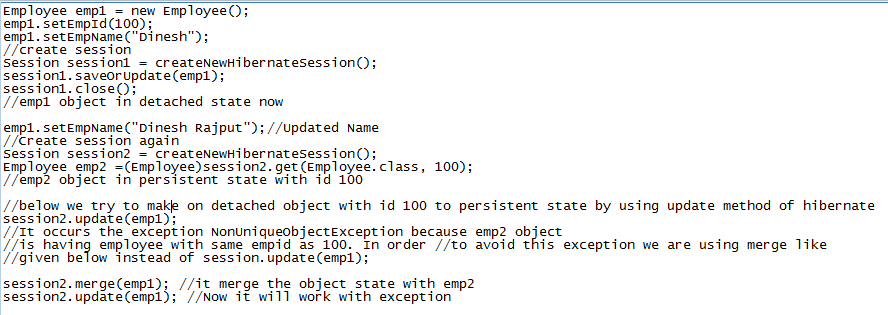


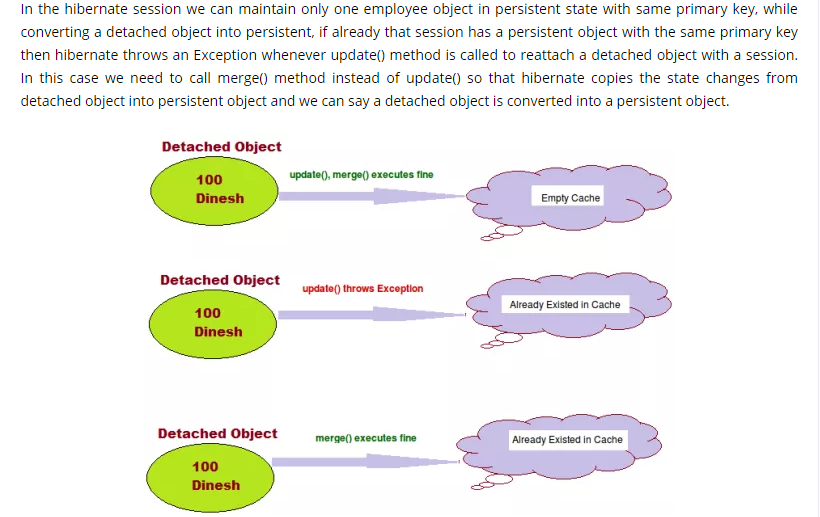


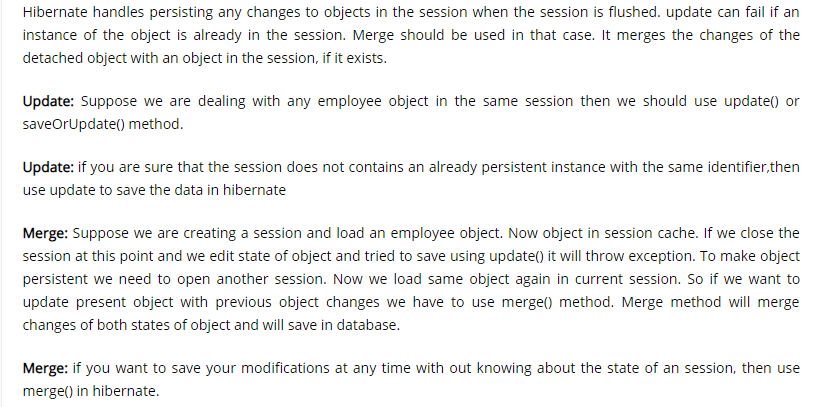


Difference Between Merge And Update Methods In Hibernate:









**Hibernate**: Hibernate is one of the most widely used ORM tool for Java applications. It’s used a lot in enterprise applications for database operations. Hibernate is java based ORM tool that provides framework for mapping application domain objects to the relational database tables and vice versa.

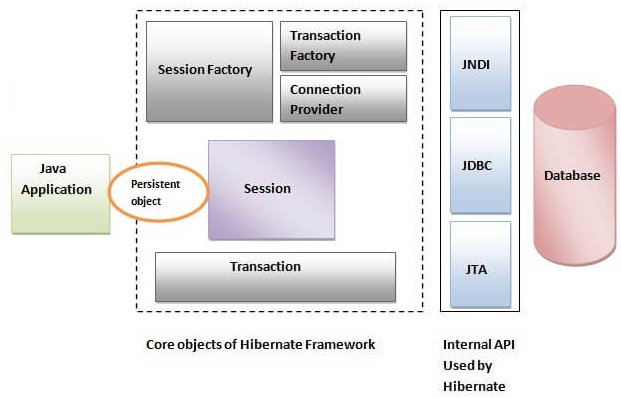
**ORM:** ORM (Object Relational Mapping) is the fundamental concept of Hibernate framework which maps database tables with Java Objects and then provides various API’s to perform different types of operations on the data tables.

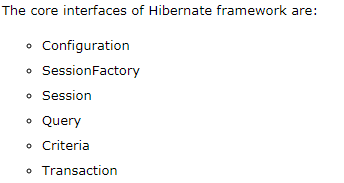
**What are the important benefits of using Hibernate Framework?**

1. Hibernate eliminates all the boiler-plate code that comes with JDBC and takes care of managing resources, so we can focus on business logic.
2. Hibernate framework provides support for XML as well as JPA annotations, that makes our code implementation independent.
3. Hibernate provides a powerful query language (HQL) that is similar to SQL.
4. Hibernate is an open source project from Red Hat Community and used worldwide.
5. Hibernate is easy to integrate with other Java EE frameworks, it’s so popular that Spring Framework provides built-in support for integrating hibernate with Spring applications.
6. Hibernate supports lazy initialization using proxy objects and perform actual database queries only when it’s required.
7. Hibernate cache helps us in getting better performance.

# 

# **Hibernate architecture:**





# **SessionFactory:**

SessionFactory provides the instance of Session. It is a factory of Session. It holds the data of second level cache that is not enabled by default.

SessionFactory is a thread-safe object, many threads cannot access it simultaneously.

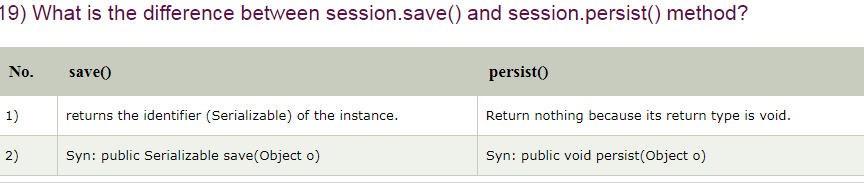
**Session:**

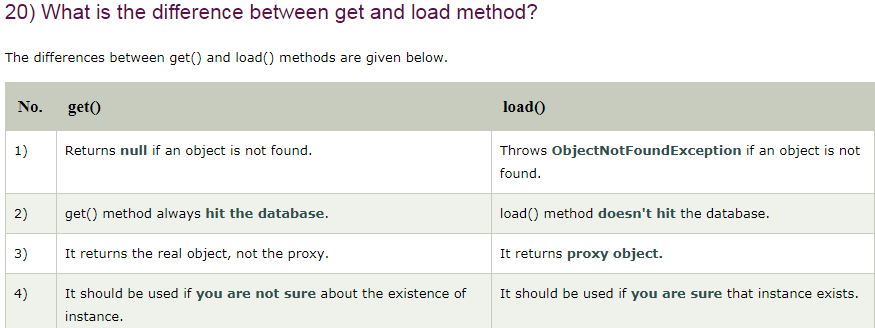
It maintains a connection between the hibernate application and database.

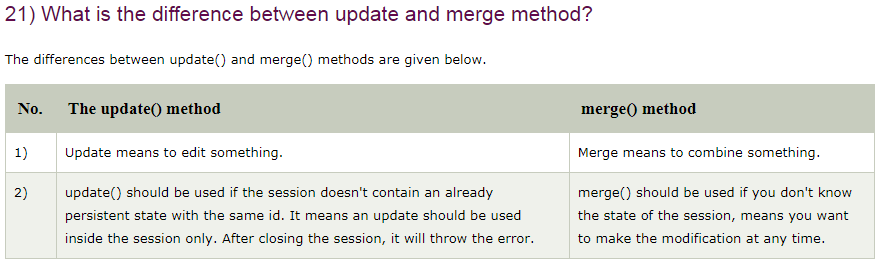
It provides methods to store, update, delete or fetch data from the database such as persist(), update(), delete(), load(), get() etc.

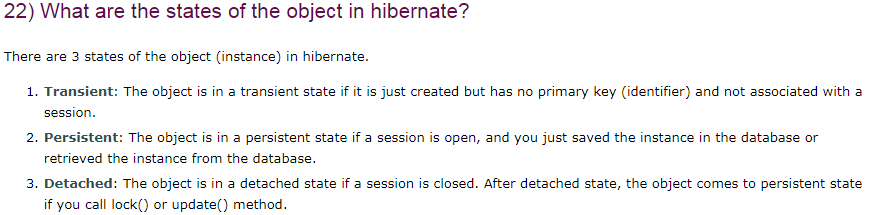
It is a factory of Query, Criteria and Transaction i.e. it provides factory methods to return these instances.

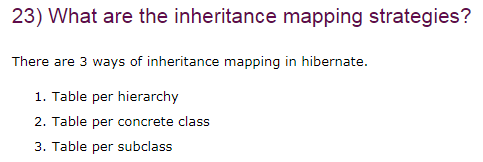
No, Session is not a thread-safe object, many threads can access it simultaneously. In other words, you can share it between threads.

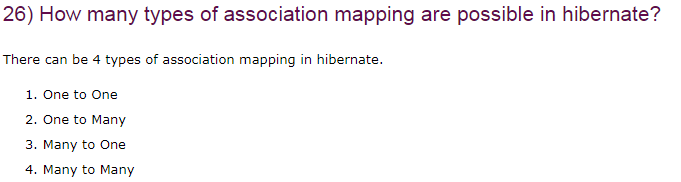


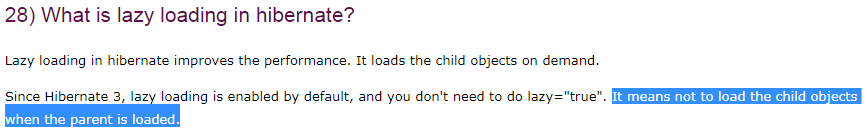


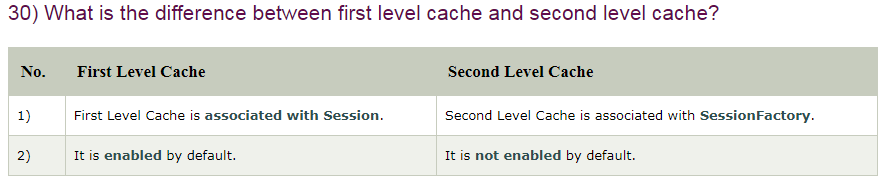




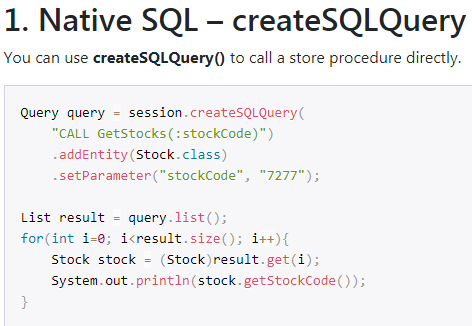


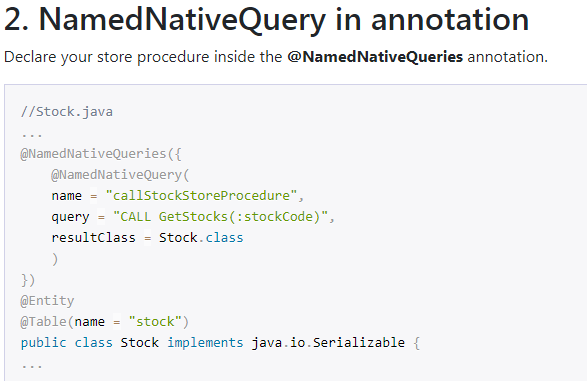


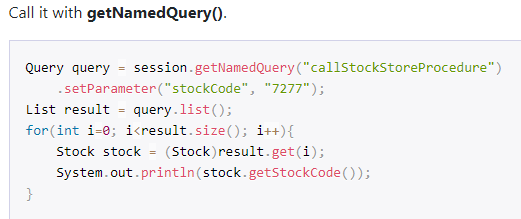


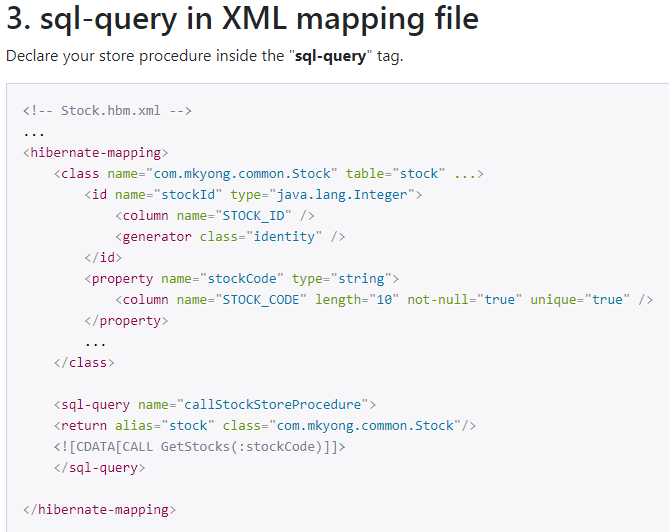


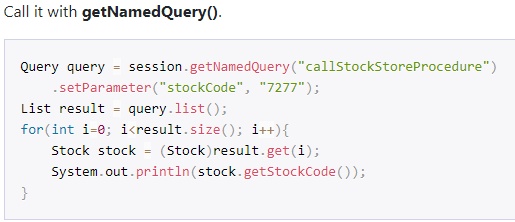
# **Hibernate call store procedure:** there are three approaches to call a database store procedure.







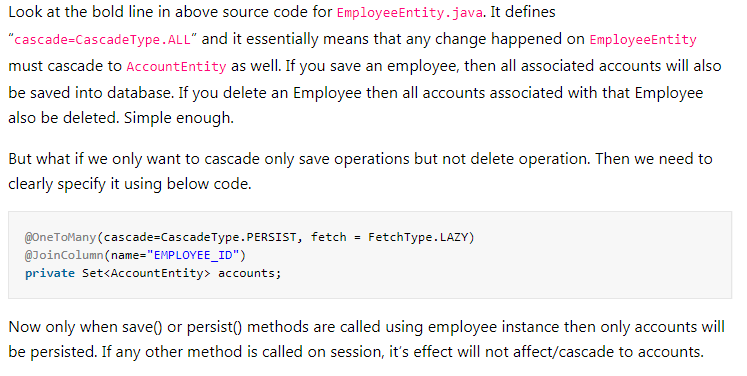


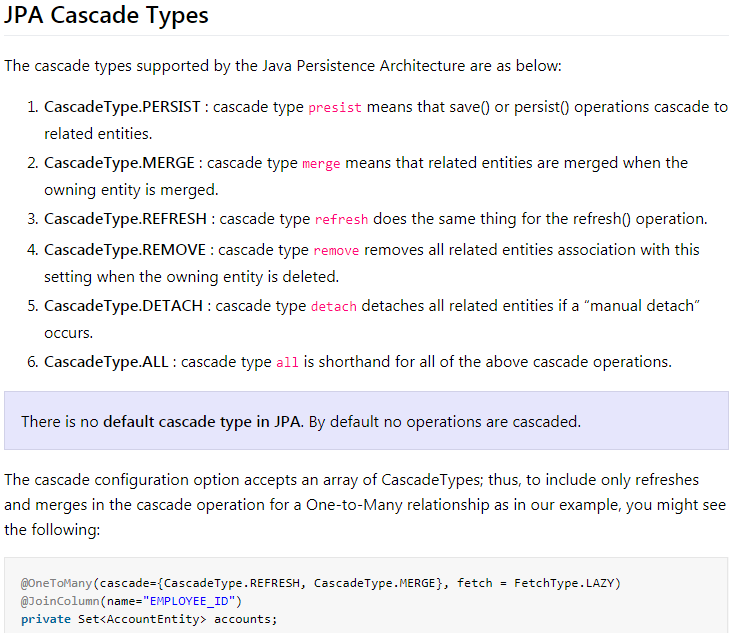


# **Hibernate JPA Cascade Types:**

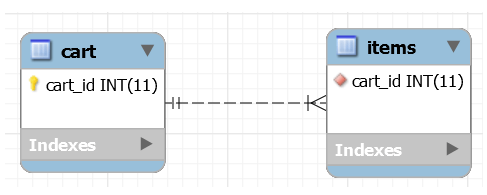
**EmployeeEntity.java**

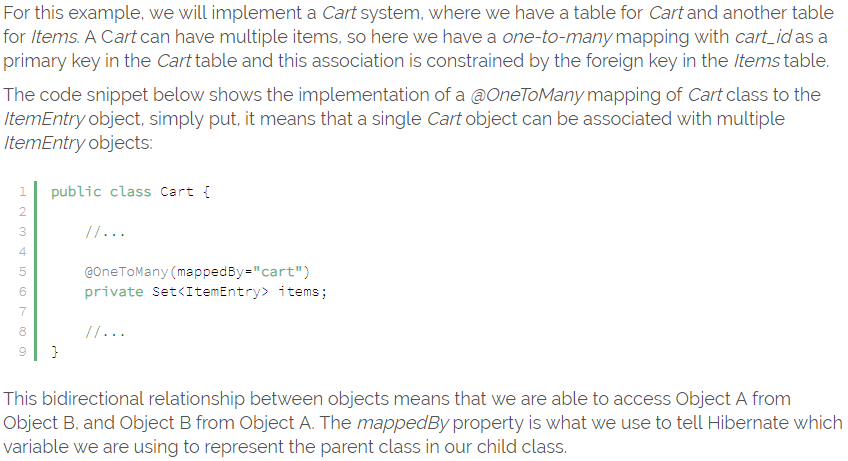
**AccountEntity.java**





# **MappedBy:**

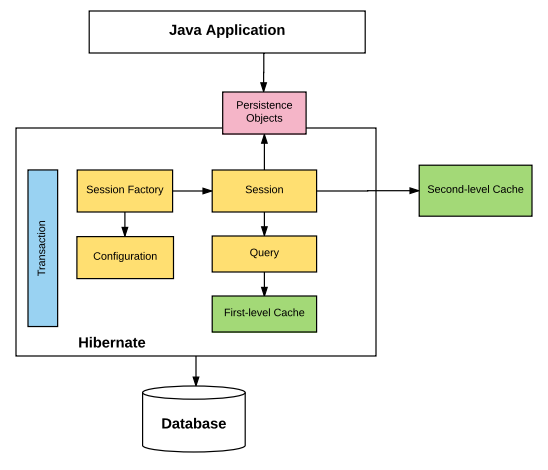




# Introduction:

Hibernate is an open source Java persistence framework project. It perform powerful object relational mapping and query databases using HQL and SQL. Hibernate is a great tool for ORM mappings in java.

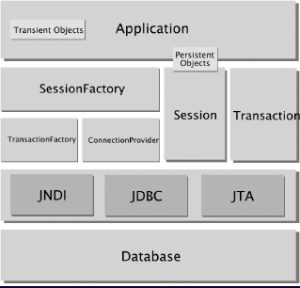
**Architecture:**

****

Let’s understand what each block represents.

1. **Configuration** : Generally written in hibernate.properties or hibernate.cfg.xml files. For Java configuration, you may find class annotated with @Configuration. It is used by Session Factory to work with Java Application and the Database. It represents an entire set of mappings of an application Java Types to an SQL database.
2. **Session Factory** : Any user application requests Session Factory for a session object. Session Factory uses configuration information from above listed files, to instantiates the session object appropriately.
3. **Session** : This represents the interaction between the application and the database at any point of time. This is represented by the org.hibernate.Session class. The instance of a session can be retrieved from the SessionFactory bean.
4. **Query** : It allows applications to query the database for one or more stored objects. Hibernate provides different techniques to query database, including NamedQuery and Criteria API.
5. **First-level cache** : It represents the default cache used by Hibernate Session object while interacting with the database. It is also called as session cache and caches objects within the current session. All requests from the Session object to the database must pass through the first-level cache or session cache. One must note that the first-level cache is available with the session object until the Session object is live.
6. **Transaction** : enables you to achieve data consistency, and rollback incase something goes unexpected.
7. **Persistent objects** : These are plain old Java objects (POJOs), which get persisted as one of the rows in the related table in the database by hibernate.They can be configured in configurations files (hibernate.cfg.xml or hibernate.properties) or annotated with @Entity annotation.
8. **Second-level cache** : It is used to store objects across sessions. This needs to be explicitly enabled and one would be required to provide the cache provider for a second-level cache. One of the common second-level cache providers is *EhCache*.

## Hibernate architecture

s[](https://howtodoinjava.files.wordpress.com/2012/11/arch_hibernate.png)

The above diagram shows a **comprehensive architecture** of Hibernate. Here are some definitions of the objects depicted in the diagrams:

1. **Application:**  
   The main application consist of all user written java files and other resources.
2. [**Transient Objects**](http://docs.jboss.org/hibernate/orm/3.3/reference/en/html/objectstate.html#objectstate-overview)**:**  
   Instances of persistent classes that are not currently associated with a Session. They may have been instantiated by the application and not yet persisted, or they may have been instantiated by a closed Session.
3. [**Persistent objects**](http://docs.jboss.org/hibernate/orm/3.3/reference/en/html/objectstate.html#objectstate-overview)**:**  
   Short-lived, single threaded objects containing persistent state and business function. These can be ordinary Java Beans/POJOs. They are associated with exactly one Session. Once the Session is closed, they will be detached and free to use in any application layer (for example, directly as data transfer objects to and from presentation).
4. [**TransactionFactory**](http://docs.jboss.org/hibernate/orm/3.2/api/org/hibernate/transaction/TransactionFactory.html)**:**  
   (Optional) A factory for Transaction instances. It is not exposed to the application, but it can be extended and/or implemented by the developer.
5. [**ConnectionProvider**](http://docs.jboss.org/hibernate/orm/3.6/javadocs/org/hibernate/connection/ConnectionProvider.html)**:**  
   (Optional) A factory for, and pool of, JDBC connections. It abstracts the application from underlying Data source or DriverManager. It is not exposed to application, but it can be extended and/or implemented by the developer.
6. [**Transaction**](http://docs.jboss.org/hibernate/orm/3.3/reference/en/html/transactions.html)**:**  
   (Optional) A single-threaded, short-lived object used by the application to specify atomic units of work. It abstracts the application from the underlying JDBC, JTA or CORBA transaction. A Session might span several Transactions in some cases. However, transaction demarcation, either using the underlying API or Transaction, is never optional.

## Relation with JPA

[**JPA (Java Persistence API)**](https://en.wikipedia.org/wiki/Java_Persistence_API) is an interface for persistence providers to implement. Hibernate is one such implementation of JPA. You can annotate your classes as much as you would like with JPA annotations, however without an implementation nothing will happen. **Think of JPA as the guidelines that must be followed or an interface, while Hibernates JPA implementation is code that meets the API as defined by JPA and provides the under the hood functionality**.

When you use hibernate with JPA you are actually using the Hibernate JPA implementation. The benefit of this is that you can swap out hibernates implementation of JPA for another implementation of the JPA specification. When you use straight hibernate your locking into the implementation because other ORMs may use different methods/configurations and annotations, therefore you cannot just switch over to another ORM.

# **Hibernate – Load or Get Entity Example**

September 5, 2014 by Lokesh Gupta

In previous tutorials, we learned about [**connecting pooling**](https://howtodoinjava.com/hibernate/hibernate-c3p0-connection-pool-configuration-tutorial/), [**persistence Life-cycle States**](https://howtodoinjava.com/hibernate/hibernate-entity-persistence-lifecycle-states/), [**association Mappings**](https://howtodoinjava.com/hibernate/how-to-define-association-mappings-between-hibernate-entities/) and [**entities Equality and Identity concepts**](https://howtodoinjava.com/hibernate/hibernate-entities-equality-and-identity/). In next some tutorials, I am going to cover some concepts around **CRUD operations** which you will do while interacting with database though [**hibernate**](https://howtodoinjava.com/hibernate-tutorials/). Let’s start the discussion with loading an hibernate entity using either load or get method.

**1) Loading an hibernate entity using session.load() method**

Hibernate’s Session interface provides several load() methods for loading entities from your database. Each load() method requires the object’s primary key as an identifier, and it is mandatory to provide it. In addition to the ID, Hibernate also needs to know which class or entity name to use to find the object with that ID. After the load() method returns, you need to cast the returned object to suitable type of class to further use it. It’s all what load() method need from you to work it correctly.

Let’s look at different flavors of load() method available in hibernate session:

1. public Object load(Class theClass, Serializable id) throws HibernateException
2. public Object load(String entityName, Serializable id) throws HibernateException
3. public void load(Object object, Serializable id) throws HibernateException

First load() method need the class type which you would like to load along with unique ID. Second load() method asks for **entityName** directly and unique ID. Both method return the populated entity object as return value which you will cast to desired type.

The last load() method takes an object as an argument. The object should be of the same class as the object you would like loaded, and it should be empty. Hibernate will populate that object with the object you requested.

The other load() methods available through hibernate session take a lock mode as an argument too. The lock mode specifies whether Hibernate should look into the cache for the object and which database lock level Hibernate should use for the row (or rows) of data that represent this object. In official documentation, hibernate developers claim that *Hibernate will usually pick the correct lock mode* for you, although in some situations it is important to manually choose the correct lock.

We will discuss more about locks when we will be done with basic hibernate concepts.

Let’s look at the examples of each load method in simplest form to be clear what we read above.

|  |
| --- |
| public class TestHibernate  {     public static void main(String[] args)     {        Session sessionOne = HibernateUtil.getSessionFactory().openSession();        sessionOne.beginTransaction();          // Create new Employee object        EmployeeEntity emp = new EmployeeEntity();        emp.setFirstName("Lokesh");        emp.setLastName("Gupta");          //Save employee        sessionOne.save(emp);        //store the employee id generated for future use        Integer empId = emp.getEmployeeId();        sessionOne.getTransaction().commit();          /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/          //Let's open a new session to test load() methods        Session sessionTwo = HibernateUtil.getSessionFactory().openSession();        sessionTwo.beginTransaction();          //first load() method example        EmployeeEntity emp1 = (EmployeeEntity) sessionTwo.load(EmployeeEntity.class, empId);        System.out.println(emp1.getFirstName() + " - " +emp1.getLastName());          //Let's verify the entity name        System.out.println(sessionTwo.getEntityName(emp1));          sessionTwo.getTransaction().commit();          /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/          Session sessionThree = HibernateUtil.getSessionFactory().openSession();        sessionThree.beginTransaction();          //second load() method example        EmployeeEntity emp2 = (EmployeeEntity) sessionThree.load("com.howtodoinjava.demo.entity.EmployeeEntity", empId);        System.out.println(emp2.getFirstName() + " - " +emp2.getLastName());          sessionThree.getTransaction().commit();          /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/          Session sessionFour = HibernateUtil.getSessionFactory().openSession();        sessionFour.beginTransaction();          //third load() method example        EmployeeEntity emp3 = new EmployeeEntity();        sessionFour.load(emp3 , empId);        System.out.println(emp3.getFirstName() + " - " +emp3.getLastName());          sessionFour.getTransaction().commit();          HibernateUtil.shutdown();     }  }      Output:    Hibernate: insert into Employee (FIRST\_NAME, LAST\_NAME, ID) values (?, ?, ?)  Hibernate: select employeeen0\_.ID as ID1\_1\_0\_, employeeen0\_.FIRST\_NAME as FIRST\_NA2\_1\_0\_, employeeen0\_.LAST\_NAME as LAST\_NAM3\_1\_0\_              from Employee employeeen0\_ where employeeen0\_.ID=?  Lokesh - Gupta      //First load method  com.howtodoinjava.demo.entity.EmployeeEntity  Hibernate: select employeeen0\_.ID as ID1\_1\_0\_, employeeen0\_.FIRST\_NAME as FIRST\_NA2\_1\_0\_, employeeen0\_.LAST\_NAME as LAST\_NAM3\_1\_0\_              from Employee employeeen0\_ where employeeen0\_.ID=?  Lokesh - Gupta      //Second load method  Hibernate: select employeeen0\_.ID as ID1\_1\_0\_, employeeen0\_.FIRST\_NAME as FIRST\_NA2\_1\_0\_, employeeen0\_.LAST\_NAME as LAST\_NAM3\_1\_0\_              from Employee employeeen0\_ where employeeen0\_.ID=?  Lokesh - Gupta      //Third load method |

So we are able to load the entity from all three load methods successfully. Now move on to get()method.

**2) Loading an hibernate entity using session.get() method**

The get() method is very much similar to load() method. The get() methods take an identifier and either an entity name or a class. There are also two get() methods that take a lock mode as an argument, but we will discuss lock modes later. The rest get() methods are as follows:

1. public Object get(Class clazz, Serializable id) throws HibernateException
2. public Object get(String entityName, Serializable id) throws HibernateException

There is not much difference in code while working with either load() or get() method, all you need is to replace the load() method with get() method in first two examples. There is no get() equivalent to last load() method.

You can modify above example and test the code. Let me know if you find any problem.

**Difference between load() and get() method in hibernate session**

Why we have two method to do the same job. Actually this is frequently asked [**interview question**](https://howtodoinjava.com/java-interview-questions/) as well.

**Read More**: [Java Interview Question](https://howtodoinjava.com/java-interview-questions/)

The difference between both methods lies in return value “if the identifier does not exist in database”. **In case of get() method you will get return value as NULL if identifier is absent; But in case of load() method, you will get a runtime exception** something like below:

**Hibernate 4 – Get Entity Reference for Lazy Loading**

August 6, 2013 by Lokesh Gupta

By Wikipedia definition, [**Lazy loading**](https://en.wikipedia.org/wiki/Lazy_loading) is a design pattern commonly used in computer programming to defer initialization of an object until the point at which it is needed. We know that in hibernate lazy loading can be done by specifying **“fetch= FetchType.LAZY”** in hibernate mapping annotations. e.g.

|  |
| --- |
| @ManyToOne ( fetch = FetchType.LAZY )  @JoinColumns( {          @JoinColumn(name="fname", referencedColumnName = "firstname"),          @JoinColumn(name="lname", referencedColumnName = "lastname")          } )  public EmployeeEntity getEmployee() {      return employee;  } |

The point is that it is applied **only when you are defining mapping between two entities**. If above entity has been defined in DepartmentEntity then if you fetch DepartmentEntity then EmployeeEntity will be lazy loaded.

But, what if you want to lazy load DepartmentEntity itself i.e. **master entity itself should be lazy loaded**.

This problem can be solved by using [**getReference()**](http://docs.jboss.org/hibernate/orm/4.1/javadocs/org/hibernate/IdentifierLoadAccess.html#getReference%28java.io.Serializable%29) method inside **IdentifierLoadAccess** class.

Let’s understand the usage by this example.

For reference, latest hibernate maven dependency is as follows:

|  |
| --- |
| <dependency>      <groupId>org.hibernate</groupId>      <artifactId>hibernate-core</artifactId>      <version>4.3.0.Beta3</version>  </dependency> |

Now, I have an master entity class EmployeeEntity which can have multiple attributes and mapping with other entities.

**EmployeeEntity.java**

|  |
| --- |
| @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  } |

I want to lazy load above master entity lazy loaded in my code i.e. I can get reference of entity in one place but might be actually needing it another place. Only when I need it, I want to initialize or load its data. Till the time, I want only the reference.

Let’s do this in code example:

**TestHibernate.java**

|  |
| --- |
| 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);            session.getTransaction().commit();          session.close();            session = HibernateUtil.getSessionFactory().openSession();          session.beginTransaction();            //Get only the reference of EmployeeEntity for now          EmployeeEntity empGet = (EmployeeEntity) session.byId( EmployeeEntity.class).getReference( 1 );            System.out.println("No data initialized till now; Lets fetch some data..");              //Now EmployeeEntity will be loaded from database when we need it          System.out.println(empGet.getFirstName());          System.out.println(empGet.getLastName());            session.getTransaction().commit();          HibernateUtil.shutdown();      }  }    Output in console:    Hibernate: insert into Employee (EMAIL, FIRST\_NAME, LAST\_NAME) values (?, ?, ?)    No data initialized till now; Lets fetch some data..    Hibernate: select employeeen0\_.ID as ID1\_0\_0\_, employeeen0\_.EMAIL as EMAIL2\_0\_0\_, employeeen0\_.FIRST\_NAME as FIRST3\_0\_0\_,  employeeen0\_.LAST\_NAME as LAST4\_0\_0\_ from Employee employeeen0\_ where employeeen0\_.ID=?    demo  user |

**Hibernate Save() and saveOrUpdate() Methods**

September 22, 2014 by Lokesh Gupta

We have learned that [**hibernate**](https://howtodoinjava.com/hibernate-tutorials/) works only with[**persistent entities**](https://howtodoinjava.com/hibernate/hibernate-entity-persistence-lifecycle-states/) and persistent entities are classes which are attached to any hibernate session. Please not that creating an instance of a class, you mapped with a hibernate annotations, does not automatically persist the object to the database. It must be save explicitly after attaching it to a valid hibernate session.

**Using save() method**

In hibernate, we generally use one of below two versions of save() method:

|  |
| --- |
| public Serializable save(Object object) throws HibernateException  public Serializable save(String entityName,Object object) throws HibernateException |

Both **save()** methods take a transient object reference (which must not be null) as an argument. Second method takes an extra parameter ‘***entityName***‘ which is useful in case you have mapped multiple entities to a Java class. Here you can specify which entity you are saving using **save()** method.

Let’s write down a simple example to see above theory in practical:

**EmployeeEntity.java**

|  |
| --- |
| @Entity  @Table(name = "Employee")  public class EmployeeEntity implements Serializable  {     private static final long serialVersionUID = -1798070786993154676L;     @Id     @Column(name = "ID", unique = true, nullable = false)     private Integer  employeeId;       @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;       @Override     public boolean equals(Object o) {         if (this == o) return true;         if (!(o instanceof EmployeeEntity)) return false;           EmployeeEntity otherEmployee = (EmployeeEntity) o;           if (getEmployeeId() != null ?             !getEmployeeId().equals(otherEmployee.getEmployeeId()) : otherEmployee.getEmployeeId() != null)             return false;         if (getFirstName() != null ?             !getFirstName().equals(otherEmployee.getFirstName()) : otherEmployee.getFirstName() != null)             return false;         if (getLastName() != null ?             !getLastName().equals(otherEmployee.getLastName()) : otherEmployee.getLastName() !=null)             return false;           return true;     }       @Override     public int hashCode() {     int result = getEmployeeId() != null ? getEmployeeId().hashCode() : 0;         result = 31 \* result + (getFirstName() != null ? getFirstName().hashCode() : 0);         result = 31 \* result + (getLastName() != null?getLastName().hashCode() : 0);         return result;     }       //Getters and Setters are hidden here  } |

Now let’s save this entity.

**SimplestSaveEntityExample.java**

|  |
| --- |
| public class SimplestSaveEntityExample  {     public static void main(String[] args)     {        Session sessionOne = HibernateUtil.getSessionFactory().openSession();        sessionOne.beginTransaction();          //Create new Employee object        EmployeeEntity emp = new EmployeeEntity();        emp.setEmployeeId(1);        emp.setFirstName("Lokesh");        emp.setLastName("Gupta");          //Save employee        sessionOne.save(emp);          sessionOne.getTransaction().commit();        HibernateUtil.shutdown();     }  }    Output:    Hibernate: insert into Employee (FIRST\_NAME, LAST\_NAME, ID) values (?, ?, ?) |

We got our Employee entity saved. So easy. But in reality, it is not so simple usecase. There you may need to update again employee entity and then save again in another session. Should you call save()method again? Let’s check out.

|  |
| --- |
| public class SaveEntityAgainInAnotherSession  {     public static void main(String[] args)     {        Session sessionOne = HibernateUtil.getSessionFactory().openSession();        sessionOne.beginTransaction();          //Create new Employee object        EmployeeEntity emp = new EmployeeEntity();        emp.setEmployeeId(1);        emp.setFirstName("Lokesh");        emp.setLastName("Gupta");          //Save employee        sessionOne.save(emp);        sessionOne.getTransaction().commit();          Session sessionTwo = HibernateUtil.getSessionFactory().openSession();        sessionTwo.beginTransaction();          emp.setLastName("temp");        //Save employee again second time        sessionTwo.save(emp);          sessionTwo.getTransaction().commit();        HibernateUtil.shutdown();     }  }    Hibernate: insert into Employee (FIRST\_NAME, LAST\_NAME, ID) values (?, ?, ?)  Hibernate: insert into Employee (FIRST\_NAME, LAST\_NAME, ID) values (?, ?, ?)   WARN SqlExceptionHelper:144 - SQL Error: -104, SQLState: 23000  ERROR SqlExceptionHelper:146 - Violation of unique constraint SYS\_PK\_49: duplicate value(s) forcolumn(s) ID in statement [insert into Employee (FIRST\_NAME, LAST\_NAME, ID) values (?, ?, ?)]   INFO AbstractBatchImpl:208 - HHH000010: On release of batch it still contained JDBC statements |

Here hibernate tried to insert the entity again. Though it was **failed due to primary key check**, but check may not be there for other entities and you may end up with duplicate rows.

***Note:*** While second save() method causes duplicate row in different sessions, BUT in same session they will work correct. Look at below example.

|  |
| --- |
| public class SaveEntityAgainInSameSession  {     public static void main(String[] args)     {        Session sessionOne = HibernateUtil.getSessionFactory().openSession();        sessionOne.beginTransaction();          //Create new Employee object        EmployeeEntity emp = new EmployeeEntity();        emp.setEmployeeId(1);        emp.setFirstName("Lokesh");        emp.setLastName("Gupta");          //Save employee        sessionOne.save(emp);          emp.setLastName("temp");        //Save employee again second time        sessionOne.save(emp);          sessionOne.getTransaction().commit();        HibernateUtil.shutdown();     }  }    Hibernate: insert into Employee (FIRST\_NAME, LAST\_NAME, ID) values (?, ?, ?)  Hibernate: update Employee set FIRST\_NAME=?, LAST\_NAME=? where ID=? |

It’s confusing. Right? Let’s make it simple. And rule is below:

Remember that you should not call save() method on a persistent entity (entity associated with any hibernate session). Any changes done to persistent entity is automatically saved.

Let’s understand this concept in simple example:

|  |
| --- |
| public class NoSaveCallForPersistentEntity  {     public static void main(String[] args)     {        Session sessionOne = HibernateUtil.getSessionFactory().openSession();        sessionOne.beginTransaction();          //Create new Employee object        EmployeeEntity emp = new EmployeeEntity();        emp.setEmployeeId(1);        emp.setFirstName("Lokesh");        emp.setLastName("Gupta");          //Save employee        sessionOne.save(emp);          emp.setLastName("temp");          sessionOne.getTransaction().commit();          //Let's see what got updated in DB        Session sessionTwo = HibernateUtil.getSessionFactory().openSession();        sessionTwo.beginTransaction();          EmployeeEntity employee = ( EmployeeEntity ) sessionTwo.load(EmployeeEntity.class, 1);        System.out.println(employee.getLastName());          sessionTwo.getTransaction().commit();        HibernateUtil.shutdown();     }  }    Output:    Hibernate: insert into Employee (FIRST\_NAME, LAST\_NAME, ID) values (?, ?, ?)  Hibernate: update Employee set FIRST\_NAME=?, LAST\_NAME=? where ID=?    Hibernate: select employeeen0\_.ID as ID1\_1\_0\_, employeeen0\_.FIRST\_NAME as FIRST\_NA2\_1\_0\_,  employeeen0\_.LAST\_NAME as LAST\_NAM3\_1\_0\_ from Employee employeeen0\_ where employeeen0\_.ID=?    temp |

In above example, we made the ‘emp‘ object persistent using first save() method. Afterward when we updated the last name to ‘***temp***‘, an update query was executed as expected. This we verified in returned data as well. This is the correct way to work with persistent entities.

**Using saveOrUpdate() method**

In discussion of save() method, we forgot about case where we had to save persistent entity in another session and that got resulted in duplicate key error. That is also a valid scenario.

To handle such cases, you must use saveOrUpdate() method. Strictly speaking, you should use saveOrUpdate() with even non-persistent entities. Personally, I do not see any harm in doing so. Though, It may make you a little bit careless. So be cautious.

Let’s see how it can be used along with entity persisted with save() method.

|  |
| --- |
| public class SaveOrUpdateMethodExample  {     public static void main(String[] args)     {        Session sessionOne = HibernateUtil.getSessionFactory().openSession();        sessionOne.beginTransaction();          //Create new Employee object        EmployeeEntity emp = new EmployeeEntity();        emp.setEmployeeId(1);        emp.setFirstName("Lokesh");        emp.setLastName("Gupta");          //Save employee        sessionOne.save(emp);        sessionOne.getTransaction().commit();          Session sessionTwo = HibernateUtil.getSessionFactory().openSession();        sessionTwo.beginTransaction();          emp.setLastName("temp");        //Save employee again second time        sessionTwo.saveOrUpdate(emp);          sessionTwo.getTransaction().commit();        HibernateUtil.shutdown();     }  }    Output:    Hibernate: insert into Employee (FIRST\_NAME, LAST\_NAME, ID) values (?, ?, ?)  Hibernate: select employeeen\_.ID, employeeen\_.FIRST\_NAME as FIRST\_NA2\_1\_, employeeen\_.LAST\_NAME as          LAST\_NAM3\_1\_ from Employee employeeen\_ where employeeen\_.ID=?  Hibernate: update Employee set FIRST\_NAME=?, LAST\_NAME=? where ID=? |

Now we are able to save the entity as well as update the entity as well using saveOrUpdate() method.

Please remember that if you have used saveOrUpdate() method in place of save() method above, then also result would have been same. saveOrUpdate() can be used with persistent as well as non-persistent entities both. Persistent entities will get updated, and transient entities will be inserted into database.

**Suggestion For Production Code**

It wouldn’t be advisable to try to match this code construction in production code. Ideally, what you would do is pass VO object to DAO layer, load the entity from the session and update the entity with by copying VO data onto it. This means that the updates take place on a persistent object, and we don’t actually have to call Session.save() or Session.saveOrUpdate() at all.

Once an object is in a persistent state, Hibernate manages updates to the database itself as you change the fields and properties of the object. It’s big relief.

**Bullet Points**

1. Save() method stores an object into the database. It will Persist the given transient instance, first assigning a generated identifier. It returns the id of the entity created.
2. SaveOrUpdate() calls either save() or update() on the basis of identifier exists or not. e.g if identifier does not exist, save() will be called or else update() will be called.
3. Probably you will get very few chances to actually call save() or saveOrUpdate() methods, as hibernate manages all changes done in persistent objects.

Let me know is something is not clear or needs more explanation.

**Hibernate Entities Equality and Identity**

September 5, 2014 by Lokesh Gupta

Many times in our application, we face a situation where we have to compare two objects to check their equality for satisfying some business rules. In core java, we have already much knowledge about checking equality of objects, but in hibernate we need to take care of few extra things as well. Let’s learn what are those extra concepts.

We already learned about various [**states of hibernate entities in their life-cycle**](https://howtodoinjava.com/hibernate/hibernate-entity-persistence-lifecycle-states/). There we discussed that hibernate mostly work with persistent objects only. As we know that when we have a persistent object in hibernate, that object represents both :

* An instance of a class in a particular Java virtual machine (JVM)
* A row (or rows) in a database table (or tables)

We know enough around first concept. I will focus on second point.

**Objects fetched from same session**

Requesting a persistent object again **from the same Hibernate session returns the same Java instance of a class**, which means that you can compare the objects using the standard Java ‘==’ equality syntax.

Let’s see a quick example:

|  |
| --- |
| public static void main(String[] args)  {      Session sessionOne = HibernateUtil.getSessionFactory().openSession();      sessionOne.beginTransaction();        // Create new Employee object      EmployeeEntity emp = new EmployeeEntity();      emp.setFirstName("Lokesh");      emp.setLastName("Gupta");      //Save employee      sessionOne.save(emp);        sessionOne.getTransaction().commit();        //Get employee id      Integer genEmpId = emp.getEmployeeId();        //New session where we will fetch the employee two times and compare the objects      Session sessionTwo = HibernateUtil.getSessionFactory().openSession();      sessionTwo.beginTransaction();        EmployeeEntity employeeObj1 = (EmployeeEntity) sessionTwo.get(EmployeeEntity.class, genEmpId);      EmployeeEntity employeeObj2 = (EmployeeEntity) sessionTwo.get(EmployeeEntity.class, genEmpId);        //Checking equality      System.out.println(employeeObj1 == employeeObj2);        HibernateUtil.shutdown();  }    Output:    true |

You see above that we got two instances on EmployeeEntity and both are actually same java object instance.

**Objects fetched from different sessions**

If you request a persistent object from **more than one Hibernate session, Hibernate will provide distinct instances from each session**, and the == operator will return false if you compare these object instances.

Let’s compare instances “emp” and “employeeObj1” in above example and you will get the result as false; because both are fetched in separate sessions.

|  |
| --- |
| System.out.println(emp == employeeObj1);  System.out.println(emp.equals(employeeObj1));    Output:    false  false |

So if you are comparing objects in two different sessions, you will need to implement the equals()method on your Java persistence objects, which you should do as a regular occurrence anyway. (Just don’t forget to override hashCode() along with it.)

**Read More:** [Overriding hashCode() and equals() methods](https://howtodoinjava.com/core-java/related-concepts/working-with-hashcode-and-equals-methods-in-java/)

Hibernate wraps the actual object in a proxy so always use the getter methods inside instead of actual properties to compare.

Now let’s add equals() method as suggested and then see the behavior change while checking the equality of both instances on EmployeeEntity.

|  |
| --- |
| @Entity  @Table(name = "Employee")  public class EmployeeEntity implements Serializable  {     private static final long serialVersionUID = -1798070786993154676L;     @Id     @Column(name = "ID", unique = true, nullable = false)     @GeneratedValue(strategy = GenerationType.SEQUENCE)     private Integer           employeeId;     @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;       @Override     public boolean equals(Object o) {         if (this == o) return true;         if (!(o instanceof EmployeeEntity)) return false;           EmployeeEntity otherEmployee = (EmployeeEntity) o;           if (getEmployeeId() != null ?             !getEmployeeId().equals(otherEmployee.getEmployeeId()) : otherEmployee.getEmployeeId() != null)             return false;         if (getFirstName() != null ?             !getFirstName().equals(otherEmployee.getFirstName()) : otherEmployee.getFirstName() != null)             return false;         if (getLastName() != null ?             !getLastName().equals(otherEmployee.getLastName()) : otherEmployee.getLastName() !=null)             return false;           return true;     }       @Override     public int hashCode() {     int result = getEmployeeId() != null ? getEmployeeId().hashCode() : 0;         result = 31 \* result + (getFirstName() != null ? getFirstName().hashCode() : 0);         result = 31 \* result + (getLastName() != null?getLastName().hashCode() : 0);         return result;     }       //Setters and Getters  } |

Now lets again check the equality using equals() method. [‘==’ will return false, we know that].

|  |
| --- |
| System.out.println(emp.equals(employeeObj1));    Output:    true |

Now both objects are equal logically as well as programmatically.

**Bullet Points**

1. Requesting a persistent object again from the same Hibernate session returns the “same java instance” of a class.
2. Requesting a persistent object from the different Hibernate session returns “different java instance” of a class.
3. As a best practice, always implement [equals() and hashCode()](https://howtodoinjava.com/core-java/related-concepts/working-with-hashcode-and-equals-methods-in-java/) methods in your hibernate entities; and always compare them using equals() method only.

**Hibernate JPA Cascade Types**

September 25, 2014 by Lokesh Gupta

We learned about [**mapping associated entities**](https://howtodoinjava.com/hibernate/how-to-define-association-mappings-between-hibernate-entities/) in hibernate already in previous tutorials such as [**one-to-one mapping**](https://howtodoinjava.com/hibernate/hibernate-one-to-one-mapping-using-annotations/) and [**one-to-many**](https://howtodoinjava.com/hibernate/hibernate-one-to-many-mapping-using-annotations/)**mappings**. There we wanted to save the mapped entity whenever relationship owner entity got saved. To enable this we had use “**CascadeType**” attribute. In this tutorial, we will learn about various type of available options for cascading via **CascadeType**.

Before moving forward, let’s look at how this cascade type attribute is defined in your code. Let’s have an example for more clear understanding. Take a scenario where an Employee can have multiple Accounts; but one account must be associated with only one employee. Let’s create entities with minimum information for sake of clarity.

**EmployeeEntity.java**

|  |
| --- |
| @Entity  @Table(name = "Employee")  public class EmployeeEntity implements Serializable  {      private static final long serialVersionUID = -1798070786993154676L;      @Id      @Column(name = "ID", unique = true, nullable = false)      private Integer           employeeId;      @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;        @OneToMany(cascade=CascadeType.ALL, fetch = FetchType.LAZY)      @JoinColumn(name="EMPLOYEE\_ID")      private Set<AccountEntity> accounts;        //Getters and Setters Ommited  } |

**AccountEntity.java**

|  |
| --- |
| @Entity  @Table(name = "Account")  public class AccountEntity implements Serializable  {      private static final long serialVersionUID = 1L;      @Id      @Column(name = "ID", unique = true, nullable = false)      @GeneratedValue(strategy = GenerationType.SEQUENCE)      private Integer           accountId;      @Column(name = "ACC\_NO", unique = false, nullable = false, length = 100)      private String            accountNumber;        @OneToOne (mappedBy="accounts",  fetch = FetchType.LAZY)      private EmployeeEntity employee;    } |

Look at the bold line in above source code for EmployeeEntity.java. It defines “cascade=CascadeType.ALL” and it essentially means that any change happened on EmployeeEntitymust cascade to AccountEntity as well. If you save an employee, then all associated accounts will also be saved into database. If you delete an Employee then all accounts associated with that Employee also be deleted. Simple enough.

But what if we only want to cascade only save operations but not delete operation. Then we need to clearly specify it using below code.

|  |
| --- |
| @OneToMany(cascade=CascadeType.PERSIST, fetch = FetchType.LAZY)  @JoinColumn(name="EMPLOYEE\_ID")  private Set<AccountEntity> accounts; |

Now only when save() or persist() methods are called using employee instance then only accounts will be persisted. If any other method is called on session, it’s effect will not affect/cascade to accounts.

**JPA Cascade Types**

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.

The cascade configuration option accepts an array of CascadeTypes; thus, to include only refreshes and merges in the cascade operation for a One-to-Many relationship as in our example, you might see the following:

|  |
| --- |
| @OneToMany(cascade={CascadeType.REFRESH, CascadeType.MERGE}, fetch = FetchType.LAZY)  @JoinColumn(name="EMPLOYEE\_ID")  private Set<AccountEntity> accounts; |

Above cascading will cause accounts collection to be only merged and refreshed.

**Hibernate Specific Cascade Types**

There’s one more cascading operation that’s not part of the normal set above discussed, called “**orphan removal**“, which removes an owned object from the database when it’s removed from its owning relationship.

Let’s understand with an example. In our Employee and Account entity example, I have updated them as below and have mentioned “**orphanRemoval = true**” on accounts. It essentially means that whenever I will remove an ‘account from accounts set’ (which means I am removing the relationship between that account and Employee); the account entity which is not associated with any other Employee on database (i.e. orphan) should also be deleted.

**EmployeeEntity.java**

|  |
| --- |
| @Entity  @Table(name = "Employee")  public class EmployeeEntity implements Serializable  {      private static final long serialVersionUID = -1798070786993154676L;      @Id      @Column(name = "ID", unique = true, nullable = false)      private Integer           employeeId;      @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;        @OneToMany(orphanRemoval = true, mappedBy = "employee")      private Set<AccountEntity> accounts;    } |

**AccountEntity.java**

|  |
| --- |
| @Entity (name = "Account")  @Table(name = "Account")  public class AccountEntity implements Serializable  {      private static final long serialVersionUID = 1L;      @Id      @Column(name = "ID", unique = true, nullable = false)      @GeneratedValue(strategy = GenerationType.SEQUENCE)      private Integer           accountId;      @Column(name = "ACC\_NO", unique = false, nullable = false, length = 100)      private String            accountNumber;        @ManyToOne      private EmployeeEntity employee;  } |

**TestOrphanRemovalCascade.java**

|  |
| --- |
| public class TestOrphanRemovalCascade  {     public static void main(String[] args)     {        setupTestData();          Session sessionOne = HibernateUtil.getSessionFactory().openSession();        org.hibernate.Transaction tx = sessionOne.beginTransaction();          //Load the employee in another session        EmployeeEntity employee = (EmployeeEntity) sessionOne.load(EmployeeEntity.class, 1);        //Verify there are 3 accounts        System.out.println("Step 1 : " + employee.getAccounts().size());          //Remove an account from first position of collection        employee.getAccounts().remove(employee.getAccounts().iterator().next());          //Verify there are 2 accounts in collection        System.out.println("Step 2 : " + employee.getAccounts().size());          tx.commit();        sessionOne.close();          //In another session check the actual data in database        Session sessionTwo = HibernateUtil.getSessionFactory().openSession();        sessionTwo.beginTransaction();          EmployeeEntity employee1 = (EmployeeEntity) sessionTwo.load(EmployeeEntity.class, 1);        //Verify there are 2 accounts now associated with Employee        System.out.println("Step 3 : " + employee1.getAccounts().size());          //Verify there are 2 accounts in Account table        Query query = sessionTwo.createQuery("from Account a");        @SuppressWarnings("unchecked")        List<AccountEntity> accounts = query.list();        System.out.println("Step 4 : " + accounts.size());          sessionTwo.close();          HibernateUtil.shutdown();     }       private static void setupTestData(){        Session session = HibernateUtil.getSessionFactory().openSession();        session.beginTransaction();          //Create Employee        EmployeeEntity emp = new EmployeeEntity();        emp.setEmployeeId(1);        emp.setFirstName("Lokesh");        emp.setLastName("Gupta");        session.save(emp);          //Create Account 1        AccountEntity acc1 = new AccountEntity();        acc1.setAccountId(1);        acc1.setAccountNumber("11111111");        acc1.setEmployee(emp);        session.save(acc1);          //Create Account 2        AccountEntity acc2 = new AccountEntity();        acc2.setAccountId(2);        acc2.setAccountNumber("2222222");        acc2.setEmployee(emp);        session.save(acc2);          //Create Account 3        AccountEntity acc3 = new AccountEntity();        acc3.setAccountId(3);        acc3.setAccountNumber("33333333");        acc3.setEmployee(emp);        session.save(acc3);          session.getTransaction().commit();        session.close();     }  }    Output:    Hibernate: insert into Employee (FIRST\_NAME, LAST\_NAME, ID) values (?, ?, ?)  Hibernate: insert into Account (ACC\_NO, employee\_ID, ID) values (?, ?, ?)  Hibernate: insert into Account (ACC\_NO, employee\_ID, ID) values (?, ?, ?)  Hibernate: insert into Account (ACC\_NO, employee\_ID, ID) values (?, ?, ?)  Hibernate: select employeeen0\_.ID as ID1\_1\_0\_, employeeen0\_.FIRST\_NAME as FIRST\_NA2\_1\_0\_, employeeen0\_.LAST\_NAME as  LAST\_NAM3\_1\_0\_ from Employee employeeen0\_ where employeeen0\_.ID=?  Hibernate: select accounts0\_.employee\_ID as employee3\_1\_0\_, accounts0\_.ID as ID1\_0\_0\_, accounts0\_.ID as ID1\_0\_1\_,  accounts0\_.ACC\_NO as ACC\_NO2\_0\_1\_, accounts0\_.employee\_ID as employee3\_0\_1\_ from Account accounts0\_ where accounts0\_.employee\_ID=?  Step 1 : 3  Step 2 : 2  Hibernate: delete from Account where ID=?  Hibernate: select employeeen0\_.ID as ID1\_1\_0\_, employeeen0\_.FIRST\_NAME as FIRST\_NA2\_1\_0\_, employeeen0\_.LAST\_NAME as  LAST\_NAM3\_1\_0\_ from Employee employeeen0\_ where employeeen0\_.ID=?  Hibernate: select accounts0\_.employee\_ID as employee3\_1\_0\_, accounts0\_.ID as ID1\_0\_0\_, accounts0\_.ID as ID1\_0\_1\_,  accounts0\_.ACC\_NO as ACC\_NO2\_0\_1\_, accounts0\_.employee\_ID as employee3\_0\_1\_ from Account accounts0\_ where accounts0\_.employee\_ID=?  Step 3 : 2  Hibernate: select accountent0\_.ID as ID1\_0\_, accountent0\_.ACC\_NO as ACC\_NO2\_0\_, accountent0\_.employee\_ID as employee3\_0\_  from Account accountent0\_  Step 4 : 2 |

It’s a very good way of removing the matching/mismatching items from a collection (i.e. many-to-one or one-to-many relationships). You just remove the item from collection and hibernate take care of rest of the things for you. It will check whether entity is referenced from any place or not; If it is not then it will delete the entity from database itself.

**Hibernate – Lazy Loading Tutorial**

September 26, 2014 by Lokesh Gupta

So far in previous tutorials, we have learned the concepts around [entities persistence life cycle states](https://howtodoinjava.com/hibernate/hibernate-entity-persistence-lifecycle-states/), Some CRUD operations such as [Save a hibernate entity](https://howtodoinjava.com/hibernate/save-and-saveorupdate-for-saving-hibernate-entities/), [merge or refresh an entity](https://howtodoinjava.com/hibernate/merging-and-refreshing-hibernate-entities/) and then we learned about [cascading effect](https://howtodoinjava.com/hibernate/hibernate-jpa-cascade-types/) as well. Moving forward in this this tutorial, I will be discussing a must-known feature in hibernate, specially if you working in a very large application, known as **lazy loading**.

**When Lazy Loading is Needed : Sample Problem**

Consider one of common Internet web application: the online store. The store maintains a catalog of products. At the crudest level, this can be modeled as a catalog entity managing a series of product entities. In a large store, there may be tens of thousands of products grouped into various overlapping categories.

When a customer visits the store, the catalog must be loaded from the database. We probably don’t want the implementation to load every single one of the entities representing the tens of thousands of products to be loaded into memory. For a sufficiently large retailer, this might not even be possible, given the amount of physical memory available on the machine. Even if this were possible, it would probably cripple the performance of the site. Instead, we want only the catalog to load, possibly with the categories as well. Only when the user drills down into the categories should a subset of the products in that category be loaded from the database.

To manage this problem, Hibernate provides a facility called **lazy loading**. When enabled, an entity’s associated entities will be loaded only when they are directly requested.

**How Lazy Loading Solve the Problem**

Now when we have understood the problem, let’s understand how lazy loading actually helps in real life. If we consider to solve the problem discussed above then we would be accessing a category (or catalog) in below manner:

|  |
| --- |
| //Following code loads only a single category from the database:  Category category = (Category)session.get(Category.class,new Integer(42)); |

However, **if all products of this category are accessed, and lazy loading is in effect, the products are pulled from the database as needed**. For instance, in the following snippet, the associated product objects will be loaded since it is explicitly referenced in second line.

|  |
| --- |
| //Following code loads only a single category from the database  Category category = (Category)session.get(Category.class,new Integer(42));  //This code will fetch all products for category 42 from database 'NOW'  Set<Product> products = category.getProducts(); |

This solve our problem of loading the products only when they are needed.

**How to Enable Lazy Loading in Hibernate**

Before moving further, it is important to recap the default behavior of lazy loading in case of using hibernate mappings vs annotations.

The default behavior is to load ‘property values eagerly’ and to load ‘collections lazily’. Contrary to what you might remember if you have used plain Hibernate 2 (mapping files) before, where all references (including collections) are loaded eagerly by default.

Also note that @OneToMany and @ManyToMany associations are defaulted to LAZY loading; and @OneToOne and @ManyToOne are defaulted to EAGER loading. This is important to remember to avoid any pitfall in future.

To enable lazy loading explicitly you must use "fetch = FetchType.LAZY" on a association which you want to lazy load when you are using hibernate annotations.

An example usage will look like this:

|  |
| --- |
| @OneToMany( mappedBy = "category", fetch = FetchType.LAZY )  private Set<ProductEntity> products; |

Another attribute parallel to "FetchType.LAZY" is "FetchType.EAGER" which is just opposite to LAZY i.e. it will load association entity as well when owner entity is fetched first time.

**How Lazy Loading Works in Hibernate**

The simplest way that Hibernate can apply lazy load behavior upon your entities and associations is **by providing a proxy implementation** of them. Hibernate intercepts calls to the entity by substituting a proxy for it derived from the entity’s class. Where the requested information is missing, it will be loaded from the database before control is ceded to the parent entity’s implementation.

Please note that when the association is represented as a collection class, then a wrapper (essentially a proxy for the collection, rather than for the entities that it contains) is created and substituted for the original collection. When you access this collection proxy then what you get inside returned proxy collection are not proxy entities; rather they are actual entities. You need not to put much pressure on understanding this concept because on runtime it hardly matters.

**Effect of Lazy Loading on Detached Entities**

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**.

The cure is to ensure either that the entity is made persistent again by attaching it to a session or that all of the fields that will be required are accessed (so they are loaded into entity) before the entity is detached from the session.

That’s all for this simple, yet very important concept in Hibernate. Please drop a comment if something is not clear OR you want to discuss anything

**Hibernate Criteria Queries Tutorial and Examples**

October 28, 2014 by Lokesh Gupta

Hibernate provides three different ways to retrieve data from database. We have already discussed [**HQL and native SQL queries**](https://howtodoinjava.com/hibernate/complete-hibernate-query-language-hql-tutorial/). Now we will discuss our third option i.e. Criteria. The **Criteria Query API**lets you build nested, structured query expressions in Java, providing a compile-time syntax checking that is not possible with a query language like HQL or SQL. The Criteria API also includes **query by example (QBE)** functionality. This lets you supply example objects that contain the properties you would like to retrieve instead of having to step-by-step spell out the components of the query. It also includes projection and aggregation methods, including count(). Let’s explore it’s different features in detail.

**Table of Contents**

[Basic Usage Example](https://howtodoinjava.com/hibernate/hibernate-criteria-queries-tutorial-and-examples/#basic_usage)

[Using Restrictions with Criteria](https://howtodoinjava.com/hibernate/hibernate-criteria-queries-tutorial-and-examples/#restrictions)

[Paging Through the Result Set](https://howtodoinjava.com/hibernate/hibernate-criteria-queries-tutorial-and-examples/#paging)

[Obtaining a Unique Result](https://howtodoinjava.com/hibernate/hibernate-criteria-queries-tutorial-and-examples/#unique_result)

[Obtaining Distinct Results](https://howtodoinjava.com/hibernate/hibernate-criteria-queries-tutorial-and-examples/#distinct)

[Sorting the Query’s Results](https://howtodoinjava.com/hibernate/hibernate-criteria-queries-tutorial-and-examples/#sorting)

[Performing Associations (Joins)](https://howtodoinjava.com/hibernate/hibernate-criteria-queries-tutorial-and-examples/#associations)

[Adding Projections](https://howtodoinjava.com/hibernate/hibernate-criteria-queries-tutorial-and-examples/#projections_aggregates)

[Query By Example (QBE)](https://howtodoinjava.com/hibernate/hibernate-criteria-queries-tutorial-and-examples/#query_by_example)

[Summary](https://howtodoinjava.com/hibernate/hibernate-criteria-queries-tutorial-and-examples/#summary)

**Basic Usage Example**

The Criteria API allows you to build up a criteria query object programmatically; the org.hibernate.Criteria interface defines the available methods for one of these objects. The Hibernate Session interface contains several createCriteria() methods. Pass the persistent object’s class or its entity name to the createCriteria() method, and Hibernate will create a Criteria object that returns instances of the persistence object’s class when your application executes a criteria query.

The simplest example of a criteria query is one with no optional parameters or restrictions—the criteria query will simply return every object that corresponds to the class.

|  |
| --- |
| Criteria crit = session.createCriteria(Product.class);  List<Product> results = crit.list(); |

Moving on from this simple example, we will add constraints to our criteria queries so we can whittle down the result set.

**Using Restrictions with Criteria**

The Criteria API makes it easy to use restrictions in your queries to selectively retrieve objects; for instance, your application could retrieve only products with a price over $30. You may add these restrictions to a Criteria object with the add() method. The add() method takes an org.hibernate.criterion.Criterion object that represents an individual restriction. You can have more than one restriction for a criteria query.

**i) Restrictions.eq() Example**

To retrieve objects that have a property value that “**equals**” your restriction, use the eq() method on Restrictions, as follows:

|  |
| --- |
| Criteria crit = session.createCriteria(Product.class);  crit.add(Restrictions.eq("description","Mouse"));  List<Product> results = crit.list() |

Above query will search all products having description as “Mouse”.

**ii) Restrictions.ne() Example**

To retrieve objects that have a property value “not equal to” your restriction, use the ne() method on Restrictions, as follows:

|  |
| --- |
| Criteria crit = session.createCriteria(Product.class);  crit.add(Restrictions.ne("description","Mouse"));  List<Product> results = crit.list() |

Above query will search all products having description anything but not “Mouse”.

You cannot use the not-equal restriction to retrieve records with a NULL value in the database for that property (in SQL, and therefore in Hibernate, NULL represents the absence of data, and so cannot be compared with data). If you need to retrieve objects with NULL properties, you will have to use the isNull() restriction.

**iii) Restrictions.like() and Restrictions.ilike() Example**

Instead of searching for exact matches, we can retrieve all objects that have a property matching part of a given pattern. To do this, we need to create an SQL LIKE clause, with either the like() or the ilike() method. The ilike() method is case-insensitive.

|  |
| --- |
| Criteria crit = session.createCriteria(Product.class);  crit.add(Restrictions.like("name","Mou%",MatchMode.ANYWHERE));  List<Product> results = crit.list(); |

Above example uses an org.hibernate.criterion.MatchMode object to specify how to match the specified value to the stored data. The MatchMode object (a type-safe enumeration) has four different matches:

ANYWHERE: Anyplace in the string  
END: The end of the string  
EXACT: An exact match  
START: The beginning of the string

**iv) Restrictions.isNull() and Restrictions.isNotNull() Example**

The isNull() and isNotNull() restrictions allow you to do a search for objects that have (or do not have) null property values.

|  |
| --- |
| Criteria crit = session.createCriteria(Product.class);  crit.add(Restrictions.isNull("name"));  List<Product> results = crit.list(); |

**v) Restrictions.gt(), Restrictions.ge(), Restrictions.lt() and Restrictions.le() Examples**

Several of the restrictions are useful for doing math comparisons. The greater-than comparison is gt(), the greater-than-or-equal-to comparison is ge(), the less-than comparison is lt(), and the less-than-or-equal-to comparison is le(). We can do a quick retrieval of all products with prices over $25 like this, relying on Java’s type promotions to handle the conversion to Double:

|  |
| --- |
| Criteria crit = session.createCriteria(Product.class);  crit.add(Restrictions.gt("price", 25.0));  List<Product> results = crit.list(); |

**vi) Combining Two or More Criteria Examples**

Moving on, we can start to do more complicated queries with the Criteria API. For example, we can combine AND and OR restrictions in logical expressions. When we add more than one constraint to a criteria query, it is interpreted as an AND, like so:

|  |
| --- |
| Criteria crit = session.createCriteria(Product.class);  crit.add(Restrictions.lt("price",10.0));  crit.add(Restrictions.ilike("description","mouse", MatchMode.ANYWHERE));  List<Product> results = crit.list(); |

If we want to have two restrictions that return objects that satisfy either or both of the restrictions, we need to use the or() method on the Restrictions class, as follows:

|  |
| --- |
| Criteria crit = session.createCriteria(Product.class);  Criterion priceLessThan = Restrictions.lt("price", 10.0);  Criterion mouse = Restrictions.ilike("description", "mouse", MatchMode.ANYWHERE);  LogicalExpression orExp = Restrictions.or(priceLessThan, mouse);  crit.add(orExp);  List results=crit.list(); |

The orExp logical expression that we have created here will be treated like any other criterion. We can therefore add another restriction to the criteria:

|  |
| --- |
| Criteria crit = session.createCriteria(Product.class);  Criterion price = Restrictions.gt("price",new Double(25.0));  Criterion name = Restrictions.like("name","Mou%");  LogicalExpression orExp = Restrictions.or(price,name);  crit.add(orExp);  crit.add(Restrictions.ilike("description","blocks%"));  List results = crit.list(); |

**vii) Using Disjunction Objects with Criteria**

If we wanted to create an OR expression with more than two different criteria (for example, “price > 25.0 OR name like Mou% OR description not like blocks%”), we would use an org.hibernate.criterion.Disjunction object to represent a disjunction.

You can obtain this object from the disjunction() factory method on the Restrictions class. The disjunction is more convenient than building a tree of OR expressions in code. To represent an AND expression with more than two criteria, you can use the conjunction() method, although you can easily just add those to the Criteria object. The conjunction can be more convenient than building a tree of AND expressions in code. Here is an example that uses the disjunction:

|  |
| --- |
| Criteria crit = session.createCriteria(Product.class);  Criterion priceLessThan = Restrictions.lt("price", 10.0);  Criterion mouse = Restrictions.ilike("description", "mouse", MatchMode.ANYWHERE);  Criterion browser = Restrictions.ilike("description", "browser", MatchMode.ANYWHERE);  Disjunction disjunction = Restrictions.disjunction();  disjunction.add(priceLessThan);  disjunction.add(mouse);  disjunction.add(browser);  crit.add(disjunction);  List results = crit.list(); |

**viii) Restrictions.sqlRestriction() Example**

sqlRestriction() restriction allows you to directly specify SQL in the Criteria API. It’s useful if you need to use SQL clauses that Hibernate does not support through the Criteria API.

Your application’s code does not need to know the name of the table your class uses. Use {alias} to signify the class’s table, as follows:

|  |
| --- |
| Criteria crit = session.createCriteria(Product.class);  crit.add(Restrictions.sqlRestriction("{alias}.description like 'Mou%'"));  List<Product> results = crit.list(); |

**Paging Through the ResultSet**

One common application pattern that criteria can address is pagination through the result set of a database query. There are two methods on the Criteria interface for paging, just as there are for Query: setFirstResult() and setMaxResults(). The setFirstResult() method takes an integer that represents the first row in your result set, starting with row 0. You can tell Hibernate to retrieve a fixed number of objects with the setMaxResults() method. Using both of these together, we can construct a paging component in our web or Swing application.

|  |
| --- |
| Criteria crit = session.createCriteria(Product.class);  crit.setFirstResult(1);  crit.setMaxResults(20);  List<Product> results = crit.list(); |

As you can see, this makes paging through the result set easy. You can increase the first result you return (for example, from 1, to 21, to 41, etc.) to page through the result set.

**Obtaining a Unique Result**

Sometimes you know you are going to return only zero or one object from a given query. This could be because you are calculating an aggregate or because your restrictions naturally lead to a unique result. If you want obtain a single Object reference instead of a List, the uniqueResult() method on the Criteria object returns an object or null. If there is more than one result, the uniqueResult() method throws a HibernateException.

The following short example demonstrates having a result set that would have included more than one result, except that it was limited with the setMaxResults() method:

|  |
| --- |
| Criteria crit = session.createCriteria(Product.class);  Criterion price = Restrictions.gt("price",new Double(25.0));  crit.setMaxResults(1);  Product product = (Product) crit.uniqueResult(); |

Again, please note that you need to make sure that your query returns only one or zero results if you use the uniqueResult() method. Otherwise, Hibernate will throw a NonUniqueResultExceptionexception.

**Obtaining Distinct Results**

If you would like to work with distinct results from a criteria query, Hibernate provides a result transformer for distinct entities, org.hibernate.transform.DistinctRootEntityResultTransformer, which ensures that no duplicates will be in your query’s result set. **Rather than using SELECT DISTINCT with SQL, the distinct result transformer compares each of your results using their default hashCode() methods, and only adds those results with unique hash codes to your result set**. This may or may not be the result you would expect from an otherwise equivalent SQL DISTINCT query, so **be careful with this**.

|  |
| --- |
| Criteria crit = session.createCriteria(Product.class);  Criterion price = Restrictions.gt("price",new Double(25.0));  crit.setResultTransformer( DistinctRootEntityResultTransformer.INSTANCE )  List<Product> results = crit.list(); |

An additional performance note: the comparison is done in Hibernate’s Java code, not at the database, so non-unique results will still be transported across the network.

**Sorting the Query’s Results**

Sorting the query’s results works much the same way with criteria as it would with HQL or SQL. The Criteria API provides the org.hibernate.criterion.Order class to sort your result set in either ascending or descending order, according to one of your object’s properties.

This example demonstrates how you would use the Order class:

|  |
| --- |
| Criteria crit = session.createCriteria(Product.class);  crit.add(Restrictions.gt("price",10.0));  crit.addOrder(Order.desc("price"));  List<Product> results = crit.list(); |

You may add more than one Order object to the Criteria object. Hibernate will pass them through to the underlying SQL query. Your results will be sorted by the first order, then any identical matches within the first sort will be sorted by the second order, and so on. Beneath the covers, **Hibernate passes this on to an SQL ORDER BY clause after substituting the proper database column name for the property**.

**Performing Associations (Joins)**

The association works when going from **either one-to-many or from many-to-one**. First, we will demonstrate how to use one-to-many associations to obtain suppliers who sell products with a price over $25. Notice that we create a new Criteria object for the products property, add restrictions to the products’ criteria we just created, and then obtain the results from the supplier Criteria object:

|  |
| --- |
| Criteria crit = session.createCriteria(Supplier.class);  Criteria prdCrit = crit.createCriteria("products");  prdCrit.add(Restrictions.gt("price",25.0));  List results = crit.list(); |

Going the other way, we obtain all the products from the supplier MegaInc using many-to-one associations:

|  |
| --- |
| Criteria crit = session.createCriteria(Product.class);  Criteria suppCrit = crit.createCriteria("supplier");  suppCrit.add(Restrictions.eq("name","Hardware Are We"));  List results = crit.list(); |

**Adding Projections and Aggregates**

Instead of working with objects from the result set, you can treat the results from the result set as a set of rows and columns, also known as a projection of the data. This is similar to how you would use data from a SELECT query with JDBC.

To use projections, start by getting the org.hibernate.criterion.Projection object you need from the org.hibernate.criterion.Projections factory class. The Projections class is similar to the Restrictions class in that it provides several static factory methods for obtaining Projection instances. After you get a Projection object, add it to your Criteria object with the setProjection() method. When the Criteria object executes, the list contains object references that you can cast to the appropriate type.

**Example 1 : Single Aggregate ( Getting Row Count )**

|  |
| --- |
| Criteria crit = session.createCriteria(Product.class);  crit.setProjection(Projections.rowCount());  List<Long> results = crit.list(); |

Other aggregate functions available through the Projections factory class include the following:

1. **avg(String propertyName)**: Gives the average of a property’s value
2. **count(String propertyName)**: Counts the number of times a property occurs
3. **countDistinct(String propertyName)**: Counts the number of unique values the property contains
4. **max(String propertyName)**: Calculates the maximum value of the property values
5. **min(String propertyName)**: Calculates the minimum value of the property values
6. **sum(String propertyName)**: Calculates the sum total of the property values

**Example 2 : Multiple Aggregates**

We can apply more than one projection to a given Criteria object. To add multiple projections, get a projection list from the projectionList() method on the Projections class. The org.hibernate.criterion.ProjectionList object has an add() method that takes a Projection object. You can pass the projections list to the setProjection() method on the Criteria object because ProjectionList implements the Projection interface.

|  |
| --- |
| Criteria crit = session.createCriteria(Product.class);  ProjectionList projList = Projections.projectionList();  projList.add(Projections.max("price"));  projList.add(Projections.min("price"));  projList.add(Projections.avg("price"));  projList.add(Projections.countDistinct("description"));  crit.setProjection(projList);  List<object[]> results = crit.list(); |

**Example 3 : Getting Selected Columns**

Another use of projections is to retrieve individual properties, rather than entities. For instance, we can retrieve just the name and description from our product table, instead of loading the entire object representation into memory.

|  |
| --- |
| Criteria crit = session.createCriteria(Product.class);  ProjectionList projList = Projections.projectionList();  projList.add(Projections.property("name"));  projList.add(Projections.property("description"));  crit.setProjection(projList);  crit.addOrder(Order.asc("price"));  List<object[]> results = crit.list(); |

**Query By Example (QBE)**

In QBE, instead of programmatically building a Criteria object with Criterion objects and logical expressions, you can partially populate an instance of the object. You use this instance as a template and have Hibernate build the criteria for you based upon its values. This keeps your code clean and makes your project easier to test.

For instance, if we have a user database, we can construct an instance of a user object, set the property values for type and creation date, and then use the Criteria API to run a QBE query. Hibernate will return a result set containing all user objects that match the property values that were set. Behind the scenes, Hibernate inspects the Example object and constructs an SQL fragment that corresponds to the properties on the Example object.

The following basic example searches for suppliers that match the name on the example Supplier object:

|  |
| --- |
| Criteria crit = session.createCriteria(Supplier.class);  Supplier supplier = new Supplier();  supplier.setName("MegaInc");  crit.add(Example.create(supplier));  List results = crit.list(); |

**Summary**

Using the Criteria API is an excellent way to get started developing with HQL. The developers of Hibernate have provided a clean API for adding restrictions to queries with Java objects. Although HQL isn’t too difficult to learn, some developers prefer the Criteria Query API, as it offers compile-time syntax checking—although column names and other schema-dependent information cannot be checked until run time

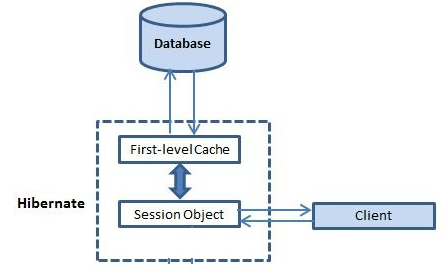
**Understanding Hibernate First Level Cache with Example**

July 1, 2013 by Lokesh Gupta

Caching is a 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 achieves the second goal by implementing first level cache.

**Fist level cache** in hibernate is enabled by default and you do not need to do anything to get this functionality working. In fact, you can not disable it even forcefully.

Its easy to understand the first level cache if we understand the fact that**it is associated with Session object**. As we know session object is created on demand from session factory and**it is lost, once the session is closed**. Similarly, first level cache associated with session object is available only till session object is live. It is available to session object only and is **not accessible to any other session object** in any other part of application.



**Important facts**

1. First level cache is associated with “session” object and other session objects in application can not see it.
2. The scope of cache objects is of session. Once session is closed, cached objects are gone forever.
3. First level cache is enabled by default and you can not disable it.
4. When we query an entity first time, it is retrieved from database and stored in first level cache associated with hibernate session.
5. If we query same object again with same session object, it will be loaded from cache and no sql query will be executed.
6. The loaded entity can be removed from session using evict() method. The next loading of this entity will again make a database call if it has been removed using evict() method.
7. The whole session cache can be removed using clear() method. It will remove all the entities stored in cache.

Lets verify above facts using examples.

**First level cache retrieval example**

In this example, I am retrieving DepartmentEntity object from database using hibernate session. I will retrieve it multiple times, and will observe the sql logs to see the differences.

|  |
| --- |
| //Open the hibernate session  Session session = HibernateUtil.getSessionFactory().openSession();  session.beginTransaction();    //fetch the department entity from database first time  DepartmentEntity department = (DepartmentEntity) session.load(DepartmentEntity.class, newInteger(1));  System.out.println(department.getName());    //fetch the department entity again  department = (DepartmentEntity) session.load(DepartmentEntity.class, new Integer(1));  System.out.println(department.getName());    session.getTransaction().commit();  HibernateUtil.shutdown();    Output:    Hibernate: select department0\_.ID as ID0\_0\_, department0\_.NAME as NAME0\_0\_ from DEPARTMENT department0\_ where department0\_.ID=?  Human Resource  Human Resource |

As you can see that **second “session.load()” statement does not execute select query again and load the department entity directly**.

**First level cache retrieval example with new session**

**With new session, entity is fetched from database again irrespective of it is already present in any other session in application.**

|  |
| --- |
| //Open the hibernate session  Session session = HibernateUtil.getSessionFactory().openSession();  session.beginTransaction();    Session sessionTemp = HibernateUtil.getSessionFactory().openSession();  sessionTemp.beginTransaction();  try  {      //fetch the department entity from database first time      DepartmentEntity department = (DepartmentEntity) session.load(DepartmentEntity.class, newInteger(1));      System.out.println(department.getName());        //fetch the department entity again      department = (DepartmentEntity) session.load(DepartmentEntity.class, new Integer(1));      System.out.println(department.getName());        department = (DepartmentEntity) sessionTemp.load(DepartmentEntity.class, new Integer(1));      System.out.println(department.getName());  }  finally  {      session.getTransaction().commit();      HibernateUtil.shutdown();        sessionTemp.getTransaction().commit();      HibernateUtil.shutdown();  }    Output:    Hibernate: select department0\_.ID as ID0\_0\_, department0\_.NAME as NAME0\_0\_ from DEPARTMENT department0\_ where department0\_.ID=?  Human Resource  Human Resource    Hibernate: select department0\_.ID as ID0\_0\_, department0\_.NAME as NAME0\_0\_ from DEPARTMENT department0\_ where department0\_.ID=?  Human Resource |

You can see that even if the department entity was stored in “session” object, still another database query was executed when we use another session object “sessionTemp”.

**Removing cache objects from first level cache example**

Though we can not disable the first level cache in hibernate, but we can certainly remove some of objects from it when needed. This is done using two methods :

* evict()
* clear()

Here **evict() is used to remove a particular object from cache associated with session**, and **clear() method is used to remove all cached objects associated with session**. So they are essentially like remove one and remove all.

|  |
| --- |
| //Open the hibernate session  Session session = HibernateUtil.getSessionFactory().openSession();  session.beginTransaction();  try  {      //fetch the department entity from database first time      DepartmentEntity department = (DepartmentEntity) session.load(DepartmentEntity.class, newInteger(1));      System.out.println(department.getName());        //fetch the department entity again      department = (DepartmentEntity) session.load(DepartmentEntity.class, new Integer(1));      System.out.println(department.getName());        session.evict(department);      //session.clear();        department = (DepartmentEntity) session.load(DepartmentEntity.class, new Integer(1));      System.out.println(department.getName());  }  finally  {      session.getTransaction().commit();      HibernateUtil.shutdown();  }    Output:    Hibernate: select department0\_.ID as ID0\_0\_, department0\_.NAME as NAME0\_0\_ from DEPARTMENT department0\_ where department0\_.ID=?  Human Resource  Human Resource    Hibernate: select department0\_.ID as ID0\_0\_, department0\_.NAME as NAME0\_0\_ from DEPARTMENT department0\_ where department0\_.ID=?  Human Resource |

Clearly, evict() method removed the department object from cache so that it was fetched again from database.

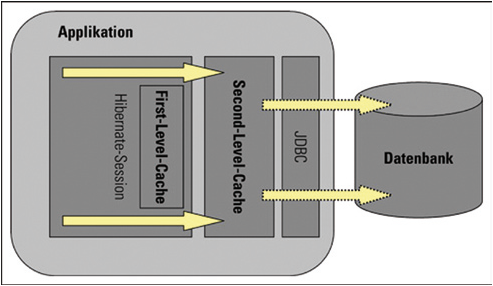
**How Hibernate Second Level Cache Works?**

July 2, 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 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.

Above statement means, **second level cache is created in session factory scope** and is **available to be used in all sessions** which are created using that particular session factory.  
It also means that **once session factory is closed, all cache associated with it die** and cache manager also closed down.  
Further, It also means that if you have two instances of session factory (normally no application does that), you will have two cache managers in your application and while accessing cache stored in physical store, you might get unpredictable results like cache-miss.

Hibernate first and second level cache

In this tutorial, I am giving concepts around hibernate second level cache and give example using code snippets.

**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.

|  |
| --- |
| /\*\*   \* Evicts all second level cache hibernate entites. This is generally only   \* needed when an external application modifies the databaase.   \*/  public void evict2ndLevelCache() {      try {          Map<String, ClassMetadata> classesMetadata = sessionFactory.getAllClassMetadata();          for (String entityName : classesMetadata.keySet()) {              logger.info("Evicting Entity from 2nd level cache: " + entityName);              sessionFactory.evictEntity(entityName);          }      } catch (Exception e) {          logger.logp(Level.SEVERE, "SessionController", "evict2ndLevelCache", "Error evicting 2nd level hibernate cache entities: ", e);      }  } |

To understand more using examples, I wrote an application for testing in which I configured EhCache as 2nd level cache. Lets see various scenarios:

**a) Entity is fetched very first time**

|  |
| --- |
| DepartmentEntity department = (DepartmentEntity) session.load(DepartmentEntity.class, newInteger(1));  System.out.println(department.getName());    System.out.println(HibernateUtil.getSessionFactory().getStatistics().getEntityFetchCount());           //Prints 1  System.out.println(HibernateUtil.getSessionFactory().getStatistics().getSecondLevelCacheHitCount());   //Prints 0    Output: 1 0 |

***Explanation***: Entity is not present in either 1st or 2nd level cache so, it is fetched from database.

**b) Entity is fetched second time**

|  |
| --- |
| //Entity is fecthed very first time  DepartmentEntity department = (DepartmentEntity) session.load(DepartmentEntity.class, newInteger(1));  System.out.println(department.getName());    //fetch the department entity again  department = (DepartmentEntity) session.load(DepartmentEntity.class, new Integer(1));  System.out.println(department.getName());    System.out.println(HibernateUtil.getSessionFactory().getStatistics().getEntityFetchCount());           //Prints 1  System.out.println(HibernateUtil.getSessionFactory().getStatistics().getSecondLevelCacheHitCount());   //Prints 0    Output: 1 0 |

***Explanation***: Entity is present in first level cache so, it is fetched from there. No need to go to second level cache.

**c) Entity is evicted from first level cache and fetched again**

|  |
| --- |
| //Entity is fecthed very first time  DepartmentEntity department = (DepartmentEntity) session.load(DepartmentEntity.class, newInteger(1));  System.out.println(department.getName());    //fetch the department entity again  department = (DepartmentEntity) session.load(DepartmentEntity.class, new Integer(1));  System.out.println(department.getName());    //Evict from first level cache  session.evict(department);    department = (DepartmentEntity) session.load(DepartmentEntity.class, new Integer(1));  System.out.println(department.getName());    System.out.println(HibernateUtil.getSessionFactory().getStatistics().getEntityFetchCount());           //Prints 1  System.out.println(HibernateUtil.getSessionFactory().getStatistics().getSecondLevelCacheHitCount());   //Prints 1    Output: 1 1 |

***Explanation***: First time entity is fetched from database. Which cause it store in 1st and 2nd level cache. Second load call fetched from first level cache. Then we evicted entity from 1st level cache. So third load() call goes to second level cache and getSecondLevelCacheHitCount() returns 1.

**d) Access second level cache from another session**

|  |
| --- |
| //Entity is fecthed very first time  DepartmentEntity department = (DepartmentEntity) session.load(DepartmentEntity.class, newInteger(1));  System.out.println(department.getName());    //fetch the department entity again  department = (DepartmentEntity) session.load(DepartmentEntity.class, new Integer(1));  System.out.println(department.getName());    //Evict from first level cache  session.evict(department);    department = (DepartmentEntity) session.load(DepartmentEntity.class, new Integer(1));  System.out.println(department.getName());    department = (DepartmentEntity) anotherSession.load(DepartmentEntity.class, new Integer(1));  System.out.println(department.getName());    System.out.println(HibernateUtil.getSessionFactory().getStatistics().getEntityFetchCount());           //Prints 1  System.out.println(HibernateUtil.getSessionFactory().getStatistics().getSecondLevelCacheHitCount());   //Prints 2    Output: 1 2 |

***Explanation***: When another session created from same session factory try to get entity, it is successfully looked up in second level cache and no database call is made.

**Hibernate / JPA 2 Persistence Annotations Tutorial**

October 21, 2014 by Lokesh Gupta

This [**Hibernate**](https://howtodoinjava.com/hibernate-tutorials/) (or JPA 2) Persistence Annotations Tutorial contains overview of all important annotations which you may need while annotating your java POJOs to make them act as persistent JPA entities. This tutorial first defines a POJO “*EmployeeEntity*“, define some attribute inside it and also has respective getter and setter methods. As we learn the new annotations, we will apply these annotations on this EmployeeEntity and then we will understand what that specific annotation means.

Let’s quickly list down the annotations, we are going to discuss in this tutorial.

**Table of Contents**

[**Most used JPA Annotations**](https://howtodoinjava.com/hibernate/hibernate-jpa-2-persistence-annotations-tutorial/#jpa_annotations)

[Entity Beans with @Entity](https://howtodoinjava.com/hibernate/hibernate-jpa-2-persistence-annotations-tutorial/#entity)

[Primary Keys with @Id and @GeneratedValue](https://howtodoinjava.com/hibernate/hibernate-jpa-2-persistence-annotations-tutorial/#id_generatedValue)

[Generating Primary Key Values with @SequenceGenerator](https://howtodoinjava.com/hibernate/hibernate-jpa-2-persistence-annotations-tutorial/#sequenceGenerator)

[Generating Primary Key Values with @TableGenerator](https://howtodoinjava.com/hibernate/hibernate-jpa-2-persistence-annotations-tutorial/#tableGenerator)

[Compound Primary Keys with @Id, @IdClass, or @EmbeddedId](https://howtodoinjava.com/hibernate/hibernate-jpa-2-persistence-annotations-tutorial/#embeddedId)

[Database Table Mapping with @Table and @SecondaryTable](https://howtodoinjava.com/hibernate/hibernate-jpa-2-persistence-annotations-tutorial/#table_secondaryTable)

[Persisting Basic Types with @Basic](https://howtodoinjava.com/hibernate/hibernate-jpa-2-persistence-annotations-tutorial/#basic)

[Omitting Persistence with @Transient](https://howtodoinjava.com/hibernate/hibernate-jpa-2-persistence-annotations-tutorial/#transient)

[Mapping Properties and Fields with @Column](https://howtodoinjava.com/hibernate/hibernate-jpa-2-persistence-annotations-tutorial/#column)

[**Modeling Entity Relationships**](https://howtodoinjava.com/hibernate/hibernate-jpa-2-persistence-annotations-tutorial/#mappings)

[**Mapping Inheritance Hierarchies**](https://howtodoinjava.com/hibernate/hibernate-jpa-2-persistence-annotations-tutorial/#inheritance)

[Single Table](https://howtodoinjava.com/hibernate/hibernate-jpa-2-persistence-annotations-tutorial/#single_table)

[Joined Table](https://howtodoinjava.com/hibernate/hibernate-jpa-2-persistence-annotations-tutorial/#joined_table)

[Table per Concrete Class](https://howtodoinjava.com/hibernate/hibernate-jpa-2-persistence-annotations-tutorial/#per_concrete_class)

[**Other JPA 2 Persistence Annotations**](https://howtodoinjava.com/hibernate/hibernate-jpa-2-persistence-annotations-tutorial/#other)

[Temporal Data with @Temporal](https://howtodoinjava.com/hibernate/hibernate-jpa-2-persistence-annotations-tutorial/#temporal)

[Element Collections with @ElementCollection](https://howtodoinjava.com/hibernate/hibernate-jpa-2-persistence-annotations-tutorial/#elementCollection)

[Large Objects with @Lob](https://howtodoinjava.com/hibernate/hibernate-jpa-2-persistence-annotations-tutorial/#lob)

[Mapped Superclasses with @MappedSuperclass](https://howtodoinjava.com/hibernate/hibernate-jpa-2-persistence-annotations-tutorial/#mappedSuperclass)

[Ordering Collections with @OrderColumn](https://howtodoinjava.com/hibernate/hibernate-jpa-2-persistence-annotations-tutorial/#orderColumn)

[**Named Queries (HQL or JPQL)**](https://howtodoinjava.com/hibernate/hibernate-jpa-2-persistence-annotations-tutorial/#named_queries)

[@NamedQuery and @NamedQueries](https://howtodoinjava.com/hibernate/hibernate-jpa-2-persistence-annotations-tutorial/#namedQuery)

[Named Native Queries using @NamedNativeQuery](https://howtodoinjava.com/hibernate/hibernate-jpa-2-persistence-annotations-tutorial/#namedNativeQuery)

Above list is a pretty big list and leaves less scope of fully detailed information of items listed, yet I will try to include all relevant information, which will help you in making decisions. Let’s get started by having written down our EmployeeEntity POJO first and then start decorating it with JPA annotationsin whole discussion.

|  |
| --- |
| package com.howtodoinjava.demo.test;    import java.io.Serializable;    public class EmployeeEntity implements Serializable  {     private static final long serialVersionUID = 570235488278506106L;       private Integer           employeeId;     private String            firstName;     private String            lastName;       public Integer getEmployeeId()     {        return employeeId;     }       public void setEmployeeId(Integer employeeId)     {        this.employeeId = employeeId;     }       public String getFirstName()     {        return firstName;     }       public void setFirstName(String firstName)     {        this.firstName = firstName;     }       public String getLastName()     {        return lastName;     }       public void setLastName(String lastName)     {        this.lastName = lastName;     }  } |

**Most used JPA Annotations**

**Hibernate / JPA 2 @Entity Annotation Tutorial**

This shall be the first step to mark the POJO as JPA entity. To do this, we need to apply @Entityannotation as follows:

|  |
| --- |
| import javax.persistence.Entity;    @Entity  public class EmployeeEntity implements Serializable  {      public EmployeeEntity(){        }      //Other code  } |

The JPA 2 standard annotations are contained in the ***javax.persistence package***, so we imported the annotation from this package. The @Entity annotation marks this class as an entity bean, so it **must have a no-argument constructor** that is visible with at least protected scope (JPA specific). Hibernate supports package scope as the minimum, but you lose portability to other JPA implementations because they might be allowing only protected level scope. Ideally, you should make this constructor public, which makes it highly compatible with other specifications as well. There are some more rules such as POJO class lust not be final; and it must not be abstract as well.

Now let’s have a quick overview of “@org.hibernate.annotations.Entity” annotation (hibernate specific) as well. Using it, you can define some extra behavior for entity which is hibernate specific.

@javax.persistence.Entity is still mandatory, @org.hibernate.annotations.Entity is a compliment, and not a replacement.

|  |
| --- |
| @javax.persistence.Entity  @org.hibernate.annotations.Entity(        selectBeforeUpdate = true,        dynamicInsert = true, dynamicUpdate = true,        optimisticLock = OptimisticLockType.ALL,        polymorphism = PolymorphismType.EXPLICIT,        mutable = false)  public class EmployeeEntity implements Serializable  {      public EmployeeEntity(){        }      //Other code  } |

All above attributes have been marked deprecated long ago, but still supported. I gave example only for information purpose.

Infact, with new version of hibernate, you need not to used @org.hibernate.annotations.Entity at all. Instead, you can directly add the desired behavior using annotations only. Let’s look equivalent code for entity written above.

|  |
| --- |
| import org.hibernate.annotations.DynamicInsert;  import org.hibernate.annotations.DynamicUpdate;  import org.hibernate.annotations.OptimisticLockType;  import org.hibernate.annotations.OptimisticLocking;  import org.hibernate.annotations.Polymorphism;  import org.hibernate.annotations.PolymorphismType;  import org.hibernate.annotations.SelectBeforeUpdate;    @javax.persistence.Entity  @SelectBeforeUpdate  @DynamicInsert  @DynamicUpdate  @Polymorphism (type = PolymorphismType.EXPLICIT)  @OptimisticLocking (type = OptimisticLockType.ALL)  public class EmployeeEntity implements Serializable  {      public EmployeeEntity(){        }      //Other code  } |

Moving on, use these annotations in your code if needed.

**Primary Keys with @Id and @GeneratedValue**

Each entity bean has to have a primary key, which you annotate on the class with the @Id annotation. Typically, the primary key will be a single field, though it can also be a composite of multiple fields which we will see in later sections.

The placement of the @Id annotation determines the default access strategy that Hibernate will use for the mapping. If the annotation is applied to a field as shown below, then “**field access**” will be used.

|  |
| --- |
| @Id  private Integer employeeId; |

If, instead, the annotation is applied to the accessor for the field then property access will be used.

|  |
| --- |
| @Id  public Integer getEmployeeId()  {    return employeeId;  } |

Property access means that Hibernate will call the mutator/setter instead of actually setting the field directly, what it does in case of field access. This gives flexibility to alter the value of actual value set in id field if needed. Additionally, you can apply extra logic on setting of ‘id’ field in mutator for other fields as well.

By default, the @Id annotation will not create a primary key generation strategy, which means that you, as the code’s author, need to determine what valid primary keys are, by setting them explicitly calling setter methods. OR you can use @GeneratedValue annotation.

@GeneratedValue annotation takes a pair of attributes: *strategy* and *generator* as below:

|  |
| --- |
| @Id  @GeneratedValue (strategy = GenerationType.SEQUENCE)  private Integer employeeId;    //OR a more complex use can be    @Id  @GeneratedValue(strategy=GenerationType.TABLE , generator="employee\_generator")  @TableGenerator(name="employee\_generator",                 table="pk\_table",                 pkColumnName="name",                 valueColumnName="value",                 allocationSize=100)  private Integer employeeId; |

The strategy attribute must be a value from the javax.persistence.GeneratorType enumeration. If you do not specify a generator type, the default is AUTO. There are four different types of primary key generators on GeneratorType, as follows:

1. **AUTO**: Hibernate decides which generator type to use, based on the database’s support for primary key generation.
2. **IDENTITY**: The database is responsible for determining and assigning the next primary key.
3. **SEQUENCE**: Some databases support a SEQUENCE column type. It uses @SequenceGenerator.
4. **TABLE**: This type keeps a separate table with the primary key values. It uses @TableGenerator.

The generator attribute allows the use of a custom generation mechanism shown in above code example.

**Generating Primary Key Values with @SequenceGenerator**

A sequence is a database object that can be used as a source of primary key values. It is similar to the use of an identity column type, except that a sequence is independent of any particular table and can therefore be used by multiple tables.

To declare the specific sequence object to use and its properties, you must include the @SequenceGenerator annotation on the annotated field. Here’s an example:

|  |
| --- |
| @Id  @SequenceGenerator(name="seq1",sequenceName="HIB\_SEQ")  @GeneratedValue(strategy=SEQUENCE,generator="seq1")  private Integer employeeId; |

Here, a sequence-generation annotation named seq1 has been declared. This refers to the database sequence object called HIB\_SEQ. The name seq1 is then referenced as the generator attribute of the @GeneratedValue annotation. Only the sequence generator name is mandatory; the other attributes will take sensible default values, but you should provide an explicit value for the sequenceName attribute as a matter of good practice anyway. If not specified, the sequenceName value to be used is selected by the persistence provider.

**Generating Primary Key Values with @TableGenerator**

The @TableGenerator annotation is used in a very similar way to the @SequenceGenerator annotation, but because @TableGenerator manipulates a standard database table to obtain its primary key values, instead of using a vendor-specific sequence object, it is guaranteed to be portable between database platforms.

For optimal portability and optimal performance, you should not specify the use of a table generator, but instead use the @GeneratorValue(strategy=GeneratorType.AUTO) configuration, which allows the persistence provider to select the most appropriate strategy for the database in use.

As with the sequence generator, the name attributes of @TableGenerator are mandatory and the other attributes are optional, with the table details being selected by the persistence provider. Let’s look at the example again.

|  |
| --- |
| @Id  @GeneratedValue(strategy=GenerationType.TABLE , generator="employee\_generator")  @TableGenerator(name="employee\_generator",                 table="pk\_table",                 pkColumnName="name",                 valueColumnName="value",                 allocationSize=100)  private Integer employeeId; |

The optional attributes are as follows:

* **allocationSize**: Allows the number of primary keys set aside at one time to be tuned for performance.
* **catalog**: Allows the catalog that the table resides within to be specified.
* **initialValue**: Allows the starting primary key value to be specified.
* **pkColumnName**: Allows the primary key column of the table to be identified. The table can contain the details necessary for generating primary key values for multiple entities.
* **pkColumnValue**: Allows the primary key for the row containing the primary key generation information to be identified.
* **schema**: Allows the schema that the table resides within to be specified.
* **table**: The name of the table containing the primary key values.
* **uniqueConstraints**: Allows additional constraints to be applied to the table for schema generation.
* **valueColumnName**: Allows the column containing the primary key generation information for the current entity to be identified.

Because the table can be used to contain the primary key values for a variety of entries, it is likely to have a single row for each of the entities using it. It therefore needs its own primary key (pkColumnName), as well as a column containing the next primary key value to be used (pkColumnValue) for any of the entities obtaining their primary keys from it.

**Compound Primary Keys with @Id, @IdClass, or @EmbeddedId**

While the use of single-column surrogate keys is advantageous for various reasons, you may sometimes be forced to work with business keys. When these are contained in a single column, you can use @Id without specifying a generation strategy forcing the user to assign a primary key value before the entity can be persisted.

But in case of multi-column primary key, You must create a class to represent this primary key. It will not require a primary key of its own, of course, but it must be a public class, must have a default constructor, must be serializable, and must [**implement hashCode() and equals() methods**](https://howtodoinjava.com/core-java/related-concepts/working-with-hashcode-and-equals-methods-in-java/) to allow the Hibernate code to test for primary key collisions.

Your three strategies for using this primary key class once it has been created are as follows:

1. Mark it as @Embeddable and add to your entity class a normal property for it, marked with @Id.
2. Add to your entity class a normal property for it, marked with @EmbeddableId.
3. Add properties to your entity class for all of its fields, mark them with @Id, and mark your entity class with @IdClass, supplying the class of your primary key class.

The use of @Id with a class marked as @Embeddable is the most natural approach. The @Embeddable tag can be used for non-primary key embeddable values anyway. It allows you to treat the compound primary key as a single property, and it permits the reuse of the @Embeddable class in other tables.

One thing worth pointing out: the embedded primary key classes must be serializable.

A detailed example can be read here : <http://docs.oracle.com/javaee/6/api/javax/persistence/Embeddable.html>

**Database Table Mapping with @Table and @SecondaryTable**

By default, table names are derived from the entity names. Therefore, given a class Employee with a simple @Entity annotation, the table name would be “employee”, adjusted for the database’s configuration. If the entity name is changed (by providing a different name in the @Entity annotation, such as **@Entity(“EMP\_MASTER”))**, the new name will be used for the table name.

The table name can be customized further, and other database-related attributes can be configured via the @Table annotation. This annotation allows you to specify many of the details of the table that will be used to persist the entity in the database. As already pointed out, if you omit the annotation, Hibernate will default to using the class name for the table name, so you need only provide this annotation if you want to override that behavior. The @Table annotation provides four attributes, allowing you to override the name of the table, its catalog, and its schema, and to enforce unique constraints on columns in the table. Typically, you would only provide a substitute table name thus: @Table(name="ORDER\_HISTORY"). The unique constraints will be applied if the database schema is generated from the annotated classes, and will supplement any column-specific constraints. They are not otherwise enforced.

The @SecondaryTable annotation provides a way to model an entity bean that is persisted across several different database tables. Here, in addition to providing an @Table annotation for the primary database table, your entity bean can have an @SecondaryTable annotation, or an @SecondaryTablesannotation in turn containing zero or more @SecondaryTable annotations. The @SecondaryTableannotation takes the same basic attributes as the @Table annotation, with the addition of the join attribute. The join attribute defines the join column for the primary database table. It accepts an array of javax.persistence.PrimaryKeyJoinColumn objects. If you omit the join attribute, then it will be assumed that the tables are joined on identically named primary key columns.

When an attribute in the entity is drawn from the secondary table, it must be marked with the @Columnannotation, with a table attribute identifying the appropriate table.

|  |
| --- |
| @Entity  @Table(name = "employee")  @SecondaryTable(name = "employee\_details")  public class EmployeeEntity implements Serializable  {     @Id     @GeneratedValue (strategy = GenerationType.SEQUENCE)     private Integer employeeId;     private String  firstName;     private String  lastName;       @Column(table = "employee\_details")     public String address;  } |

Columns in the primary or secondary tables can be marked as having unique values within their tables by adding one or more appropriate @UniqueConstraint annotations to @Table or @SecondaryTable’s uniqueConstraints attribute. Alternatively, You may also set uniqueness at the field level with the unique attribute on the @Column attribute.

|  |
| --- |
| @Entity  @Table(        name="employee",        uniqueConstraints={@UniqueConstraint(columnNames="firstName")}        )  @SecondaryTable(name = "employee\_details")  public class EmployeeEntity implements Serializable{    } |

**Persisting Basic Types with @Basic**

By default, properties and instance variables in your POJO are persistent; Hibernate will store their values for you. The simplest mappings are therefore for the “basic” types. These include primitives, primitive wrappers, arrays of primitives or wrappers, enumerations, and any types that implement Serializable but are not themselves mapped entities.

These are all mapped implicitly—no annotation is needed. By default, such fields are mapped to a single column, and eager fetching is used to retrieve them (i.e., when the entity is retrieved from the database, all the basic fields and properties are retrieved). Also, when the field or property is not a primitive, it can be stored and retrieved as a null value.

This default behavior can be overridden by applying the @Basic annotation to the appropriate class member. The annotation takes two optional attributes, and is itself entirely optional. The first attribute is named optional and takes a boolean. Defaulting to true, this can be set to false to provide a hint to schema generation that the associated column should be created NOT NULL. The second is named fetch and takes a member of the enumeration FetchType. This is EAGER by default, but can be set to LAZY to permit loading on access of the value.

|  |
| --- |
| @Basic (fetch = FetchType.LAZY, optional = false)  private String  firstName; |

The use of lazy loading is unlikely to be valuable, except when large serializable objects have been mapped as basic types (rather than given entity mappings of their own) and retrieval time may become significant. While the (default) EAGER value must be honored, the LAZY flag is considered to be a hint, and can be ignored by the persistence engine.

The @Basic attribute is usually omitted, with the @Column attribute being used where the @Basicannotation’s optional attribute might otherwise be used to provide the NOT NULL behavior.

**Omitting Persistence with @Transient**

Some fields, such as calculated values, may be used at run time only, and they should be discarded from objects as they are persisted into the database. The JPA specification provides the @Transientannotation for these transient fields. The @Transient annotation does not have any attributes—you just add it to the instance variable or the getter method as appropriate for the entity bean’s property access strategy.

The @Transient annotation highlights one of the more important differences between using annotations with Hibernate and using XML mapping documents. With annotations, Hibernate will default to persisting all of the fields on a mapped object. When using XML mapping documents, Hibernate requires you to tell it explicitly which fields will be persisted.

For example, if our EmployeeEntity has two additional fields “age” and “dateOfBirth” then you would like to store dateOfBirth in database, but you would like to calculate age in runtime based on value of dateOfBirth. So, ‘age’ field must be marked as transient.

|  |
| --- |
| @Transient  private Integer age; |

**Mapping Properties and Fields with @Column**

The @Column annotation is used to specify the details of the column to which a field or property will be mapped. Some of the details are schema related, and therefore apply only if the schema is generated from the annotated files. Others apply and are enforced at run time by Hibernate (or the JPA 2 persistence engine). It is optional, with an appropriate set of default behaviors, but is often useful when overriding default behavior, or when you need to fit your object model into a preexisting schema.

The following attributes commonly being overridden:

1. **name** : permits the name of the column to be explicitly specified—by default, this would be the name of the property.
2. **length** : permits the size of the column used to map a value (particularly a String value) to be explicitly defined. The column size defaults to 255, which might otherwise result in truncated String data, for example.
3. **nullable** : permits the column to be marked NOT NULL when the schema is generated. The default is that fields should be permitted to be null; however, it is common to override this when a field is, or ought to be, mandatory.
4. **unique** : permits the column to be marked as containing only unique values. This defaults to false, but commonly would be set for a value that might not be a primary key but would still cause problems if duplicated (such as username).

|  |
| --- |
| @Column(name="FNAME",length=100,nullable=false)  private String  firstName; |

There are some more attributes which are rather less used in real life projects. These are *table*, *insertable*, *updatable*, *columnDefinition*, *precision* and *scale*. I will leave you to explore them in detail.

**Modeling Entity Relationships**

I have already covered the modeling related concepts in separate detailed posts. Please read more about them in these linked articles, as having duplicate information here does not make sense.

1. [**Modeling @OneToOne relationship**](https://howtodoinjava.com/hibernate/hibernate-one-to-one-mapping-using-annotations/)
2. [**Modeling @OneToMany relationship**](https://howtodoinjava.com/hibernate/hibernate-one-to-many-mapping-using-annotations/)
3. [**Modeling @ManyToMany relationship**](https://howtodoinjava.com/hibernate/hibernate-many-to-many-mapping-using-annotations)

**Mapping Inheritance Hierarchies**

Entities are not always associated with other entities as attributes; sometimes they are related using normal OOPs inheritance rules. Hibernate allows you to honor such relationships using @Inheritance annotation.

The JPA 2 standard and Hibernate both support three approaches to mapping inheritance hierarchies into the database. These are as follows:

1. **Single table (SINGLE\_TABLE)**: One table for each class hierarchy
2. **Joined (JOINED)**: One table for each subclass (including interfaces and abstract classes)
3. **Table-per-class (TABLE\_PER\_CLASS)**: One table for each concrete class implementation

Persistent entities that are related by inheritance must be marked up with the @Inheritance annotation. This takes a single strategy attribute, which is set to one of three javax.persistence.InheritanceType enumeration values corresponding to these approaches (i.e. **SINGLE\_TABLE**, **JOINED** or **TABLE\_PER\_CLASS**).

Lets discuss them in some detail.

**Single Table**

The single-table approach manages one database table for the main superclass and all its subtypes. There are columns for each mapped field or property of the superclass, and for each distinct field or property of the derived types. When following this strategy, you will need to ensure that columns are appropriately renamed when any field or property names collide in the hierarchy.

To determine the appropriate type to instantiate when retrieving entities from the database, a @DiscriminatorColumn annotation should be provided in the root (and only in the root) of the persistent hierarchy.

Let’s look at a quick example for now. I am leaving you to read more on this in official hibernate documentation. I will cover them in detail in some later post.

|  |
| --- |
| //The Root of the Inheritance Hierarchy Mapped with the SINGLE\_TABLE Strategy    @Entity  @Inheritance(strategy = SINGLE\_TABLE)  @DiscriminatorColumn(      name="DISCRIMINATOR",      discriminatorType=INTEGER  )  @DiscriminatorValue("1")  public class Book {  ...  }    //A Derived Entity in the Inheritance Hierarchy  @Entity  @DiscriminatorValue("2")  public class ComputerBook extends Book {  ...  } |

**Joined Table**

An alternative to the monolithic single-table approach is the otherwise similar joined-table approach. Here a discriminator column is used, but the fields of the various derived types are stored in distinct tables.

|  |
| --- |
| @Entity  @Inheritance(strategy = JOINED)  @DiscriminatorColumn      name="DISCRIMINATOR"  )  public class Book {  ...  } |

**Table per Concrete Class**

Finally, there is the table-per-class approach, in which all of the fields of each type in the inheritance hierarchy are stored in distinct tables. Because of the close correspondence between the entity and its table, the @DiscriminatorColumn annotation is not applicable to this inheritance strategy.

|  |
| --- |
| @Entity  @Inheritance(strategy = TABLE\_PER\_CLASS)  public class Book {  ...  } |

**Other JPA 2 Persistence Annotations**

Although we have now covered most of the core JPA 2 persistence annotations, there are a few others that you will encounter fairly frequently. We cover some of these in passing in the following sections.

**Temporal Data with @Temporal**

Fields or properties of an entity that have java.util.Date or java.util.Calendar types represent temporal data. By default, these will be stored in a column with the **TIMESTAMP** data type, but this default behavior can be overridden with the @Temporal annotation.

The annotation accepts a single value attribute from the javax.persistence.TemporalTypeenumeration. This offers three possible values: **DATE**, **TIME**, and **TIMESTAMP**. These correspond, respectively, to java.sql.Date, java.sql.Time, and java.sql.Timestamp. The table column is given the appropriate data type at schema generation time.

|  |
| --- |
| @Temporal(TemporalType.TIME)  java.util.Date startingTime; |

**Element Collections with @ElementCollection**

In addition to mapping collections using one-to-many mappings, JPA 2 introduced an @ElementCollection annotation for mapping collections of basic or embeddable classes. You can use the @ElementCollection annotation to simplify your mappings.

|  |
| --- |
| @ElementCollection  List<String> passwordHints; |

There are two attributes on the @ElementCollection annotation: **targetClass** and **fetch**. The targetClass attribute tells Hibernate which class is stored in the collection. If you use generics on your collection, you do not need to specify targetClass because Hibernate will infer the correct class. The fetch attribute takes a member of the enumeration, FetchType. This is EAGER by default, but can be set to LAZY to permit loading when the value is accessed.

**Large Objects with @Lob**

A persistent property or field can be marked for persistence as a database-supported large object type by applying the @Lob annotation.

The annotation takes no attributes, but the underlying large object type to be used will be inferred from the type of the field or parameter. String- and character-based types will be stored in an appropriate character-based type i.e. CLOB. All other objects will be stored in a BLOB.

|  |
| --- |
| @Lob  String content; // a very long article |

The @Lob annotation can be used in combination with the @Basic or the @ElementCollectionannotation.

**Mapped Superclasses with @MappedSuperclass**

A special case of inheritance occurs when the root of the hierarchy is not itself a persistent entity, but various classes derived from it are. Such a class can be abstract or concrete. The @MappedSuperclassannotation allows you to take advantage of this circumstance.

The class marked with @MappedSuperclass is not an entity, and is not query-able (it cannot be passed to methods that expect an entity in the Session or EntityManager objects). It cannot be the target of an association.

The mapping information for the columns of the superclass will be stored in the same table as the details of the derived class.

**Ordering Collections with @OrderColumn**

While @OrderBy allows data to be ordered once it has been retrieved from the database, JPA 2 also provides an annotation that allows the ordering of appropriate collection types (e.g., List) to be maintained in the database; it does so by maintaining an order column to represent that order. Here’s an example:

|  |
| --- |
| @OneToMany  @OrderColumn(     name="employeeNumber"  )  List<Employee> employees; |

Here, we are declaring that an employeeNumber column will maintain a value, starting at 0 and incrementing as each entry is added to the list. The default starting value can be overridden by the base attribute. By default, the column can contain null (unordered) values. The nullability can be overridden by setting the nullable attribute to false. By default, when the schema is generated from the annotations, the column is assumed to be an integer type; however, this can be overridden by supplying a columnDefinition attribute specifying a different column definition string.

**Named Queries (HQL or JPQL)**

**@NamedQuery and @NamedQueries**

@NamedQuery and @NamedQueries allow one or more Hibernate Query Language or Java Persistence Query Language (JPQL) queries to be associated with an entity. The required attributes are as follows:

1. **name** is the name by which the query is retrieved.
2. **query** is the JPQL (or HQL) query associated with the name.

Take example of below “Author” entity.

|  |
| --- |
| @Entity  @NamedQuery(          name="findAuthorsByName",          query="from Author where name = :author"  )  public class Author {  ...  } |

The query would retrieve Author entities by name, so it is natural to associate it with that entity; however, there is no actual requirement that a named query be associated in this way with the entity that it concerns.

You do not need to directly associate the query with the entity against which it is declared, but it is normal to do so. If a query has no natural association with any of the entity declarations, it is possible to make the @NamedQuery annotation at the package level.

**Named Native Queries using @NamedNativeQuery and @NamedNativeQueries**

@NamedNativeQuery lets you write a named SQL query, while @NamedQuery lets you write a named HQL query (or JPQL).

In general, you should prefer to write HQL queries because then you can let Hibernate handle the intricacies of converting the HQL into the various SQL dialects. This will make your job much simpler when you choose to switch DBMS providers.

|  |
| --- |
| @NamedQueries({     @NamedQuery(name="get-emp-by-name",query="FROM EmployeeBean WHERE fName=:fName")  })    //Equivalent NamedNativeQuery    @NamedNativeQueries(      {          @NamedNativeQuery(              name="get-emp-by-name-native",              query="SELECT \* FROM Employees WHERE firstName=:fName",              resultClass=EmployeeEntity.class)      }  ) |

That’s all about this limited tutorial covering **most important JPA 2 persistence annotations** in short. I will be covering them in detail in later tutorials.

**Hibernate One-to-Many Mapping Using Annotations**

November 17, 2012 by Lokesh Gupta

One to many mapping is made between two entities where first entity can have relation with multiple second entity instances but second can be associated with only one instance of first entity. Its 1 to n relationship. For example, in any company an employee can register multiple bank accounts but one bank account will be associated with one and only one employee. In this post, we will learn to make such mapping in database using hibernate 3.

[**Download source code**](https://docs.google.com/open?id=0B7yo2HclmjI4b2dvbk40cUtvSzQ)

**Sections in this post**

Problem statement

Designing the solution

Using foreign key association

Using a join table

**Problem statement**

We have to write two entities i.e. EmployeeEntity and AccountEntity such that multiple accounts can be associated with a single employee, but these accounts can not be shared between two or more employees.

**Designing the solution**

This problem can be solved in two different ways. One is to have a foreign key column in account table i.i EMPLOYEE\_ID. This column will refer to primary key of Employee table. This way no two accounts can be associated with multiple employees. Obviously, account number needs to be unique for enforcing this restriction.

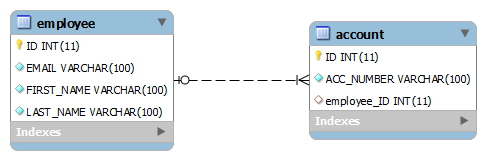
Another approach is to have a common join table lets say EMPLOYEE\_ACCOUNT. This table will have two column i.e. EMP\_ID which will be foreign key referring to primary key in EMPLOYEE table and similarly ACCOUNT\_ID which will be foreign key referring to primary key of ACCOUNT table.

Lets write some code for both approaches.

**Using foreign key association**

In this approach, both entity will be responsible for making the relationship and maintaining it. EmployeeEntity should declare that relationship is One to many, and AccountEntity should declare that relationship from its end is many to one.

Lets first see the schema design.

[](https://howtodoinjava.files.wordpress.com/2012/11/one-to-many-association-in-hiberate-using-foreign-key.png)

Lets write **EmployeeEntity.java**

|  |
| --- |
| package hibernate.test.oneToMany.foreignKeyAsso;    import java.io.Serializable;  import java.util.Set;    import javax.persistence.CascadeType;  import javax.persistence.Column;  import javax.persistence.Entity;  import javax.persistence.GeneratedValue;  import javax.persistence.GenerationType;  import javax.persistence.Id;  import javax.persistence.JoinColumn;  import javax.persistence.OneToMany;  import javax.persistence.Table;  import javax.persistence.UniqueConstraint;    @Entity(name = "ForeignKeyAssoEntity")  @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;        @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;        @OneToMany(cascade=CascadeType.ALL)      @JoinColumn(name="EMPLOYEE\_ID")      private Set<AccountEntity> accounts;        public Integer getEmployeeId() {          return employeeId;      }        public void setEmployeeId(Integer employeeId) {          this.employeeId = employeeId;      }        public String getEmail() {          return email;      }        public void setEmail(String email) {          this.email = email;      }        public String getFirstName() {          return firstName;      }        public void setFirstName(String firstName) {          this.firstName = firstName;      }        public String getLastName() {          return lastName;      }        public void setLastName(String lastName) {          this.lastName = lastName;      }        public Set<AccountEntity> getAccounts() {          return accounts;      }        public void setAccounts(Set<AccountEntity> accounts) {          this.accounts = accounts;      }  } |

Write **AccountEntity.java**

|  |
| --- |
| package hibernate.test.oneToMany.foreignKeyAsso;    import java.io.Serializable;    import javax.persistence.Column;  import javax.persistence.Entity;  import javax.persistence.GeneratedValue;  import javax.persistence.GenerationType;  import javax.persistence.Id;  import javax.persistence.ManyToOne;  import javax.persistence.Table;  import javax.persistence.UniqueConstraint;    @Entity(name = "ForeignKeyAssoAccountEntity")  @Table(name = "ACCOUNT", uniqueConstraints = {  @UniqueConstraint(columnNames = "ID")})  public class AccountEntity implements Serializable  {        private static final long serialVersionUID = -6790693372846798580L;        @Id      @GeneratedValue(strategy = GenerationType.IDENTITY)      @Column(name = "ID", unique = true, nullable = false)      private Integer accountId;        @Column(name = "ACC\_NUMBER", unique = true, nullable = false, length = 100)      private String accountNumber;        @ManyToOne      private EmployeeEntity employee;        public Integer getAccountId() {          return accountId;      }        public void setAccountId(Integer accountId) {          this.accountId = accountId;      }        public String getAccountNumber() {          return accountNumber;      }        public void setAccountNumber(String accountNumber) {          this.accountNumber = accountNumber;      }        public EmployeeEntity getEmployee() {          return employee;      }        public void setEmployee(EmployeeEntity employee) {          this.employee = employee;      }  } |

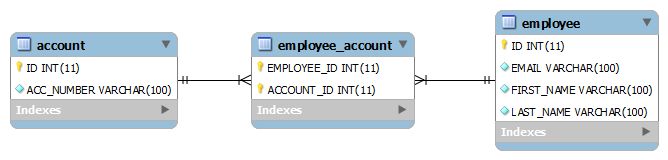
**Lets Test the code:**

|  |
| --- |
| package hibernate.test.oneToMany;    import hibernate.test.HibernateUtil;  import hibernate.test.oneToMany.foreignKeyAsso.AccountEntity;  import hibernate.test.oneToMany.foreignKeyAsso.EmployeeEntity;    import java.util.HashSet;  import java.util.Set;    import org.hibernate.Session;    public class TestForeignKeyAssociation  {        public static void main(String[] args)      {          Session session = HibernateUtil.getSessionFactory().openSession();          session.beginTransaction();            AccountEntity account1 = new AccountEntity();          account1.setAccountNumber("Account detail 1");            AccountEntity account2 = new AccountEntity();          account2.setAccountNumber("Account detail 2");            AccountEntity account3 = new AccountEntity();          account3.setAccountNumber("Account detail 3");            //Add new Employee object          EmployeeEntity firstEmployee = new EmployeeEntity();          firstEmployee.setEmail("demo-user-first@mail.com");          firstEmployee.setFirstName("demo-one");          firstEmployee.setLastName("user-one");            EmployeeEntity secondEmployee = new EmployeeEntity();          secondEmployee.setEmail("demo-user-second@mail.com");          secondEmployee.setFirstName("demo-two");          secondEmployee.setLastName("user-two");            Set<AccountEntity> accountsOfFirstEmployee = new HashSet<AccountEntity>();          accountsOfFirstEmployee.add(account1);          accountsOfFirstEmployee.add(account2);            Set<AccountEntity> accountsOfSecondEmployee = new HashSet<AccountEntity>();          accountsOfSecondEmployee.add(account3);            firstEmployee.setAccounts(accountsOfFirstEmployee);          secondEmployee.setAccounts(accountsOfSecondEmployee);          //Save Employee          session.save(firstEmployee);          session.save(secondEmployee);            session.getTransaction().commit();          HibernateUtil.shutdown();      }  }  Output:    Hibernate: insert into Employee (EMAIL, FIRST\_NAME, LAST\_NAME) values (?, ?, ?)  Hibernate: insert into ACCOUNT (ACC\_NUMBER, employee\_ID) values (?, ?)  Hibernate: insert into ACCOUNT (ACC\_NUMBER, employee\_ID) values (?, ?)  Hibernate: insert into Employee (EMAIL, FIRST\_NAME, LAST\_NAME) values (?, ?, ?)  Hibernate: insert into ACCOUNT (ACC\_NUMBER, employee\_ID) values (?, ?)  Hibernate: update ACCOUNT set EMPLOYEE\_ID=? where ID=?  Hibernate: update ACCOUNT set EMPLOYEE\_ID=? where ID=?  Hibernate: update ACCOUNT set EMPLOYEE\_ID=? where ID=? |

**Using a join table**

This approach uses a join table to store the associations between account and employee entities. @JoinTable annotation has been used to make this association.

Lets see how the database schema will look like:

[](https://howtodoinjava.files.wordpress.com/2012/11/one-to-many-association-in-hiberate-using-join-table.png)one To Many association in hibernate using join table

**EmployeeEntity.java**

|  |
| --- |
| package hibernate.test.oneToMany.joinTable;    import java.io.Serializable;  import java.util.Set;    import javax.persistence.CascadeType;  import javax.persistence.Column;  import javax.persistence.Entity;  import javax.persistence.GeneratedValue;  import javax.persistence.GenerationType;  import javax.persistence.Id;  import javax.persistence.JoinColumn;  import javax.persistence.JoinTable;  import javax.persistence.OneToMany;  import javax.persistence.Table;  import javax.persistence.UniqueConstraint;    @Entity(name = "JoinTableEmployeeEntity")  @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;        @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;        @OneToMany(cascade=CascadeType.ALL)      @JoinTable(name="EMPLOYEE\_ACCOUNT", joinColumns={@JoinColumn(name="EMPLOYEE\_ID", referencedColumnName="ID")}      , inverseJoinColumns={@JoinColumn(name="ACCOUNT\_ID", referencedColumnName="ID")})      private Set<AccountEntity> accounts;        public Integer getEmployeeId() {          return employeeId;      }        public void setEmployeeId(Integer employeeId) {          this.employeeId = employeeId;      }        public String getEmail() {          return email;      }        public void setEmail(String email) {          this.email = email;      }        public String getFirstName() {          return firstName;      }        public void setFirstName(String firstName) {          this.firstName = firstName;      }        public String getLastName() {          return lastName;      }        public void setLastName(String lastName) {          this.lastName = lastName;      }        public Set<AccountEntity> getAccounts() {          return accounts;      }        public void setAccounts(Set<AccountEntity> accounts) {          this.accounts = accounts;      }  } |

**AccountEntity.java**

|  |
| --- |
| package hibernate.test.oneToMany.joinTable;    import java.io.Serializable;    import javax.persistence.Column;  import javax.persistence.Entity;  import javax.persistence.GeneratedValue;  import javax.persistence.GenerationType;  import javax.persistence.Id;  import javax.persistence.Table;  import javax.persistence.UniqueConstraint;    @Entity(name = "JoinTableAccountEntity")  @Table(name = "ACCOUNT", uniqueConstraints = {  @UniqueConstraint(columnNames = "ID")})  public class AccountEntity implements Serializable  {        private static final long serialVersionUID = -6790693372846798580L;        @Id      @GeneratedValue(strategy = GenerationType.IDENTITY)      @Column(name = "ID", unique = true, nullable = false)      private Integer accountId;        @Column(name = "ACC\_NUMBER", unique = true, nullable = false, length = 100)      private String accountNumber;        public Integer getAccountId() {          return accountId;      }        public void setAccountId(Integer accountId) {          this.accountId = accountId;      }        public String getAccountNumber() {          return accountNumber;      }        public void setAccountNumber(String accountNumber) {          this.accountNumber = accountNumber;      }  } |

**Configuring entities in config file**

**We have available both entities to runtime, we have to add them in hibernate.cfg.xml file. Please note that only one set of entities should be configured in configuration file otherwise unexpected results can occur.**

|  |
| --- |
| < ?xml version="1.0" encoding="utf-8"?>  < !DOCTYPE hibernate-configuration PUBLIC  "-//Hibernate/Hibernate Configuration DTD 3.0//EN"  "<http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd>">  <hibernate-configuration>      <session-factory>          <property name="hibernate.connection.driver\_class">com.mysql.jdbc.Driver</property>          <propertyname="hibernate.connection.url">jdbc:[mysql://localhost:3306/hibernatetest</property](mysql://localhost:3306/hibernatetest%3c/property)>          <property name="hibernate.connection.password">XXXXXX</property>          <property name="hibernate.connection.username">root</property>          <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>          <property name="show\_sql">true</property>          <property name="hbm2ddl.auto">create</property>          <mapping clas="hibernate.test.oneToMany.foreignKeyAsso.AccountEntity"></mapping>          <mapping clas="hibernate.test.oneToMany.foreignKeyAsso.EmployeeEntity"></mapping>      </session-factory>  </hibernate-configuration> |

**Testing the code**

Now, its time to test the code. I have written following code to test above entities.

|  |
| --- |
| package hibernate.test.oneToMany;    import hibernate.test.HibernateUtil;  import hibernate.test.oneToMany.joinTable.AccountEntity;  import hibernate.test.oneToMany.joinTable.EmployeeEntity;    import java.util.HashSet;  import java.util.Set;    import org.hibernate.Session;    public class TestJoinTable  {      public static void main(String[] args)      {          Session session = HibernateUtil.getSessionFactory().openSession();          session.beginTransaction();            AccountEntity account1 = new AccountEntity();          account1.setAccountNumber("123-345-65454");            AccountEntity account2 = new AccountEntity();          account2.setAccountNumber("123-345-6542222");            //Add new Employee object          EmployeeEntity emp = new EmployeeEntity();          emp.setEmail("demo-user@mail.com");          emp.setFirstName("demo");          emp.setLastName("user");            Set<AccountEntity> accounts = new HashSet<AccountEntity>();          accounts.add(account1);          accounts.add(account2);            emp.setAccounts(accounts);          //Save Employee          session.save(emp);            session.getTransaction().commit();          HibernateUtil.shutdown();      }  }    Output:    Hibernate: insert into Employee (EMAIL, FIRST\_NAME, LAST\_NAME) values (?, ?, ?)  Hibernate: insert into ACCOUNT (ACC\_NUMBER) values (?)  Hibernate: insert into ACCOUNT (ACC\_NUMBER) values (?)  Hibernate: insert into EMPLOYEE\_ACCOUNT (EMPLOYEE\_ID, ACCOUNT\_ID) values (?, ?)  Hibernate: insert into EMPLOYEE\_ACCOUNT (EMPLOYEE\_ID, ACCOUNT\_ID) values (?, ?) |

**Hibernate Many-to-Many Mapping Using Annotations**

November 17, 2012 by Lokesh Gupta

Many to many mapping is made between two entities where one can have relation with multiple other entity instances. For example, for a subscription service SubscriptionEntity and ReaderEntity can be two type of entities. Any subscription can have multiple readers, where a reader can subscribe to multiple subscriptions. In this post, we will learn to make such mapping in database using hibernate 3.

[**Download source code**](https://drive.google.com/file/d/0B7yo2HclmjI4ZHdoLVF2Zk1Xc1E/view?usp=drive_web)

**Sections in this post**

Proposed solution

Writing owner entity

Writing mapped entity

Configuring entities in config file

Testing the code

**Proposed solution**

To demonstrate many to many mapping, we will associate two entities i.e. ReaderEntity and SubscriptionEntity. Their database schema should look like this. Using these tables, any application can save multiple associations between readers and subscriptions.

[](https://howtodoinjava.files.wordpress.com/2012/11/many-to-many-hibernate-mapping.png)

**Writing owner entity**

Owner entity is the entity which is responsible make making the association and maintaining it. In our case, I am making ReaderEntity the owner entity. @JoinTable annotation has been used to make this association.

|  |
| --- |
| package hibernate.test.manyToMany.joinTable;    import java.io.Serializable;  import java.util.Set;    import javax.persistence.CascadeType;  import javax.persistence.Column;  import javax.persistence.Entity;  import javax.persistence.GeneratedValue;  import javax.persistence.GenerationType;  import javax.persistence.Id;  import javax.persistence.JoinColumn;  import javax.persistence.JoinTable;  import javax.persistence.ManyToMany;  import javax.persistence.Table;  import javax.persistence.UniqueConstraint;    @Entity(name = "ReaderEntity")  @Table(name = "READER", uniqueConstraints = {          @UniqueConstraint(columnNames = "ID"),          @UniqueConstraint(columnNames = "EMAIL") })  public class ReaderEntity implements Serializable {        private static final long serialVersionUID = -1798070786993154676L;        @Id      @GeneratedValue(strategy = GenerationType.IDENTITY)      @Column(name = "ID", unique = true, nullable = false)      private Integer readerId;        @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;        @ManyToMany(cascade=CascadeType.ALL)      @JoinTable(name="READER\_SUBSCRIPTIONS", joinColumns={@JoinColumn(referencedColumnName="ID")}                                          , inverseJoinColumns={@JoinColumn(referencedColumnName="ID")})      private Set<SubscriptionEntity> subscriptions;        public Integer getReaderId() {          return readerId;      }        public void setReaderId(Integer readerId) {          this.readerId = readerId;      }        public String getEmail() {          return email;      }        public void setEmail(String email) {          this.email = email;      }        public String getFirstName() {          return firstName;      }        public void setFirstName(String firstName) {          this.firstName = firstName;      }        public String getLastName() {          return lastName;      }        public void setLastName(String lastName) {          this.lastName = lastName;      }        public Set<SubscriptionEntity> getSubscriptions() {          return subscriptions;      }        public void setSubscriptions(Set<SubscriptionEntity> subscriptions) {          this.subscriptions = subscriptions;      }  } |

**Writing mapped entity**

Our mapped entity is SubscriptionEntity which is mapped to ReaderEntity using “**mappedBy**” attribute.

|  |
| --- |
| package hibernate.test.manyToMany.joinTable;    import java.io.Serializable;  import java.util.Set;    import javax.persistence.Column;  import javax.persistence.Entity;  import javax.persistence.GeneratedValue;  import javax.persistence.GenerationType;  import javax.persistence.Id;  import javax.persistence.ManyToMany;  import javax.persistence.Table;  import javax.persistence.UniqueConstraint;    @Entity(name = "SubscriptionEntity")  @Table(name = "SUBSCRIPTION", uniqueConstraints = {          @UniqueConstraint(columnNames = "ID")})  public class SubscriptionEntity implements Serializable  {        private static final long serialVersionUID = -6790693372846798580L;        @Id      @GeneratedValue(strategy = GenerationType.IDENTITY)      @Column(name = "ID", unique = true, nullable = false)      private Integer subscriptionId;        @Column(name = "SUBS\_NAME", unique = true, nullable = false, length = 100)      private String subscriptionName;        @ManyToMany(mappedBy="subscriptions")      private Set<ReaderEntity> readers;        public Integer getSubscriptionId() {          return subscriptionId;      }        public void setSubscriptionId(Integer subscriptionId) {          this.subscriptionId = subscriptionId;      }        public String getSubscriptionName() {          return subscriptionName;      }        public void setSubscriptionName(String subscriptionName) {          this.subscriptionName = subscriptionName;      }        public Set<ReaderEntity> getReaders() {          return readers;      }        public void setReaders(Set<ReaderEntity> readers) {          this.readers = readers;      }  } |

**Configuring entities in config file**

We have available both entities to runtime, we have to add them in ***hibernate.cfg.xml*** file.

|  |
| --- |
| <?xml version="1.0" encoding="utf-8"?>  <!DOCTYPE hibernate-configuration PUBLIC  "-//Hibernate/Hibernate Configuration DTD 3.0//EN"  "<http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd>">  <hibernate-configuration>      <session-factory>          <property name="hibernate.connection.driver\_class">com.mysql.jdbc.Driver</property>          <propertyname="hibernate.connection.url">jdbc:[mysql://localhost:3306/hibernatetest</property](mysql://localhost:3306/hibernatetest%3c/property)>          <property name="hibernate.connection.password">XXXXXX</property>          <property name="hibernate.connection.username">root</property>          <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>          <property name="show\_sql">true</property>          <property name="hbm2ddl.auto">create</property>          <mapping class="hibernate.test.manyToMany.joinTable.ReaderEntity"/>          <mapping class="hibernate.test.manyToMany.joinTable.SubscriptionEntity"/>      </session-factory>  </hibernate-configuration> |

**Testing the code**

Now, its time to test the code. I have written following code to test above entities.

|  |
| --- |
| package hibernate.test.manyToMany;    import hibernate.test.HibernateUtil;  import hibernate.test.manyToMany.joinTable.\*;    import java.util.HashSet;  import java.util.Set;    import org.hibernate.Session;    public class TestJoinTable  {        public static void main(String[] args)      {          Session session = HibernateUtil.getSessionFactory().openSession();          session.beginTransaction();            //Add subscription          SubscriptionEntity subOne = new SubscriptionEntity();          subOne.setSubscriptionName("Entertainment");            SubscriptionEntity subTwo = new SubscriptionEntity();          subTwo.setSubscriptionName("Horror");            Set<SubscriptionEntity> subs = new HashSet<SubscriptionEntity>();          subs.add(subOne);          subs.add(subTwo);            //Add readers          ReaderEntity readerOne = new ReaderEntity();          readerOne.setEmail("demo-user1@mail.com");          readerOne.setFirstName("demo");          readerOne.setLastName("user");            ReaderEntity readerTwo = new ReaderEntity();          readerTwo.setEmail("demo-user2@mail.com");          readerTwo.setFirstName("demo");          readerTwo.setLastName("user");            Set<ReaderEntity> readers = new HashSet<ReaderEntity>();          readers.add(readerOne);          readers.add(readerTwo);            readerOne.setSubscriptions(subs);          readerTwo.setSubscriptions(subs);            session.save(readerOne);          session.save(readerTwo);            session.getTransaction().commit();          HibernateUtil.shutdown();      }  }    Output:    Hibernate: insert into READER (EMAIL, FIRST\_NAME, LAST\_NAME) values (?, ?, ?)  Hibernate: insert into SUBSCRIPTION (SUBS\_NAME) values (?)  Hibernate: insert into SUBSCRIPTION (SUBS\_NAME) values (?)  Hibernate: insert into READER (EMAIL, FIRST\_NAME, LAST\_NAME) values (?, ?, ?)  Hibernate: insert into READER\_SUBSCRIPTIONS (readers\_ID, subscriptions\_ID) values (?, ?)  Hibernate: insert into READER\_SUBSCRIPTIONS (readers\_ID, subscriptions\_ID) values (?, ?)  Hibernate: insert into READER\_SUBSCRIPTIONS (readers\_ID, subscriptions\_ID) values (?, ?)  Hibernate: insert into READER\_SUBSCRIPTIONS (readers\_ID, subscriptions\_ID) values (?, ?) |

**Hibernate One-to-One Mapping Using Annotations**

November 15, 2012 by Lokesh Gupta

If you are working on any hibernate project or you are planning to work on any in future, then you can easily understand the one-to-one relationships between several entities in your application. In this post, i will discuss variations of one-to-one mappings supported in hibernate.

[**Download source code**](https://drive.google.com/file/d/0B7yo2HclmjI4VHJnQk4tYjBueDA/view?usp=drive_web)

**Sections in this post:**

Various supported techniques

Using foreign key association

Using a common join table

Using shared primary key

For this article, I am extending the example written [**for hello world example**](https://howtodoinjava.com/hibernate/hibernate-3-introduction-and-writing-hello-world-application/). We have two entities here: *Employee* and *Account*.

**Various supported techniques**

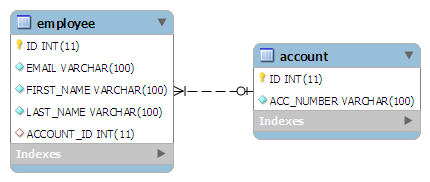
In hibernate there are 3 ways to create one-to-one relationships between two entities. Either way you have to use [**@OneToOne**](http://docs.oracle.com/javaee/5/api/javax/persistence/OneToOne.html) annotation. ***First technique*** is most widely used and uses a foreign key column in one to table. ***Second 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.

Lets see them in action one by one:

**Using foreign key association**

**In this association, a foreign key column is created in owner entity.**For example, if we make EmployeeEntity owner, then a extra column “ACCOUNT\_ID” will be created in Employee table. This column will store the foreign key for Account table.

Table structure will be like this:

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

To make such association, refer the account entity in EmployeeEntity class as follow:

|  |
| --- |
| @OneToOne  @JoinColumn(name="ACCOUNT\_ID")  private AccountEntity account; |

The join column is declared with the [**@JoinColumn**](http://docs.oracle.com/javaee/5/api/javax/persistence/JoinColumn.html) annotation which looks like the [**@Column**](http://docs.oracle.com/javaee/5/api/javax/persistence/Column.html)annotation. It has one more parameters named referencedColumnName. This parameter declares the column in the targeted entity that will be used to the join.

If no @JoinColumn is declared on the owner side, the defaults apply. A join column(s) will be created in the owner table and its name will be the concatenation of the name of the relationship in the owner side, \_ (underscore), and the name of the primary key column(s) in the owned side.

In a bidirectional relationship, one of the sides (and only one) has to be the owner: the owner is responsible for the association column(s) update. *To declare a side as not responsible for the relationship, the attribute*[***mappedBy***](http://docs.oracle.com/javaee/5/api/javax/persistence/OneToOne.html#mappedBy%28%29)*is used*. mappedBy refers to the property name of the association on the owner side.

|  |
| --- |
| @OneToOne(mappedBy="account")  private EmployeeEntity employee; |

**Above “mappedBy” attribute declares that it is dependent on owner entity for mapping.**

Lets test above mappings in running code:

|  |
| --- |
| public class TestForeignKeyAssociation {        public static void main(String[] args) {          Session session = HibernateUtil.getSessionFactory().openSession();          session.beginTransaction();            AccountEntity account = new AccountEntity();          account.setAccountNumber("123-345-65454");            // Add new Employee object          EmployeeEntity emp = new EmployeeEntity();          emp.setEmail("demo-user@mail.com");          emp.setFirstName("demo");          emp.setLastName("user");            // Save Account          session.saveOrUpdate(account);          // Save Employee          emp.setAccount(account);          session.saveOrUpdate(emp);            session.getTransaction().commit();          HibernateUtil.shutdown();      }  } |

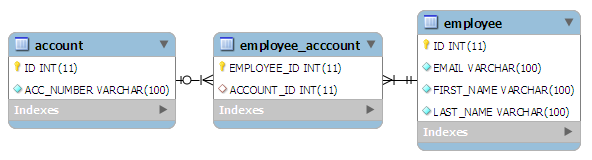
Running above code creates desired schema in database and run these SQL queries.

|  |
| --- |
| Hibernate: insert into ACCOUNT (ACC\_NUMBER) values (?)  Hibernate: insert into Employee (ACCOUNT\_ID, EMAIL, FIRST\_NAME, LAST\_NAME) values (?, ?, ?, ?) |

You can verify the data and mappings in both tables when you run above program.. :-)

**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; |

@JoinTable annotation is used in EmployeeEntity class. It declares that a new table EMPLOYEE\_ACCOUNT will be created with two columns EMPLOYEE\_ID (primary key of EMPLOYEE table) and ACCOUNT\_ID (primary key of ACCOUNT table).

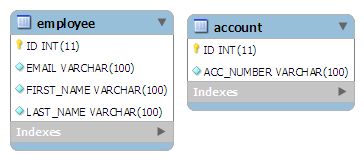
Testing above entities generates following SQL queries in log files:

|  |
| --- |
| Hibernate: insert into ACCOUNT (ACC\_NUMBER) values (?)  Hibernate: insert into Employee (EMAIL, FIRST\_NAME, LAST\_NAME) values (?, ?, ?)  Hibernate: insert into EMPLOYEE\_ACCCOUNT (ACCOUNT\_ID, EMPLOYEE\_ID) values (?, ?) |

**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; |

In AccountEntity side, it will remain dependent on owner entity for the mapping.

|  |
| --- |
| @OneToOne(mappedBy="account", cascade=CascadeType.ALL)  private EmployeeEntity employee; |

Testing above entities generates following SQL queries in log files:

|  |
| --- |
| Hibernate: insert into ACCOUNT (ACC\_NUMBER) values (?)  Hibernate: insert into Employee (ACCOUNT\_ID, EMAIL, FIRST\_NAME, LAST\_NAME) values (?, ?, ?, ?) |

So, we have seen all 3 types of one to one mappings supported in hibernate. I will suggest you to download the source code and play with it.