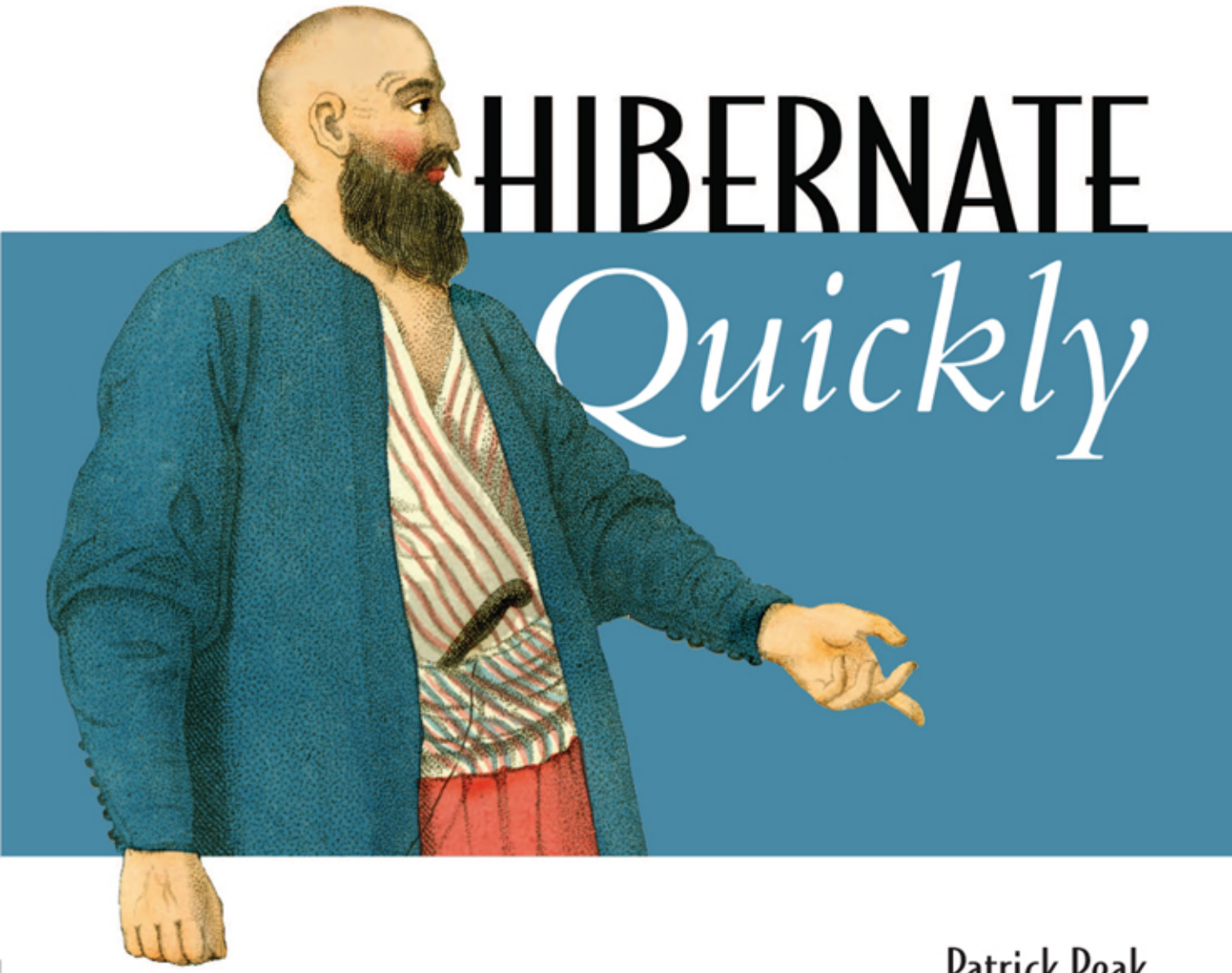
Sample Chapter



HIBERNATE *Quickly*

Patrick Peak
Nick Heudecker

MANNING

Hibernate Quickly

by Patrick Peak

and

Nick Heudecker

Chapter 6

Brief contents

6

Querying persistent objects

This chapter covers

- *Querying persistent objects using Hibernate*
- *The Hibernate Query Language*

With a solid grasp of Hibernate basics, we need to move on to querying our persistent objects. Instead of SQL, Hibernate has its own, object-oriented (OO) query language called Hibernate Query Language (HQL). HQL is intentionally similar to SQL so that it leverages existing developer knowledge and thus minimizes the learning curve. It supports the commonly used SQL features, wrapped into an OO query language. In some ways, HQL is easier to write than SQL because of its OO foundation.

Why do we need HQL? While SQL is more common and has been standardized, vendor-dependent features limit the portability of SQL statements between different databases. HQL provides an abstraction between the application and the database, and so improves portability. Another problem with SQL is that it is designed to work with relational tables, not objects. HQL is optimized to query object graphs.

This chapter introduces you to HQL gradually and quickly moves on to more complicated features and queries. First we'll cover the major

concepts important to using HQL, such as executing queries using a few different classes. After the introductory material is covered, we'll spend the rest of the chapter with HQL examples. If you have a solid grasp of SQL, you shouldn't have any problem picking up the key concepts.

Chapter goals

We have three primary goals in this chapter:

- Exploring the basics of HQL, including two query mechanisms
- Identifying variations in HQL queries, including named and positional parameters
- Understanding how to query objects, associations, and collections

Assumptions

Since HQL is based on SQL, we anticipate that you

- Understand SQL basics, including knowledge of joins, subselects, and functions.
- Have a firm grasp of JDBC, including the `PreparedStatement` and `ResultSet` interfaces.

6.1 Using HQL

Hibernate queries are structured similar to their SQL counterparts, with SELECT, FROM, and WHERE clauses. HQL also supports subselects if they are supported by the underlying database. Let's jump in with the most basic query we can create:

```
from Event
```

This query will return all of the `Event` instances in the database, as well as the associated objects and non-lazy collections. (You'll recall from chapter 5 that, by default, persistent collections are populated only when initially accessed.) The first thing you probably noticed was the lack of the SELECT clause in front of the FROM clause. Because we

want to return complete objects, the SELECT clause is implied and doesn't need to be explicitly stated.

How can we execute this query? Two methods are provided in the Hibernate API to execute queries. The `Session` interface has three `find(…)` methods that can be used for simple queries. The `Query` interface can be used for more complex queries.

6.1.1 session.find(...)

In Hibernate 2, the `Session` interface has three overloaded `find(…)` methods, two of which support parameter binding. Each of the methods returns a `java.util.List` with the query results. For instance, let's execute our earlier query:

```
List results = session.find("from Event");
```

The `Session` interface also has a set of methods, named `iterate(…)`, which are similar to the `find(…)` methods. Although they appear to be the same, each of methods functions differently. The find methods return all of the query results in a `List`, which is what you'd expect. The objects in the list are populated when the query is executed. However, the iterate methods first select all of the object ids matching a query and populate the objects on demand as they are retrieved from the `Iterator`. Here's an example:

```
Iterator results = session.iterate("from Event");
while (results.hasNext()) {
    Event myEvent = (Event) results.next();
    // …
}
```

When the `Iterator` is returned, only the id values for the `Event` instances have been retrieved. Calling `results.next()` causes the next `Event` instance to be retrieved from the database. The iterate methods are typically less efficient than their find counterparts, except when dealing with objects stored in a second-level cache.

Hibernate stores objects in a second-level cache based on the object's class type and id. The iterate methods can be more efficient if the object is already cached, since only the matching object ids are returned on the initial query and the remainder of the object is retrieved from the cache.

The find methods in the `Session` interface are ideal for simple queries. However, most applications typically require at least a handful of complex queries. The `Query` interface provides a rich interface for retrieving persistent objects with complicated queries. If you're using Hibernate 3, you must use the `Query` interface since the find methods in the `Session` interface have been deprecated (although they have been moved and are still available in the `org.hibernate.class.Session` subinterface). Hibernate 3 applications should use `createQuery()` and get `NamedQuery()` for all query execution.

6.1.2 The Query interface

Instances of the `Query` interface are created by the `Session`. The `Query` interface gives you more control over the returned objects, such as limiting the number of returned objects, setting a timeout for the query, and creating scrollable result sets. Let's execute our previous query using the `Query` interface:

```
Query q = session.createQuery("from Event");
List results = q.list();
```

If you want to set bounds on the number of `Event` objects to return, set the `maxResults` property:

```
Query q = session.createQuery("from Event");
q.setMaxResults(15);
List results = q.list();
```

The `Query` interface also provides an `iterate()` method, which behaves identically to `Session.iterate(…)`. Another feature of the `Query`

interface is the ability to return query results as a `ScrollableResults` object. The `ScrollableResults` object allows you to move through the returned objects arbitrarily, and is typically useful for returning paged collections of objects, commonly found in web applications.

Of course, our static queries aren't very useful in real applications. Real applications need to populate query parameters dynamically. JDBC's `PreparedStatement` interface supports setting positional query parameters dynamically, but populating queries can be cumbersome.

Developers must know the type of each parameter in order to call the correct setter method in the interface. They also have to keep track of which positional parameter they are setting. Hibernate expands and improves on this notion by providing both positional and named query parameters.

Positional parameters

Positional parameters in Hibernate are very similar to their `PreparedStatement` counterparts. The only significant difference is that the position index starts at 0 instead of 1, as in the `PreparedStatement`.

Suppose you want to return all of the `Event` instances with a certain name. Using a positional parameter, your code would look like

```
Query q = session.createQuery("from Event where name = ? ");
q.setParameter(0, "Opening Plenary");
List results = q.list();
```

Note that you didn't need to set the type of parameter; the `Query` object will attempt to determine the type using reflection. It's also possible to set a parameter to a specific type using the `org.hibernate.Hibernate` class:

```
q.setParameter(0, "Opening Plenary", Hibernate.STRING);
```

Named parameters are a more interesting, and more powerful, way to populate queries. Rather than using question marks for parameter placement, you can use distinctive names.

Named parameters

The easiest way to explain named parameters is with an example. Here's our earlier query with a named parameter:

```
from Event where name = :name
```

Instead of the ? to denote a parameter, you can use `:name` to populate the query:

```
Query q = session.createQuery("from Event where name = :name");
q.setParameter("name", "Opening Plenary");
List results = q.list();
```

With named parameters, you don't need to know the index position of each parameter. Named parameters may seem like a minor feature, but they can save time when populating a query—instead of counting question marks and making sure you're populating the query correctly, you simply match the named parameters with your code.

If your query has a named parameter that occurs more than once, it will be set in the query each time. For instance, if a query has the parameter `startDate` twice, it will be set to the same value:

```
Query q = session.createQuery("from Event where "+
    "startDate = :startDate or endDate < :startDate");
q.setParameter("startDate", eventStartDate);
List results = q.list();
```

We've covered the two styles of query parameters supported by Hibernate. For the purpose of our examples, we've been displaying the queries as hardcoded in application code. Anyone who's built an application will know that embedding your queries can quickly create a maintenance nightmare. Ideally, you would store the queries in a text file, and the most natural place to do that with Hibernate is in the mapping definition file.

Named queries

Named queries, not to be confused with named parameters, are queries that are embedded in the XML mapping definition. Typically, you put all of the queries for a given object into the same file. Centralizing your queries in this fashion makes maintenance quite a bit easier. Named queries are placed at the bottom of the mapping definition files:

```
<query name="Event.byName">
<![CDATA[from Event where name=?]]>
</query>
```

Note that here you wrap the actual query in a CDATA block to ensure that the XML parser skips the query text. This is necessary since some symbols, such as < and >, can cause XML parsing errors.

There is no limitation on the number of query elements you can have in a mapping file; just be sure that all of the query names are unique. You may name the queries anything you would like, although we have found that prefixing the name of the persistent class is helpful.

Accessing named queries is simple:

```
Query q = session.getNamedQuery("Event.byName");
…
List results = q.list();
```

You should take advantage of named queries when creating your mapping definitions. If you add or change a property name, you can also update all of the affected queries at the same time.

There is certainly no rule about storing your HQL queries in the mapping definition. You can just as easily put the queries into a resource bundle and provide your own mechanism to pass them to the query interfaces.

Now that you know how to query and some of the basics, what happens when you execute a query, using either the `Session` or `Query` interface?

First, the Hibernate runtime compiles the query into SQL, taking into account the database-specific SQL dialect. Next, a `PreparedStatement` is created and any query parameters are set. Finally, the runtime executes the `PreparedStatement` and converts the `ResultSet` into instances of the persistent objects. (It's a little more complicated than the three-sentence description, but I hope you get the idea.) When you retrieve persistent objects, you also need to retrieve the associated objects, such as many-to-ones and child collections.

6.1.3 Outer joins and HQL

Using SQL, the most natural way to retrieve data from multiple tables with a single query is to use a join. Joins work by linking associated tables using foreign keys. Hibernate uses an outer join to retrieve associated objects. When the HQL is compiled to SQL, Hibernate creates the outer join statements for the associated objects (assuming you've enabled outer joins—see chapter 3 for an explanation of the `max_fetch_depth` property). This results in one query returning all of the data necessary to reconstitute an `Event` instance. All of this happens behind the scenes in the Hibernate runtime. Compare this with writing the following SQL:

```
select e.*, l.* from events as e
    left outer join locations  as l on e.location_id = l.id
```

Clearly, HQL is much more concise than SQL. It's also possible to disable outer join fetching for a specific association by setting the `fetch` attribute in the mapping definition, as shown here:

```
<many-to-one name="location" fetch="select"/>
```

Alternatively, outer join fetching can be disabled globally by setting the `max_fetch_depth` property to `0` in the hibernate.cfg.xml file:

```
<property name="max_fetch_depth">0</property>
```

You will typically leave outer join fetching enabled, as it can greatly improve performance by reducing the number of trips to the database

to retrieve an object. The `max_fetch_depth` property is just one configuration parameter that can impact queries. We'll look at two more: `show_sql` and `query_substitutions`.

6.1.4 Show SQL

While debugging HQL statements, you may find it useful to see the generated SQL to make sure the query is doing what you expect. Setting the `show_sql` property to `true` will result in the generated SQL being output to the console, or whatever you have `System.out` set to. Set the property in your hibernate.cfg.xml file:

```
<property name="show_sql">true</property>
```

You will want to turn off SQL output when deploying to production, especially in an application server. Application servers typically set `System.out` to a log file, and the SQL output can be overwhelming.

6.1.5 Query substitutions

Hibernate supports many different databases and SQL dialects, each with different names for similar functions. Using query substitutions, you can normalize function names and literal values, which simplifies porting to different databases. Query substitutions can be a confusing, so let's look at a few examples.

First, to configure query substitutions in the configuration file, use the following:

```
<property name="query.substitutions">
    convert CONV, true 1, false 0
</property>
```

This configuration setting performs three substitutions:

- The `CONV` function is now aliased to `convert`.
- Boolean `true` values are replaced with `1`.
- Boolean `false values` are replaced with `0`.

Query substitutions occur when the HQL is compiled into SQL. The substitutions for boolean values are useful if your database does not support boolean data types. This allows you to use `true` and `false` in your queries, which is clearer than using `1` and `0`.

By substituting `convert` for the name of the `CONV` function, you can make your HQL statements more portable. For example, MySQL names the function `CONV`, while Oracle names the same function `CONVERT`. If you port your application to Oracle, you only need to update the query substitution property instead of the HQL.

6.1.6 Query parser

Hibernate 3 introduces a new HQL abstract syntax tree (AST) query parser, which replaces the classic parser found in earlier releases. While the parser in use doesn't make much difference to you, we mention it because in some cases, particularly when migrating an application from Hibernate 2 to 3, you may want to use the classic parser. You'll likely want to use the classic parser if the AST parser complains about your existing HQL.

To switch from the AST parser (which is the default) to the classic parser, set the following property in your hibernate.properties file:

```
hibernate.query.factory_class=
    ➥ org.hibernate.hql.classic.ClassicQueryTranslatorFactory
```

If you're configuring Hibernate using the `hibernate.cfg.xml` property, use this:

```
<property name="query.factory_class">
org.hibernate.hql.classic.ClassicQueryTranslatorFactory
</property>
```

Ideally, you'll use the newer AST parser. When the `SessionFactory` is created, the AST parser validates all of the named HQL queries found

in your mapping files, which can save you a lot of time when you're testing your application.

We've covered the introductory information necessary to use HQL in your applications. The remainder of the chapter will be spent discussing the features of the query language.

6.2 Querying objects with HQL

With a solid foundation in executing queries, you can concentrate on exploring the query language itself. If you have experience with SQL, you shouldn't have a problem getting a firm grasp of HQL.

This section doesn't have much Java code; instead, we provide a number of example queries and explanations. Although we've presented the occasional HQL statement at various points earlier in the book, this section examines features of HQL that we haven't used.

6.2.1 The FROM clause

The FROM clause allows you to specify the objects that will be queried. It also lets you create aliases for object names. Suppose you want to query all `Event` instances matching a given name. Your resulting query would be as follows:

```
from Event e where e.name='Opening Plenary'
```

This new query introduces a shorthand name, or alias, for the `Event` instance: `e`. This shorthand name can be used just like its SQL counterpart, except here you're using it to identify objects instead of tables. Unlike SQL, HQL does not allow the `as` keyword when defining the alias. For instance:

```
from Event as e where e.name='Opening Plenary'
```

The `as` in the previous query will cause an `org.hibernate.QueryException` when you attempt to execute the query. Since `as` is implied, there is no need to insert it in the query. You're also querying a property of the `Event`: the name. When querying properties, use the `JavaBean`

property name instead of the column name in the table. You shouldn't be concerned with the underlying relational tables and columns when using HQL, but instead focus on the properties of the domain objects.

You will typically have one object in the FROM clause of the query, as in our examples to this point. Querying on one object type simplifies the results, since you're only going to get a `List` containing instances of the queried object. What happens if you need to query multiple associated objects? You could have multiple objects in the FROM clause, such as

```
from Event e, Attendee a where …
```

How do you know what object type the result list will contain? The result list will contain a Cartesian product of the queried objects, which probably isn't what you want. To query on associated objects, you'll need to join them in the FROM clause.

6.2.2 Joins

You're probably familiar with SQL joins, which return data from multiple tables with a single query. You can think of HQL joins in a similar way, only you're joining object properties and associations instead of tables. If we want to return all `Events` that a specific `Attendee` is going to be attending, join the `attendee` property to the `Event` in the query:

```
from Event e join e.attendees a where a.id=314
```

You can join all associations (many-to-one and one-to-one), as well as collections, to the query's base object. (We refer to the base object in a query as the object listed in the FROM clause. In this case, the base object is the `Event`.) As the previous query shows, you can also assign an alias to joined associations and query on properties in the joined object. The naming convention for HQL aliases is to use a lowercase word, similar to the Java variable naming convention.

Types of joins

HQL has different types of joins, all but one of them taken from SQL. We summarize the join types in table 6.1.

Table 6.1 Join types

Join Type	Rule
inner join	Unmatched objects on either side of the join are discarded.
left [outer] join	All objects from the left side of the join are returned. If the object on the left side of the join has no matching object on the right side of the join, it is still returned.
right [outer] join	All objects from the right side of the join are returned. If the object on the right side of the join has no matching object on the left side of the join, it is still returned.
full join	All objects from either side of the join are returned, regardless of matching objects on the opposite side of the join.
inner join fetch	Used to retrieve an associated object or a collection of objects regardless of the `outer-join` or `lazy` property on the association. This join does not have a SQL counterpart.

Unless you specify `left`, `right`, or `full` as the prefix to the join statement, the default is to use inner joins. All of the joins in table 6.1 behave like their SQL counterparts, except for `inner join fetch`. Joining a lazy collection with `inner join fetch` will cause the collection to be returned as populated. For example:

```
from Event e inner join fetch e.speakers
```

returns all of the `Event` instances with populated collections of `Speakers`. Let's look at joining an object associated to the base object as a many-to-one:

```
from Event e join e.location l where l.name = :name
```

Joining the `Location` instance to the `Event` allows querying on the `Location` properties, and results in a more efficient SQL query. Let's say you had the following query:

```
from Event e where e.location.name = :name and
    e.location.address.state = :state
```

Since you're walking the object graph twice, once for the `Location` name and again for the `Location` state, the query compiler will join the `Location` instance to the `Event` twice. Joining the `Location` to the `Event` in the FROM clause results in only one join and a more efficient query.

Joined objects can also be returned in the SELECT clause of the HQL statement. The HQL SELECT clause is discussed next.

6.2.3 Selects

The SELECT clause allows you to specify a list of return values to be returned from a query. If you recall from chapter 1, selecting specific columns of data returned from a query is called projection. Possible return values include entire objects, specific object properties, and derived values from a query. Derived values include the results from various functions, such as `min(…)`, `max(…)`, and `count(…)`.

The SELECT clause does not force you to return entire objects. It's possible to return specific fields of objects, just as in SQL. Another interesting feature of HQL is the ability to return new objects from the selected values. We'll examine both features next.

Projection

Suppose that instead of returning the entire `Event` object in your queries, you only want to return the name of the `Event`. Retrieving the entire object just to get the name is pretty inefficient. Instead, your query will only retrieve the desired data:

```
select e.name from Event e
```

This query returns a list of `String` instances containing the `Event` names. If you want to return the `Event` start date in addition to the name, add another parameter to the SELECT clause:

```
select e.name, e.startDate from Event e
```

Each element in the returned list is an `Object[]` containing the specified values. The length of the `Object[]` array is equal to the number of columns retrieved. Listing 6.1 illustrates querying and processing multiple scalar values.

Listing 6.1 Multiple scalar values

```
Session session = factory.openSession();
String query = " select e.name, e.startDate from Event e ";
Query query = session.createQuery("query");
List results = query.list();
for (Iterator I = results.iterator(); i.hasNext();) {
    Object[] row = (Object[]) i.next();
    String name = (String) row[0];
    Date startDate = (Date) row[1];
    // …
}
```

Looking at listing 6.1, you'll notice the values in the `Object[]` array are in the same order given in the query. Also, since the array contains `Object` instances, no primitive values can be returned from a scalar query. This limitation is also present when querying a single scalar value, since a `List` cannot contain primitive values.[1]

A common use of scalar value queries is to populate summary objects containing a subset of the data in the persistent object. In our case, a summary object would consist of the `Event` name and start date. When iterating over the result list, you would need to create a separate list of summary objects. Fortunately, there's a better way to do this.

Returning new objects

The SELECT clause can be used to create new objects, populated from values in the query. Let's look at an example:

```
select new EventSummary(e.name, e.startDate) from Event e
```

[1] The contract for the `java.util.List` interface specifies that it can only store and return instances of `java.lang.Object`. Since primitive types (`int`, `long`, `boolean`, etc.) do not inherit from `java.lang.Object`, they cannot be stored in a `java.util.List`. For more information, refer to http://java.sun.com/docs/books/tutorial/collections/.

The result list will be populated with instances of the `EventSummary` class. Looking at the constructor used in the HQL statement, the `EventSummary` class must have a constructor matching the constructor used in the HQL statement: `EventSummary(String, Date)`.

We have covered the major components of HQL queries. The following section presents the aggregate functions that are available in HQL and how they can be used in SELECT and WHERE clauses.

6.2.4 Using functions

Functions are special commands that return a computed value. In SQL, there are two types of functions: aggregate and scalar. Scalar functions typically operate on a single value and return a single value. There are also scalar functions that don't require arguments, such as `now()` or `CURRENT_TIMESTAMP`, which both return a timestamp. Aggregate functions operate on a collection of values and return a summary value.

Hibernate supports five of the most commonly used SQL aggregate functions: `count`, `avg`, `min`, `max`, and `sum`. The functions perform the same operations as their SQL counterparts, and each operates on an expression. The expression contains the values that the function operates on. Table 6.2 summarizes the five aggregate functions.

The `count(…)` function can also take advantage of the `distinct` and `all` keywords to filter the computed value. Let's look at some examples of using functions in HQL queries.

Table 6.2 Hibernate aggregate functions

Function	Usage
avg(expression)	Calculates the average value of the expression.
count(expression)	Counts the number of rows returned by the expression.
max(expression)	Returns the maximum value in the expression.
min(expression)	Returns the minimum value in the expression.
sum(expression)	Returns the sum of column values in the expression.

```
select count(e) from Event e
```

This example returns the number of `Events` persisted in the database. To count the number of distinct `Events`, use the `distinct` keyword:

```
select count(distinct e) from Event e
```

All of the aggregate functions return an `Integer`. The easiest way to retrieve the result is to get the first element in the result list:

```
Integer count =
    (Integer) session.find("select count(distinct e) from "+
                           "Event e").get(0);
```

You may also use functions in scalar value queries:

```
select e.name, count(e) from Event e
```

Suppose you want to get the collection of `Attendees` for a given `Event`. With what you know so far, you would have to retrieve the `Event` and then get the collection of `Attendees`. The code for this is as follows:

```
String query = "from Event e inner join fetch e.attendees "+
               "where e.name = :name";
Query q = session.createQuery(query);
q.setParameter("name", "Opening Plenary");
Event event = (Event) q.list().get(0);
Set attendees = event.getAttendees();
session.close();
```

While this takes six lines of code, there is a much shorter way to obtain a child collection. Hibernate provides the `elements(…)` function to return the elements of a collection:

```
select elements(e.attendees) from Event e where name = :name
```

This query returns a `List` of `Attendee` instances for a given `Event`. If you join the collection in the FROM clause, you can just use the join alias. For example, the next query is the same as our previous query:

```
select elements(a) from Event e
join e.attendees a
where name = :name
```

Functions can also be used in the WHERE clause, which we cover later in this chapter. HQL properties, or attributes available for objects in a query, are presented next.

6.2.5 HQL properties

HQL supports two object properties: `id` and `class`. The `id` property gives you access to the primary key value of a persistent object. Regardless of what you name the `id` property in the object and mapping definition, using `id` in HQL queries will still return the primary key value. For instance, if you have a class with an `id` property named `objectId`, you would still use the `id` property in HQL:

```
from MyObject m where m.id > 50
```

This query selects all instances of MyObject where the `objectId` property value is greater than `50`. You can still use the `objectId` property if you prefer. Think of HQL's `id` property as shorthand for the primary key value. The `class` property provides a similar function.

The `class` property provides access to the full Java class name of persistent objects in an HQL query. This is typically useful when you have mapped a persistent class hierarchy and only want to return classes of a certain type. We'll look at an example to see how the `class` property can be used.

Let's say the `Attendee` class has an association to the `Payment` class. The `Payment` class specifies how the `Attendee` will pay for the `Events`. `Payment` has two subclasses: `CreditCardPayment` and `CashPayment`. You want to retrieve all `Attendees` who have paid with cash:

```
from Attendee a join a.payment p where p.class =
    com.manning.hq.ch06.CashPayment
```

As with the `id` property, you can also return the `class` property in the SELECT statement:

```
select p.id, p.class from Payment p;
```

This query returns all of the ids and classes for `Payment` as a `List` of `Object[]`s. However, the `class` property is not returned as an instance of `java.lang.Class`. Instead, the `class` property is a `java.lang.String`, which has the same value as the discriminator specified in the mapping definition.

The `class` property is only available on object hierarchies—in other words, objects mapped with a discriminator value. If you try to query the `class` property of an object without a discriminator value, you'll receive a `QueryException` stating that the query parser couldn't resolve the `class` property.

The `id` and `class` properties can be used in SELECT and WHERE clauses. Expressions that can occur in the WHERE clause are covered in the next section.

6.2.6 Using expressions

Expressions are used to query object properties for desired criteria. Occurring in the WHERE clause of an HQL query, expressions support most of the common SQL operators that you're probably accustomed to using. We won't explain each available expression but instead give a number of examples demonstrating expressions in HQL.

A number of available expressions are designed to query child collections or attributes of objects contained within a collection. The simplest function is `size(…)`, which returns the number of elements in a collection:

```
from Event e where size(e.attendees) > 0
```

The size of a collection can also be accessed as a property, like `id` and `class`:

```
from Event e where e.attendees.size > 0
```

Which form you choose is entirely up to you; the result is the same. If you are using an indexed collection (array, list, or map), you can take advantage of the functions shown in table 6.3.

Table 6.3 Functions for indexed collections

Function	Description
elements(expression)	Returns the elements of a collection.
indices(expression)	Returns the set of indices in a collection. May be used in a SELECT clause.
maxElement(expression)	Returns the maximum element in a collection containing basic types.
minElement(expression)	Returns the minimum element in a collection containing basic types.
maxIndex(expression)	Returns the maximum index in a collection.
minIndex(expression)	Returns the minimum index in a collection.

All of the functions in table 6.3 can only be used with databases that support subselects. Additionally, the functions can only be used in the WHERE clause of an HQL statement. We looked at the `elements(…)` function earlier, but its usage changes slightly when used in a WHERE clause. Like the `size` property, the maximum and minimum functions can also be used as properties. Let's look at a few examples. First:

```
from Speaker s where maxIndex(s.eventSessions) > 10
```

or in its property form:

```
from Speaker s where s.eventSessions.maxIndex > 10
```

The `maxElement` and `minElement` functions only work with basic data types, such as numbers (`int`s, `long`s, etc.), `String`s, and `Date`s. These functions do not work with persistent objects, like `Speaker`s or `Attendees`. For example, to select all `Event`s with more than 10 available rooms:

```
from Event e where maxElement(e.availableRooms) > 10
```

Indexed collections can also be accessed by their index. For example:

```
from Speaker s where s.eventSessions[3].name = :name
```

The above HQL queries the fourth `EventSession` object in the collection and returns the associated `Speaker` instance. (Remember that collection indexes start at 0, not 1.) Let's look at another, more complicated example:

```
from Speaker s where
    s.eventSessions[ size(s.eventSessions) - 1 ].name = :name
```

You can get creative within the brackets—for instance, passing in an expression to compare properties of the last collection element. Here's another example of querying on indexed collections using an expression within brackets.

```
select e from EventSession e, Speaker s where
    s.eventSessions[ maxIndex(e.eventSessions) ] != e
```

You'll notice that at the end of the previous query, you just referenced the `EventSession` as `e` instead of using the `id` property of the `EventSession`. This demonstrates that you can use a persistent entity in a query. Let's look at a simple example:

```
Session sess = factory.openSession();
Query q = sess.createQuery("from EventSession e where e=?");
q.setEntity(1, myOtherEventSession);
List l = q.list();
sess.close();
```

When you call `Query.setEntity(…)`, the generated SQL doesn't match on all fields of the entity object—only on the `id` value. The generated SQL for our query looks like

```
select e_.id, e_.name from event_sessions as e_ where (e_id=?)
```

Only the `id` of the passed entity is used, so it's perfectly acceptable to use entity instances in your query objects without worrying about overhead.

HQL also supports various operators, including logical and comparison operators. Logical operators include `and`, `any`, `between`, `exists`, `in`, `like`, `not`, `or`, and `some`. The comparison operators include `=`, `>`, `<`, `>=`, `<=`, and `<>`.

Grouping and ordering

The GROUP BY clause is used when returning multiple scalar values from a SELECT clause, with at least one of them the result of an aggregate function. For instance, you need a GROUP BY clause for the following HQL statement:

```
select e.name, count(elements(a)) from Event e
    join e.attendees a group by e.name
```

The GROUP BY clause is necessary so that the `count` function groups the correct `Events` together. Like most other things in relational theory, queries returning both scalar values and values from aggregate functions have a name: vector aggregates. The query shown here is a vector aggregate.

On the other hand, queries returning a single value are referred to as scalar aggregates. Scalar aggregate queries do not require a GROUP BY clause, but vector aggregate queries do. Let's look at a scalar aggregate query:

```
select count(a) from Event e join e.attendees a
```

Since there is nothing to group by in the SELECT clause, no GROUP BY clause is required.

The GROUP BY clause can be used with the HAVING clause to place search criteria on the results of GROUP BY. Using the HAVING clause does not impact the aggregates; instead, it impacts the objects returned by the query. You can use the HAVING clause as you would a WHERE clause since the same expressions are available:

```
select e.name, count(a) from Event e
    join e.attendees a group by e.name
    having length(e.name) > 10
```

This query returns the Event name and number of Attendees if the Event name has more than 10 characters. You can also be more creative in the HAVING clause:

```
select e.name, count(a) from Event e
    join e.attendees a group by e.name
    having size(a) > 10
```

This time you're getting the Event name and Attendee count for all Events with more than 10 Attendees. Of course, now that you have your Event names, you'll probably want to order them.

We've seen the ORDER BY clause in a few queries before this point, and its usage is very straightforward. The ORDER BY clause allows you to sort the result objects in a desired order. You may sort the objects in ascending or descending order, with ascending as the default. Build on the example by adding an ORDER BY clause:

```
select e.name, count(a) from Event e
    join e.attendees a, join e.location l group by e.name
    having size(a) > 10 order by e.name, l.name
```

This query returns the same objects with the same criteria, only now the returned objects are sorted according to the Event and Location names.

HQL provides a powerful mechanism to query persistent objects. The problem with HQL is that it is static and cannot easily be changed at runtime. Creating queries dynamically with string concatenation is a possibility, but that solution is tedious and cumbersome. Hibernate provides a simple API that can be used to create queries at runtime.

6.3 Criteria queries

The Criteria API provides an alternative method to query persistent objects. It allows you to build queries dynamically, using a simple API. Criteria queries are generally used when the number of search parameters can vary.

Despite their relative usefulness, Criteria queries are somewhat limited. Navigating associations is cumbersome, requiring you to create another Criteria, rather than using the dot notation found in HQL. Additionally, the Criteria API does not support the equivalent of `count(…)`, or other aggregate functions. Finally, you can only retrieve complete objects from Criteria queries.

However, the Criteria API can be excellent for certain use cases—for instance, in an advanced search screen where the user can select the field to search on as well as the search value. Let's look at a few examples of using Criteria queries:

```
Criteria criteria = session.createCriteria(Event.class);
criteria.add(Restrictions.between("duration",
    new Integer(60), new Integer(90) );
criteria.add( Restrictions.like("name", "Presen%") );
criteria.addOrder( Order.asc("name") );
List results = criteria.list();
```

The Criteria query is essentially the same as the following HQL:

```
from Event e where (e.duration between 60 and 90) and
    (e.name like 'Presen%') order by e.name
```

The methods in the `Criteria` class always return the current `Criteria` instance, allowing you to create queries in a more concise manner:

```
List results = session.createCriteria(Event.class).
    .add( Restrictions.between("duration", new Integer(60),
        new Integer(90) )
    .add( Restrictions.like("name", "Presen%") )
    .addOrder( Order.asc("name") )
    .list();
```

The result is the same, but the code is arguably cleaner and more concise.

The Criteria API isn't as fully featured as HQL, but the ability to generate a query programmatically using a simple API can lend a great deal of power to your applications.

6.4 Stored procedures

A shortcoming in earlier releases of Hibernate was the lack of support for stored procedures. Thankfully, Hibernate 3 addresses this problem. Stored procedures are defined in the mapping document and declare the name of the stored procedure as well as the return parameters. Let's look at an example.

Suppose we have the following Oracle stored procedure:

```
CREATE FUNCTION selectEvents RETURN SYS_REFCURSOR
AS
    sp_cursor SYS_REFCURSOR;
BEGIN
    OPEN st_cursor FOR
        SELECT id, event_name, start_date, duration
        FROM events;
    RETURN  sp_cursor;
 END;
```

You can see that the stored procedure retrieves four columns from the `events` table, which is used to populate an `Event` instance. Before you can use it, however, you have to declare the stored procedure in the mapping file for the `Event` class:

```
<sql-query name="selectEvents_SP" callable="true">
    <return alias="ev" class="Event">
        <return-property name="id" column="id"/>
        <return-property name="name" column="event_name"/>
        <return-property name="startDate" column="start_date"/>
        <return-property name="duration" column="duration"/>
    </return>
    { ? = call selectEvents() }
</sql-query>
```

Executing the stored procedure is the same as using a named HQL query:

```
Query query = session.getNamedQuery("selectEvents_SP");
List results = query.list();
```

If your stored procedures take parameters, you can set them using the `Query.setParameter(int, Object)` method. Your stored procedures must return a result set to be usable by Hibernate. If you have legacy procedures that don't meet this requirement, you can execute them using the JDBC `Connection`, accessed by `session.connection()`.

Stored procedures are an interesting addition to Hibernate and are useful in organizations that prefer to perform the majority of their database queries as procedures.

6.5 Hibern8IDE

One of the problems with HQL is testing the query to make sure it works. This is typically a problem when you're new to HQL or trying out new features. Hibern8IDE provides a simple interface to your mapping definitions and an HQL console for executing queries interactively. (Of course, you'll also want to add unit tests to your code base for repeatability.)

Hibern8IDE loads the Hibernate configuration file (either hibernate.cfg.xml or hibernate.properties) and the mapping definitions for your persistent objects. Once you have loaded the configuration file and mapping definitions, you can enter HQL queries in the HQL Commander tab. Hibern8IDE also supports executing named queries defined in the mapping documents. After you execute a query, the results are presented in a table that you can browse to ensure the correct objects and properties are returned.

Hibern8IDE is designed to be used from the command line, but you can also start it from an Ant build file:

```
<target name="hibern8" description="Starts Hibern8IDE.">
    <java classname="net.sf.hibern8ide.Hibern8IDE"
        classpathref="project.class.path" fork="true"/>
</target>
```

Hibern8IDE is a useful tool for exploring the query language, especially when you're first starting out with HQL. It is relatively easy to use and provides all of the necessary features for querying your objects.

Hibern8IDE only works with Hibernate 2. The project has been rebranded as HibernateConsole for Hibernate 3. HibernateConsole is a plug-in for the Eclipse IDE.

6.6 Summary

The Hibernate Query Language abstracts queries from the underlying database. While the language is similar to SQL, HQL is object oriented and has features to support querying object graphs.

There are two common methods used to execute an HQL statement. The `Session` interface provides an overloaded find method that can execute queries and return the results. The `Query` interface also offers the ability to execute queries, but it provides more fine-grained control of the query, such as limiting the number of returned objects.

Both the `Query` and `Session` interfaces allow results to be returned as a `List` or as an `Iterator`. The key difference between the two is that the `Iterator` actually retrieves objects when the `next()` method is called. When a `List` is returned, all of the contained objects are populated when the query is executed.

Like JDBC `PreparedStatements`, HQL queries can take positional parameters, denoted with a question mark. However, HQL also supports the concept of named parameters.

The `Criteria` class is used to create queries programmatically. It's handy when you don't know what the exact query will be, as in an

advanced search function where the user can query on various fields. `Criteria`s have some limitations, such as limited object graph navigation and an inability to retrieve specific fields from objects.

When you're first starting out with HQL or a query has you stumped, Hibern8IDE is a great tool. While it doesn't replace a unit test suite, it can save you time when crafting and optimizing queries, or if you just want to explore the syntax or new functionality.

JAVA

HIBERNATE *Quickly*

Hibernate v3

Patrick Peak • Nick Heudecker

Positioned as a layer between the application and the database, Hibernate is a powerful object/relational persistence and query service for Java. It takes care of automating a tedious task: the manual bridging of the gap between object oriented code and the relational database. *Hibernate Quickly* gives you all you need to start working with Hibernate now.

The book focuses on the 20% you need 80% of the time. The pages saved are used to introduce you to the Hibernate "ecosystem": how Hibernate can work with other common development tools and frameworks like XDoclet, Struts, Webwork, Spring, and Tapestry.

The book builds its code examples incrementally, introducing new concepts as it goes. It covers Hibernate's many, useful configuration and design options, breaking a complex subject into digestible pieces. With a gradual "crawl-walk-run" approach, the book teaches you what Hibernate is, what it can do, and how you can work with it effectively.

"I highly recommend this book."

—Christopher Haupt, Senior Engineering Manager, Adobe Systems Inc.

"If you want to learn Hibernate quickly, this book shows you step by step."

—Sang Shin, Java Technology Architect, Sun Microsystems

Patrick Peak is the CTO of a firm that emphasizes open source frameworks and tools for competitive advantage. He lives in Arlington, Virginia.

Nick Heudecker has large scale development experience with projects for Fortune 500 clients, the media, and government. He lives in Chicago, Illinois.

What's Inside

- Writing mapping files and creating associations
- Hibernate with XDoclet, Struts, Webwork, Spring, and Tapestry
- Querying persistent objects
- Using web application architecture
- Testing with JUnit and Ant

Ask the Authors

Ebook edition

www.manning.com/peak

ISBN 1-932394-41-9

$44.95 US/$60.95 Canada