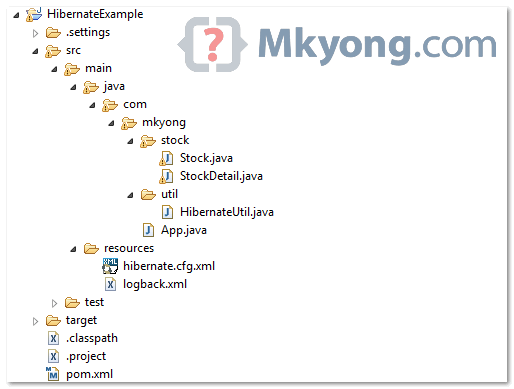
# **Hibernate – One-to-One example (Annotation)**

## **Project Structure**

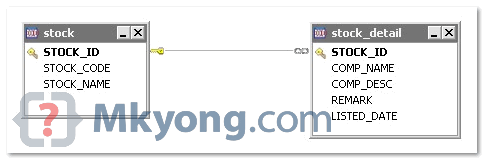
See the final project structure of this tutorial.



**Note**  
Since Hibernate 3.6, annotation codes are merged into the Hibernate core module, so, the [“previous pom.xml](https://www.mkyong.com/hibernate/hibernate-one-to-one-relationship-example-annotation/%3Ca%20href=) file can be reuse.

## **1. “One-to-one” table relationship**

See the previous one to one table relationship again.



## **2. Hibernate Model Class**

Create two model classes – Stock.java and StockDetail.java, and put the annotation code inside.

File : Stock.java

package com.mkyong.stock;

import javax.persistence.CascadeType;

import javax.persistence.Column;

import javax.persistence.Entity;

import javax.persistence.FetchType;

import javax.persistence.GeneratedValue;

import static javax.persistence.GenerationType.IDENTITY;

import javax.persistence.Id;

import javax.persistence.OneToOne;

import javax.persistence.Table;

import javax.persistence.UniqueConstraint;

@Entity

@Table(name = "stock", catalog = "mkyongdb", uniqueConstraints = {

@UniqueConstraint(columnNames = "STOCK\_NAME"),

@UniqueConstraint(columnNames = "STOCK\_CODE") })

public class Stock implements java.io.Serializable {

private Integer stockId;

private String stockCode;

private String stockName;

private StockDetail stockDetail;

public Stock() {

}

public Stock(String stockCode, String stockName) {

this.stockCode = stockCode;

this.stockName = stockName;

}

public Stock(String stockCode, String stockName, StockDetail stockDetail) {

this.stockCode = stockCode;

this.stockName = stockName;

this.stockDetail = stockDetail;

}

@Id

@GeneratedValue(strategy = IDENTITY)

@Column(name = "STOCK\_ID", unique = true, nullable = false)

public Integer getStockId() {

return this.stockId;

}

public void setStockId(Integer stockId) {

this.stockId = stockId;

}

@Column(name = "STOCK\_CODE", unique = true, nullable = false, length = 10)

public String getStockCode() {

return this.stockCode;

}

public void setStockCode(String stockCode) {

this.stockCode = stockCode;

}

@Column(name = "STOCK\_NAME", unique = true, nullable = false, length = 20)

public String getStockName() {

return this.stockName;

}

public void setStockName(String stockName) {

this.stockName = stockName;

}

@OneToOne(fetch = FetchType.LAZY, mappedBy = "stock", cascade = CascadeType.ALL)

public StockDetail getStockDetail() {

return this.stockDetail;

}

public void setStockDetail(StockDetail stockDetail) {

this.stockDetail = stockDetail;

}

}

Copy

File : StockDetail.java

package com.mkyong.stock;

import java.util.Date;

import javax.persistence.Column;

import javax.persistence.Entity;

import javax.persistence.FetchType;

import javax.persistence.GeneratedValue;

import javax.persistence.Id;

import javax.persistence.OneToOne;

import javax.persistence.PrimaryKeyJoinColumn;

import javax.persistence.Table;

import javax.persistence.Temporal;

import javax.persistence.TemporalType;

import org.hibernate.annotations.GenericGenerator;

import org.hibernate.annotations.Parameter;

@Entity

@Table(name = "stock\_detail", catalog = "mkyongdb")

public class StockDetail implements java.io.Serializable {

private Integer stockId;

private Stock stock;

private String compName;

private String compDesc;

private String remark;

private Date listedDate;

public StockDetail() {

}

public StockDetail(Stock stock, String compName, String compDesc,

String remark, Date listedDate) {

this.stock = stock;

this.compName = compName;

this.compDesc = compDesc;

this.remark = remark;

this.listedDate = listedDate;

}

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

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

@Id

@GeneratedValue(generator = "generator")

@Column(name = "STOCK\_ID", unique = true, nullable = false)

public Integer getStockId() {

return this.stockId;

}

public void setStockId(Integer stockId) {

this.stockId = stockId;

}

@OneToOne(fetch = FetchType.LAZY)

@PrimaryKeyJoinColumn

public Stock getStock() {

return this.stock;

}

public void setStock(Stock stock) {

this.stock = stock;

}

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

public String getCompName() {

return this.compName;

}

public void setCompName(String compName) {

this.compName = compName;

}

@Column(name = "COMP\_DESC", nullable = false)

public String getCompDesc() {

return this.compDesc;

}

public void setCompDesc(String compDesc) {

this.compDesc = compDesc;

}

@Column(name = "REMARK", nullable = false)

public String getRemark() {

return this.remark;

}

public void setRemark(String remark) {

this.remark = remark;

}

@Temporal(TemporalType.DATE)

@Column(name = "LISTED\_DATE", nullable = false, length = 10)

public Date getListedDate() {

return this.listedDate;

}

public void setListedDate(Date listedDate) {

this.listedDate = listedDate;

}

}

Copy

## **3. Hibernate Configuration File**

Puts annotated classes Stock.java and StockDetail.java in your Hibernate configuration file, and also MySQL connection details.

File : hibernate.cfg.xml

<?xml version="1.0" encoding="utf-8"?>

<!DOCTYPE hibernate-configuration PUBLIC

"-//Hibernate/Hibernate Configuration DTD 3.0//EN"

"http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

<session-factory>

<property name="hibernate.connection.driver\_class">com.mysql.jdbc.Driver</property>

<property name="hibernate.connection.url">jdbc:mysql://localhost:3306/mkyongdb</property>

<property name="hibernate.connection.username">root</property>

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

<property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>

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

<mapping class="com.mkyong.stock.Stock" />

<mapping class="com.mkyong.stock.StockDetail" />

</session-factory>

</hibernate-configuration>

Copy

## **4. Run It**

Run it, Hibernate will insert a row into the STOCK table and a row into the STOCK\_DETAIL table.

File : App.java

package com.mkyong;

import java.util.Date;

import org.hibernate.Session;

import com.mkyong.stock.Stock;

import com.mkyong.stock.StockDetail;

import com.mkyong.util.HibernateUtil;

public class App {

public static void main(String[] args) {

System.out.println("Hibernate one to one (Annotation)");

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

session.beginTransaction();

Stock stock = new Stock();

stock.setStockCode("7052");

stock.setStockName("PADINI");

StockDetail stockDetail = new StockDetail();

stockDetail.setCompName("PADINI Holding Malaysia");

stockDetail.setCompDesc("one stop shopping");

stockDetail.setRemark("vinci vinci");

stockDetail.setListedDate(new Date());

stock.setStockDetail(stockDetail);

stockDetail.setStock(stock);

session.save(stock);

session.getTransaction().commit();

System.out.println("Done");

}

}

Copy

Output

Hibernate one to one (Annotation)

Hibernate: insert into mkyongdb.stock (STOCK\_CODE, STOCK\_NAME) values (?, ?)

Hibernate: insert into mkyongdb.stock\_detail

(COMP\_DESC, COMP\_NAME, LISTED\_DATE, REMARK, STOCK\_ID) values (?, ?, ?, ?, ?)

Done

# Object States in Hibernate – Transient,Persistent and Detached

Object states in Hibernate plays a vital role in the execution of code in an application. Hibernate has provided three different states for an object of a pojo class. These three states are also called as life cycle states of an object.

## Hibernate Object States

There are three types of Hibernate object states.

## 1. Transient Object State:

An object which is not associated with hibernate session and does not represent a row in the database is considered as transient. It will be garbage collected if no other object refers to it.  
An object that is created for the first time using the new() operator is in transient state. When the object is in transient sate then it will not contain any identifier (primary key value). You have to use save, persist or saveOrUpdate methods to persist the transient object.

|  |  |
| --- | --- |
| 1  2  3 | Employee emp = new Employee();  emp.setName("Ravi Raj");  // here emp object is in a transient state |

## 2. Persistent Object State:

An object that is associated with the hibernate session is called as Persistent object. When the object is in persistent state, then it represent one row of the database and consists of an identifier value.You can make a transient instance persistent by associating it with a Session.

[?](http://javawebtutor.com/articles/hibernate/hibernate-object-states.php)

|  |  |
| --- | --- |
| 1  2 | Long id = (Long) session.save(emp);  // emp object is now in a persistent state |

## 3. Detached Object State:

Object which is just removed from hibernate session is called as detached object.The sate of the detached object is called as detached state.

When the object is in detached sate then it contain identity but you can’t do persistence operation with that identity.

Any changes made to the detached objects are not saved to the database. The detached object can be reattached to the new session and save to the database using update, saveOrUpdate and merge methods.

[?](http://javawebtutor.com/articles/hibernate/hibernate-object-states.php)

|  |  |
| --- | --- |
| 1  2 | session.close();  //object in detached state |

## Example

[?](http://javawebtutor.com/articles/hibernate/hibernate-object-states.php)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | package com.jwt.hibernate.test;    import org.hibernate.Session;  import org.hibernate.Transaction;  import org.hibernate.cfg.AnnotationConfiguration;    import com.jwt.hibernate.Student;    public class ObjectStatesDemo {      public static void main(String[] args) {            // Transient object state          Student student = new Student();          student.setId(101);          student.setName("Mukesh");          student.setRoll("10");          student.setDegree("B.E");          student.setPhone("9999");          // Transient object state          Session session = new AnnotationConfiguration().configure()                  .buildSessionFactory().openSession();          Transaction t = session.beginTransaction();          // Persistent object state          session.save(student);          t.commit();          // Persistent object state          session.close();          // Detached object state        }  } |

## Output :

Hibernate:

insert

into

STUDENT

(degree, name, phone, roll, id)

values

(?, ?, ?, ?, ?)

## In The Database :

mysql> select \* from student;

+-----+--------+--------+-------+------+

| id | degree | name | phone | roll |

+-----+--------+--------+-------+------+

| 101 | B.E | Mukesh | 9999 | 10 |

+-----+--------+--------+-------+------+

1 row in set (0.05 sec)

# **Hibernate get entity example – get vs load methods**

Learn to **get hibernate entity by id** using either session.load() or session.get() method. Learn the difference between get vs load method to fetch entity by id from database.

## 1. Hibernate load entity – session.load()

Hibernate’s Session interface provides several overloaded load() methods for loading entities from the 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, we need to cast the returned object to suitable type of class to further use it. It’s all what load() method need from us to work it correctly.

#### 1.1. Session load() method

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

|  |
| --- |
| Overloaded load() methods |
| public Object load(Class theClass, Serializable id) throws HibernateException  public Object load(String entityName, Serializable id) throws HibernateException  public void load(Object object, Serializable id) throws HibernateException |

1. First method need the class type which you would like to load along with unique ID.
2. Second 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.
3. Third 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 us, 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.

#### 1.2. Session load() method example

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();     }  } |

Program output.

|  |
| --- |
| Console |
| 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. Hibernate get entity by id – session.get()

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:

|  |
| --- |
| Overloaded get() methods |
| public Object get(Class clazz, Serializable id) throws HibernateException  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.

## 3. Difference between load() and get() methods

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.

The **difference between get and load** methods lies in return value when the identifier does not exist in database.

1. In case of get() method, we will get return value as NULL if identifier is absent.
2. But in case of load() method, we will get a runtime exception.

The exception in case of load method will look like this:

|  |
| --- |
| Runtime exception in load() method when ID does not exist |
| Exception in thread "main" org.hibernate.ObjectNotFoundException: No row with the given identifier exists:  [com.howtodoinjava.demo.entity.EmployeeEntity#23]  at org.hibernate.internal.SessionFactoryImpl$1$1.handleEntityNotFound(SessionFactoryImpl.java:253)  at org.hibernate.event.internal.DefaultLoadEventListener.load(DefaultLoadEventListener.java:219)  at org.hibernate.event.internal.DefaultLoadEventListener.proxyOrLoad(DefaultLoadEventListener.java:275)  at org.hibernate.event.internal.DefaultLoadEventListener.onLoad(DefaultLoadEventListener.java:151)  at org.hibernate.internal.SessionImpl.fireLoad(SessionImpl.java:1070)      at org.hibernate.internal.SessionImpl.load(SessionImpl.java:940) |

1. **What is Hibernate Framework?**

**Object-relational mapping** or ORM is the programming technique to map application domain model objects to the relational database tables. 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. We can use Hibernate persistence API for CRUD operations. Hibernate framework provide option to map plain old java objects to traditional database tables with the use of JPA annotations as well as XML based configuration.

Similarly hibernate configurations are flexible and can be done from XML configuration file as well as programmatically. For a quick overview of hibernate framework usage, you can go through [Hibernate Beginners Tutorial](http://www.journaldev.com/2882/hibernate-tutorial-for-beginners-using-xml-annotations-and-property-configurations).

1. **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.

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

Some of the important benefits of using hibernate framework are:

* 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. However, HQL is fully object-oriented and understands concepts like inheritance, polymorphism and association.
  4. Hibernate is an open source project from Red Hat Community and used worldwide. This makes it a better choice than others because learning curve is small and there are tons of online documentations and help is easily available in forums.
  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.
  8. For database vendor specific feature, hibernate is suitable because we can also execute native sql queries.

Overall hibernate is the best choice in current market for ORM tool, it contains all the features that you will ever need in an ORM tool.

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

Some of the important advantages of Hibernate framework over JDBC are:

* 1. Hibernate removes a lot of boiler-plate code that comes with JDBC API, the code looks 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. Read more at [JDBC Transaction Management](http://www.journaldev.com/2483/jdbc-transaction-management-and-savepoint-example-tutorial).
  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.

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

Some of the important interfaces of Hibernate framework are:

* 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.

1. **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.

1. **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.

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

Hibernate supports JPA annotations and it has some other annotations in org.hibernate.annotations package. Some of the important JPA and hibernate annotations used are:

* 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

Here are two classes showing usage of these annotations.

package com.journaldev.hibernate.model;

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 methods

}

package com.journaldev.hibernate.model;

import javax.persistence.Access;

import javax.persistence.AccessType;

import javax.persistence.Column;

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.Id;

import javax.persistence.OneToOne;

import javax.persistence.PrimaryKeyJoinColumn;

import javax.persistence.Table;

import org.hibernate.annotations.GenericGenerator;

import org.hibernate.annotations.Parameter;

@Entity

@Table(name = "ADDRESS")

@Access(value=AccessType.FIELD)

public class Address {

@Id

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

@GeneratedValue(generator = "gen")

@GenericGenerator(name = "gen", strategy = "foreign", parameters = { @Parameter(name = "property", value = "employee") })

private long id;

@Column(name = "address\_line1")

private String addressLine1;

@OneToOne

@PrimaryKeyJoinColumn

private Employee employee;

//getter setter methods

}

1. **What is Hibernate SessionFactory and how to configure it?**

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.

The internal state of a SessionFactory is immutable. Once it is created this internal state is set. This internal state includes all of the metadata about Object/Relational Mapping.

SessionFactory also provide methods to get the Class metadata and Statistics instance to get the stats of query executions, second level cache details etc.

1. **Hibernate SessionFactory is thread safe?**

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

1. **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.

1. **Hibernate Session is thread safe?**

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.

1. **What is difference between openSession and getCurrentSession?**

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. 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.

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

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.

There is another method openStatelessSession() that returns stateless session, for more details with examples please read [Hibernate openSession vs getCurrentSession](http://www.journaldev.com/3522/hibernate-sessionfactory-opensession-vs-getcurrentsession-vs-openstatelesssession).

1. **What is difference between Hibernate Session get() and load() method?**

Hibernate session comes with different methods to load data from database. get and load are most used methods, at first look they seems similar but there are some differences between them.

* 1. get() loads the data as soon as it’s called whereas load() returns a proxy object and loads data only when it’s actually required, so load() is better because it support lazy loading.
  2. Since load() throws exception when data is not found, we should use it only when we know data exists.
  3. We should use get() when we want to make sure data exists in the database.

For clarification regarding the differences, please read [Hibernate get vs load](http://www.journaldev.com/3472/hibernate-session-get-vs-load-difference-with-examples).

1. **What is hibernate caching? Explain Hibernate first level cache?**

As the name suggests, hibernate caches query data to make our application faster. Hibernate Cache can be very useful in gaining fast application performance if used correctly. The idea behind cache is to reduce the number of database queries, hence reducing the throughput time of the application.

Hibernate first level cache is associated with the Session object. Hibernate first level cache is enabled by default and there is no way to disable it. However hibernate provides methods through which we can delete selected objects from the cache or clear the cache completely.  
Any object cached in a session will not be visible to other sessions and when the session is closed, all the cached objects will also be lost.

# Hibernate HQL – Hibernate Query Language Examples –

By Lokesh Gupta | Filed Under: [Hibernate](https://howtodoinjava.com/hibernate/)

In this **HQL tutorial**, learn what is hibernate query language, hql syntax for various statements, named queries and native sql queries, associations and aggregations etc.

HQL is an object-oriented query language, similar to SQL, but instead of operating on tables and columns, HQL works with persistent objects and their properties. This is main difference between **hql vs sql**. HQL is a superset of the JPQL, the Java Persistence Query Language. A JPQL query is a valid HQL query, but not all HQL queries are valid JPQL queries.

HQL is a language with its own syntax and grammar. It is written as strings, like “from Product p“. HQL queries are translated by Hibernate into conventional SQL queries. [Hibernate](https://howtodoinjava.com/hibernate-tutorials/) also provides an API that allows us to directly issue SQL queries as well.

Please note that Hibernator’s query facilities do not allow you to alter the database structure. We can alter only data inside tables.

Table Of Contents

1. [HQL Syntax](https://howtodoinjava.com/hibernate/complete-hibernate-query-language-hql-tutorial/#basic_hql_systax)

1.1. [Update Operation](https://howtodoinjava.com/hibernate/complete-hibernate-query-language-hql-tutorial/#update_operation)

1.2. [Delete Operation](https://howtodoinjava.com/hibernate/complete-hibernate-query-language-hql-tutorial/#delete_operation)

1.3. [Insert Operation](https://howtodoinjava.com/hibernate/complete-hibernate-query-language-hql-tutorial/#insert_operation)

1.4. [Select Operation](https://howtodoinjava.com/hibernate/complete-hibernate-query-language-hql-tutorial/#select_operation)

2. [The from Clause and Aliases](https://howtodoinjava.com/hibernate/complete-hibernate-query-language-hql-tutorial/#from_clause)

3. [The select Clause and Projection](https://howtodoinjava.com/hibernate/complete-hibernate-query-language-hql-tutorial/#select_projection)

4. [Using Named Parameters](https://howtodoinjava.com/hibernate/complete-hibernate-query-language-hql-tutorial/#named_parameters)

5. [Paging Through the Result Set](https://howtodoinjava.com/hibernate/complete-hibernate-query-language-hql-tutorial/#paging_resultsets)

6. [Obtaining a Unique Result](https://howtodoinjava.com/hibernate/complete-hibernate-query-language-hql-tutorial/#unique_result)

7. [Sorting Results with the order by Clause](https://howtodoinjava.com/hibernate/complete-hibernate-query-language-hql-tutorial/#sorting_through_order_by)

8. [Associations](https://howtodoinjava.com/hibernate/complete-hibernate-query-language-hql-tutorial/#associations)

9. [Aggregate Methods](https://howtodoinjava.com/hibernate/complete-hibernate-query-language-hql-tutorial/#aggregate_methods)

10. [Named Queries](https://howtodoinjava.com/hibernate/complete-hibernate-query-language-hql-tutorial/#named_queries)

11. [Using Native SQL](https://howtodoinjava.com/hibernate/complete-hibernate-query-language-hql-tutorial/#native_sql)

12. [Enable Logging and Commenting](https://howtodoinjava.com/hibernate/complete-hibernate-query-language-hql-tutorial/#logging_commenting)

Let’s discuss each and every item in more detail starting from basic stuff to more complex concepts.

## 1. HQL Syntax

HQL syntax is defined as an [**ANTLR**](https://en.wikipedia.org/wiki/ANTLR) grammar. The grammar files are included in the grammar directory of the Hibernate core download. (ANTLR is a tool for building language parsers). Lets outline the syntax for the four fundamental CRUD operations here:

#### 1.1. HQL Update Statement

UPDATE alters the details of existing objects in the database. In-memory entities, managed or not, will not be updated to reflect changes resulting from issuing UPDATE statements. Here’s the syntax of the UPDATE statement:

|  |
| --- |
| hql update statement syntax |
| UPDATE [VERSIONED]     [FROM] path [[AS] alias] [, ...]     SET property = value [, ...]     [WHERE logicalExpression] |

* path – fully qualified name of the entity or entities
* alias – used to abbreviate references to specific entities or their properties, and must be used when property names in the query would otherwise be ambiguous.
* VERSIONED – means that the update will update time stamps, if any, that are part of the entity being updated.
* property – names of properties of entities listed in the FROM path.
* logicalExpression – a where clause.

An example of the update in action might look like this. In this example, we are updating employee data with **hql update query multiple columns**.

|  |
| --- |
| hql update statement example |
| Query query=session.createQuery("update Employee set age=:age where name=:name");  query.setInteger("age", 32);  query.setString("name", "Lokesh Gupta");  int modifications=query.executeUpdate(); |

#### 1.2. HQL Delete Statement

DELETE removes the details of existing objects from the database. In-memory entities will not be updated to reflect changes resulting from DELETE statements. This also means that Hibernate’s cascade rules will not be followed for deletions carried out using HQL. However, if you have specified cascading deletes at the database level (either directly or through Hibernate, using the @OnDeleteannotation), the database will still remove the child rows.

Here’s the syntax of the DELETE statement:

|  |
| --- |
| hql delete statement syntax |
| DELETE     [FROM] path [[AS] alias]     [WHERE logicalExpression] |

In practice, deletes might look like this:

|  |
| --- |
| hql delete statement example |
| Query query=session.createQuery("delete from Account where accountstatus=:status");  query.setString("status", "purged");  int rowsDeleted=query.executeUpdate(); |

#### 1.3. HQL Insert Statement

An HQL INSERT **cannot be used to directly insert arbitrary entities**—it can only be used to insert entities constructed from information obtained from SELECT queries (unlike ordinary SQL, in which an INSERT command can be used to insert arbitrary data into a table, as well as insert values selected from other tables).

Here’s the syntax of the INSERT statement:

|  |
| --- |
| hql insert statement example |
| INSERT     INTO path ( property [, ...])     select |

The name of an entity is path. The property names are the names of properties of entities listed in the FROM path of the incorporated SELECT query. The select query is an HQL SELECT query (as described in the next section).

As this HQL statement can only use data provided by an HQL select, its application can be limited. An example of copying users to a purged table before actually purging them might look like this:

|  |
| --- |
| hql insert statement example |
| Query query=session.createQuery("insert into purged\_accounts(id, code, status) "+      "select id, code, status from account where status=:status");  query.setString("status", "purged");  int rowsCopied=query.executeUpdate(); |

#### 1.4. HQL Select Statement

An HQL SELECT is used to query the database for classes and their properties. Here’s the syntax of the SELECT statement:

|  |
| --- |
| hql select statement example |
| [SELECT [DISTINCT] property [, ...]]     FROM path [[AS] alias] [, ...] [FETCH ALL PROPERTIES]     WHERE logicalExpression     GROUP BY property [, ...]     HAVING logicalExpression     ORDER BY property [ASC | DESC] [, ...] |

The fully qualified name of an entity is path. The alias names may be used to abbreviate references to specific entities or their properties, and must be used when property names used in the query would otherwise be ambiguous.

The property names are the names of properties of entities listed in the ***FROM*** path.

If ***FETCH ALL PROPERTIES*** is used, then lazy loading semantics will be ignored, and all the immediate properties of the retrieved object(s) will be actively loaded (this does not apply recursively).

WHERE is used to create **hql select query with where clause**.

When the properties listed consist only of the names of aliases in the FROM clause, the SELECT clause can be omitted in HQL. If we are using the JPA with JPQL, one of the differences between HQL and JPQL is that the SELECT clause is required in JPQL.

## 2. HQL – from clause and aliases

The most important feature in HQL to note is the **alias**. Hibernate allows us to assign aliases to the classes in our query with the as clause. Use the aliases to refer back to the class inside the query.

Take for example:

|  |
| --- |
| hql alias example |
| from Product as p    //or    from Product as product |

The 'as' keyword is optional. We can also specify the alias directly after the class name as follows:

|  |
| --- |
| hql alias example |
| from Product product |

If we need to fully qualify a class name in HQL, just specify the package and class name. Hibernate will take care of most of this behind the scenes, so we really need this only if we have classes with duplicate names in our application. If we have to do this in Hibernate, use syntax such as the following:

|  |
| --- |
| from com.howtodoinjava.geo.usa.Product |

The from clause is very basic and useful for working directly with objects. However, if you want to work with the object’s properties without loading the full objects into memory, you must use the select clause as explained in next section.

## 3. HQL select clause and projection

The select clause provides more control over the result set than the from clause. If you want to obtain the properties of objects in the result set, use the select clause. For instance, we could run a projection query on the products in the database that only returned the names, instead of loading the full object into memory, as follows:

|  |
| --- |
| select product.name from Product product |

The result set for this query will contain a List of Java String objects. Additionally, we can retrieve the prices and the names for each product in the database, like so:

|  |
| --- |
| select product.name, product.price from Product product |

If you’re only interested in a few properties, this approach can allow you to reduce network traffic to the database server and save memory on the application’s machine.

## 4. HQL Named Parameters

Hibernate supports named parameters in its HQL queries. This makes writing queries that accept input from the user easy—and you do not have to defend against SQL injection attacks.

When using JDBC query parameters, any time you add, change, or delete parts of the SQL statement, you need to update your Java code that sets its parameters, because the parameters are indexed based on the order in which they appear in the statement. Hibernate lets you provide names for the parameters in the HQL query, so you do not have to worry about accidentally moving parameters around in the query.

The simplest example of named parameters uses regular SQL types for the parameters:

|  |
| --- |
| String hql = "from Product where price > :price";  Query query = session.createQuery(hql);  query.setDouble("price",25.0);  List results = query.list(); |

## 5. HQL – Paging Through the ResultSet

Pagination through the result set of a database query is a very common application pattern. Typically, you would use pagination for a web application that returned a large set of data for a query. The web application would page through the database query result set to build the appropriate page for the user. The application would be very slow if the web application loaded all of the data into memory for each user. Instead, you can page through the result set and retrieve the results you are going to display one chunk at a time.

There are two methods on the Query interface for paging: 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 only retrieve a fixed number of objects with the setMaxResults() method. Your HQL is unchanged—you need only to modify the Java code that executes the query.

|  |
| --- |
| Query query = session.createQuery("from Product");  query.setFirstResult(1);  query.setMaxResults(2);  List results = query.list();  displayProductsList(results); |

If you turn on SQL logging, you can see which SQL commands Hibernate uses for pagination. For the open-source HSQLDB database, Hibernate uses top and limit. Microsoft SQL Server does not support the limit command, so Hibernate uses only the top command. If your application is having performance problems with pagination, this can be very helpful for debugging.

If you only have one result in your HQL result set, Hibernate has a shortcut method for obtaining just that object as discussed next.

## 6. HQL – Get a Unique Result

HQL’s Query interface provides a uniqueResult() method for obtaining just one object from an HQL query. Although your query may yield only one object, you may also use the uniqueResult() method with other result sets if you limit the results to just the first result. You could use the setMaxResults()method discussed in the previous section.

The uniqueResult() method on the Query object returns a single object, or null if there are zero results. If there is more than one result, then the uniqueResult() method throws a NonUniqueResultException.

|  |
| --- |
| String hql = "from Product where price>25.0";  Query query = session.createQuery(hql);  query.setMaxResults(1);  Product product = (Product) query.uniqueResult(); |

## 7. HQL – Sorting Results with the ‘order by’ clause

To sort your HQL query’s results, you will need to use the **order by clause**. You can order the results by any property on the objects in the result set: either ascending (asc) or descending (desc). You can use ordering on more than one property in the query, if you need to. A typical HQL query for sorting results looks like this:

|  |
| --- |
| from Product p where p.price>25.0 order by p.price desc |

If you wanted to sort by more than one property, you would just add the additional properties to the end of the order by clause, separated by commas. For instance, you could sort by product price and the supplier’s name, as follows:

|  |
| --- |
| from Product p order by p.supplier.name asc, p.price asc  [/ql]    <a name="associations"></a>  <h2>8. HQL associations</h2>    Associations allow you to use <strong>more than one class in an HQL query</strong>, just asSQL allows you to use joins between tables in a relational database. You add an association toan HQL query with the join clause. Hibernate supports five different types of joins: <strong>inner join, cross join, left outer join, right outer join, and full outerjoin</strong>.    If you use cross join, just specify both classes in the from clause (from Product p, Supplier s). For the other joins, use a join clause after the from clause. Specify the type of join, the object property to join on, and an alias for the other class.    You can use inner join to obtain the supplier for each product, and then retrieve the suppliername, product name, and product price, as so:      select s.name, p.name, p.price from Product p inner join p.supplier as s |

You can retrieve the objects using similar syntax:

|  |
| --- |
| from Product p inner join p.supplier as s |

## 9.HQL Aggregate Methods

HQL supports a range of aggregate methods, similar to SQL. They work the same way in HQL as in SQL, so you do not have to learn any specific Hibernate terminology. The difference is that in HQL, aggregate methods apply to the properties of persistent objects. You may use the count(\*) syntax to count all the objects in the result set, or **count(product.name)** to count the number of objects in the result set with a name property. Here is an example using the count(\*) method to count all products:

|  |
| --- |
| select count(\*) from Product product |

The aggregate functions available through HQL include the following:

1. avg(property name): The average of a property’s value
2. count(property name or \*): The number of times a property occurs in the results
3. max(property name): The maximum value of the property values
4. min(property name): The minimum value of the property values
5. sum(property name): The sum total of the property values

## 10. HQL Named Queries

Named queries are created via class-level annotations on entities; normally, the queries apply to the entity in whose source file they occur, but there’s no absolute requirement for this to be true.

Named queries are created with the @NamedQueries annotation, which contains an array of @NamedQuery sets; each has a query and a name.

An example of named queries may look like this:

|  |
| --- |
| @NamedQueries({          @NamedQuery(name = "supplier.findAll", query = "from Supplier s"),          @NamedQuery(name = "supplier.findByName",                  query = "from Supplier s where s.name=:name"),  }) |

Executing above named query is even simpler.

|  |
| --- |
| Query query = session.getNamedQuery("supplier.findAll");  List<Supplier> suppliers = query.list(); |

Read More – [Hibernate named query tutorial](https://howtodoinjava.com/hibernate/hibernate-named-query-tutorial/)

## 11. HQL – Native SQL

Although you should probably use HQL whenever possible, Hibernate does provide a way to use native SQL statements directly through Hibernate. One reason to use native SQL is that your database supports some special features through its dialect of SQL that are not supported in HQL. Another reason is that you may want to call stored procedures from your Hibernate application.

You can modify your SQL statements to make them work with Hibernate’s ORM layer. You do need to modify your SQL to include Hibernate aliases that correspond to objects or object properties. You can specify all properties on an object with {objectname.\*}, or you can specify the aliases directly with {objectname.property}.

Hibernate uses the mappings to translate your object property names into their underlying SQL columns. This may not be the exact way you expect Hibernate to work, so be aware that you do need to modify your SQL statements for full ORM support. You will especially run into problems with native SQL on classes with subclasses—be sure you understand how you mapped the inheritance across either a single table or multiple tables, so that you select the right properties off the table.

Underlying Hibernate’s native SQL support is the org.hibernate.SQLQuery interface, which extends the org.hibernate.Query interface. Your application will create a native SQL query from the session with the createSQLQuery() method on the Session interface.

|  |
| --- |
| public SQLQuery createSQLQuery(String queryString) throws HibernateException |

After you pass a string containing the SQL query to the createSQLQuery() method, you should associate the SQL result with an existing Hibernate entity, a join, or a scalar result. The SQLQueryinterface has addEntity(), addJoin(), and addScalar() methods.

#### 11.1. Hibernate sql query example

Using native SQL with scalar results is the simplest way to get started with native SQL. Sample Java code looks like this:

|  |
| --- |
| String sql = "select avg(product.price) as avgPrice from Product product";  SQLQuery query = session.createSQLQuery(sql);  query.addScalar("avgPrice",Hibernate.DOUBLE);  List results = query.list(); |

A bit more complicated than the previous example is the **native SQL** that returns a result set of objects. In this case, we will need to map an entity to the SQL query.

|  |
| --- |
| String sql = "select {supplier.\*} from Supplier supplier";  SQLQuery query = session.createSQLQuery(sql);  query.addEntity("supplier", Supplier.class);  List results = query.list();    //Hibernate modifies the SQL and executes the following command against the database:    select Supplier.id as id0\_, Supplier.name as name2\_0\_ from Supplier supplier |

## 12. HQL – Enable Logs and Comments

Hibernate can output the underlying SQL behind your HQL queries into your application’s log file. This is especially useful if the HQL query does not give the results you expect, or if the query takes longer than you wanted. This is not a feature you will have to use frequently, but it is useful should you have to turn to your database administrators for help in tuning your Hibernate application.

#### 12.1. HQL Logs

The easiest way to see the SQL for a Hibernate HQL query is to enable SQL output in the logs with the “**show\_sql**” property. Set this property to true in your **hibernate.cfg.xml** configuration file and Hibernate will output the SQL into the logs. When you look in your application’s output for the Hibernate SQL statements, they will be prefixed with “Hibernate:”.

If you turn your log4j logging up to debug for the Hibernate classes, you will see SQL statements in your log files, along with lots of information about how Hibernate parsed your HQL query and translated it into SQL.

#### 12.2. HQL Comments

Tracing your HQL statements through to the generated SQL can be difficult, so Hibernate provides a commenting facility on the Query object that lets you apply a comment to a specific query. The Queryinterface has a setComment() method that takes a String object as an argument, as follows:

|  |
| --- |
| public Query setComment(String comment) |

Hibernate will not add comments to your SQL statements without some additional configuration, even if you use the setComment() method. You will also need to set a Hibernate property, **hibernate.use\_sql\_comments**, to true in your Hibernate configuration.

If you set this property but do not set a comment on the query programatically, Hibernate will include the HQL used to generate the SQL call in the comment. I find this to be very useful for debugging HQL.

# Hibernate criteria queries examples

By Lokesh Gupta | Filed Under: [Hibernate](https://howtodoinjava.com/hibernate/)

[Hibernate](https://howtodoinjava.com/hibernate-tutorials/) 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. **hibernate criteria queries**. 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

[1. Hibernate criteria example](https://howtodoinjava.com/hibernate/hibernate-criteria-queries-tutorial/#basic_usage)

[2. Hibernate criteria - using Restrictions](https://howtodoinjava.com/hibernate/hibernate-criteria-queries-tutorial/#restrictions)

[3. Hibernate criteria - paging through the result set](https://howtodoinjava.com/hibernate/hibernate-criteria-queries-tutorial/#paging)

[4. Hibernate criteria - Obtain unique result](https://howtodoinjava.com/hibernate/hibernate-criteria-queries-tutorial/#unique_result)

[5. Hibernate criteria - obtaining distinct results](https://howtodoinjava.com/hibernate/hibernate-criteria-queries-tutorial/#distinct)

[6. Hibernate criteria - sort query results](https://howtodoinjava.com/hibernate/hibernate-criteria-queries-tutorial/#sorting)

[7. Hibernate criteria - perform associations (joins)](https://howtodoinjava.com/hibernate/hibernate-criteria-queries-tutorial/#associations)

[8. Hibernate criteria - add projections](https://howtodoinjava.com/hibernate/hibernate-criteria-queries-tutorial/#projections_aggregates)

[9. Hibernate criteria - query by example (QBE)](https://howtodoinjava.com/hibernate/hibernate-criteria-queries-tutorial/#query_by_example)

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

## 1. Hibernate criteria 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 overloaded 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 criteria example, we will add constraints to our criteria queries so we can whittle down the result set.

## 2. Hibernate criteria – using Restrictions

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.

#### 2.1. 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”.

#### 2.2. 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.

#### 2.3. 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

#### 2.4. 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(); |

#### 2.5. 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(); |

#### 2.6. 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(); |

#### 2.7. 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(); |

#### 2.8. 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(); |

## 3. Hibernate criteria – paging through the result set

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.

## 4. Hibernate criteria – obtain 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.

## 5. Hibernate criteria – obtain 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.

## 6. Hibernate criteria – sort query 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**.

## 7. Hibernate criteria – perform 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(); |

## 8. Hibernate criteria – add 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 Projectioninstances. 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.

#### 8.1. 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

#### 8.2. 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(); |

#### 8.3. 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(); |

## 9. Hibernate criteria – 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(); |

## 10. 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.