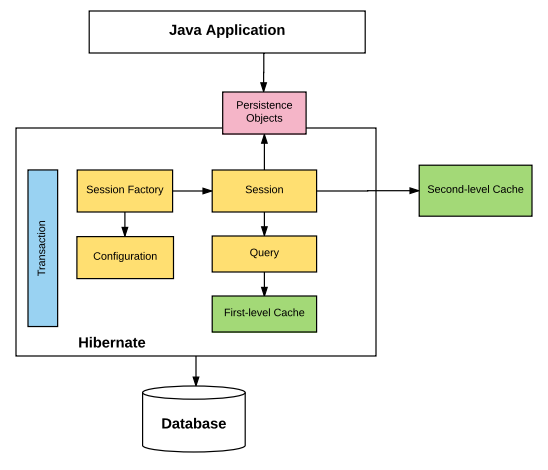
Hibernate is an open source **Java persistence** framework project. It performs powerful object-relational mapping and query databases using HQL and SQL. Hibernate is a great tool for **ORM mappings** in Java. It can cut down a lot of complexity and thus defects as well from your application, which may otherwise find a way to exist. This is specially boon for developers with limited knowledge of SQL.

**Hibernate Architecture**

The following diagram summarizes the main building blocks in hibernate 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.

## Salient features of the Hibernate framework

#### Object/Relational Mapping

Hibernate, as an ORM framework, allows the mapping of the Java domain object with database tables and vice versa. As a result, business logic is able to access and manipulate database entities via Java objects. It helps to speed up the overall development process by taking care of aspects such as transaction management, automatic primary key generation, managing database connections and related implementations, and so on.

#### JPA provider

Hibernate does support the [**Java Persistence API**](https://docs.oracle.com/javaee/6/tutorial/doc/bnbpz.html) **(JPA)** specification. JPA is a set of specifications for accessing, persisting, and managing data between Java objects and relational database entities.

#### Idiomatic persistence

Any class that follows [object-oriented principles](https://howtodoinjava.com/oops/object-oriented-principles/) such as inheritance, polymorphism, and so on, can be used as a persistent class.

#### High performance and scalability

Hibernate supports techniques such as different fetching strategies, lazy initialization, optimistic locking, and so on, to achieve high performance, and it scales well in any environment.

#### Easy to maintain

Hibernate is easier to maintain as it requires no special database tables or fields. It generates SQL at system initialization time. It is much quicker and easier to maintain compared to JDBC.

# Hibernate HQL – Hibernate Query Language Examples –

In this **HQL notes**, 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", "Smita 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 @OnDelete annotation), 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 as SQL allows you to use joins between tables in a relational database. You add an association to an 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 outer join</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 supplier name, 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 SQLQuery interface 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 Query interface 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.

Use commenting to identify the SQL output in your application’s logs if SQL logging is enabled.

# Hibernate criteria queries examples

[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 NonUniqueResultException exception.

## 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 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.

#### 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.