

# Hibernate

---

*A Developer's  
Notebook™*

James Elliott

O'REILLY®

# Criteria Queries

Relational query languages like HQL (and SQL, on which it's based) are extremely flexible and powerful, but they take a long time to truly master. Many application developers get by with a rudimentary understanding, cribbing similar examples from past projects, and calling in database experts when they need to come up with something truly new, or to understand a particularly cryptic query expression.

It can also be awkward to mix a query language's syntax with Java code. The section "Better Ways to Build Queries" in Chapter 3 showed how to at least keep the queries in a separate file so they can be seen and edited in one piece, free of Java string escape sequences and concatenation syntax. Even with that technique, though, the HQL isn't parsed until the mapping document is loaded, which means that any syntax errors it might harbor won't be caught until the application is running.

Hibernate offers an unusual solution to these problems in the form of criteria queries. They provide a way to create and connect simple Java objects that act as filters for picking your desired results. You can build up nested, structured expressions. The mechanism also allows you to supply example objects to show what you're looking for, with control over which details matter and which properties to ignore.

As you'll see, this can be quite convenient. To be fair, it has its own disadvantages. Expanding long query expressions into a Java API makes them take more room, and they'll be less familiar to experienced database developers than a SQL-like query. There are also some things you simply can't express using the current criteria API, such as *projection* (retrieving a subset of the properties of a class, e.g., "select title, id from com.oreilly.hh.Track" rather than "select \* from com.oreilly.hh.Track") and *aggregation* (summarizing results, e.g., getting the sum, average, or count of a property). The next chapter shows how to accomplish such tasks using Hibernate's object-oriented query language.

## *In this chapter:*

- *Using Simple Criteria*
- *Compounding Criteria*
- *Applying Criteria to Associations*
- *Querying by Example*

# Using Simple Criteria

Let's start by building a criteria query to find tracks shorter than a specified length, replacing the HQL we used in Example 3-9 and updating the code of Example 3-10.

## How do I do that?

The first thing we need to figure out is how to specify the kind of object we're interested in retrieving. There is no query language involved in building criteria queries. Instead, you build up a tree of Criteria objects describing what you want. The Hibernate Session acts as a factory for these criteria, and you start, conveniently enough, by specifying the type of objects you want to retrieve.

Edit *QueryTest.java*, replacing the contents of the `tracksNoLongerThan()` method with those shown in Example 8-1.

**Example 8-1.** The beginnings of a criteria query

```
public static List tracksNoLongerThan(Time length, Session session)
    throws HibernateException
{
    Criteria criteria = session.createCriteria(Track.class);
    return criteria.list();
}
```

*These examples assume the database has been set up as described in the preceding chapters. If you don't want to go through all that, download the sample code, then jump into this chapter and run the "codesgen", "schema", and "ctest" targets.*

The session's `createCriteria()` method builds a criteria query that will return instances of the persistent class you supply as an argument. Easy enough. If you run the example at this point, of course, you'll see all the tracks in the database, since we haven't gotten around to expressing any actual *criteria* to limit our results yet (Example 8-2).

**Example 8-2.** Our fledgling criteria query returns all tracks

```
% ant qtest
...
qtest:
[java] Track: "Russian Trance" (PPK) 00:03:30, from Compact Disc
[java] Track: "Video Killed the Radio Star" (The Buggles) 00:03:49, from VHS
Videocassette Tape
[java] Track: "Gravity's Angel" (Laurie Anderson) 00:06:06, from Compact Disc
[java] Track: "Adagio for Strings (Ferry Corsten Remix)" (Ferry Corsten,
William Orbit, Samuel Barber) 00:06:35, from Compact Disc
[java] Track: "Adagio for Strings (ATB Remix)" (ATB, William Orbit, Samuel
Barber) 00:07:39, from Compact Disc
[java] Track: "The World '99" (Ferry Corsten, Pulp Victim) 00:07:05, from
Digital Audio Stream
[java] Track: "Test Tone 1" 00:00:10
[java] Comment: Pink noise to test equalization
```

OK, easy enough. How about picking the tracks we want? Also easy! Add a new import statement at the top of the file:

```
import net.sf.hibernate.expression.*;
```

Then just add one more line to the method, as in Example 8-3.

**Example 8-3.** The criteria query that fully replaces the HQL version from Chapter 3

```
public static List tracksNoLongerThan(Time length, Session session)
    throws HibernateException
{
    Criteria criteria = session.createCriteria(Track.class);
    criteria.add(Expression.le("playTime", length));
    return criteria.list();
}
```

The Expression class acts as a factory for obtaining Criterion instances that can specify different kinds of constraints on your query. Its le() method creates a criterion that constrains a property to be less than or equal to a specified value. In this case we want the Track's playTime property to be no greater than the value passed in to the method. We add this to our set of desired criteria.

We'll look at some other Criterion types available through Expression in the next section. Appendix B lists them all, and you can create your own implementations of the Criterion interface if you've got something new you want to support.

*Just like HQL, expressions are always in terms of object properties, not table columns.*

Running the query this time gives us just the tracks that are no more than seven minutes long, as requested by the main() method (Example 8-4).

**Example 8-4.** Results of our complete simple criteria query

```
% ant qtest
...
qtest:
    [java] Track: "Russian Trance" (PPK) 00:03:30, from Compact Disc
    [java] Track: "Video Killed the Radio Star" (The Buggles) 00:03:49, from VHS
Videocassette Tape
    [java] Track: "Gravity's Angel" (Laurie Anderson) 00:06:06, from Compact Disc
    [java] Track: "Adagio for Strings (Ferry Corsten Remix)" (Ferry Corsten,
William Orbit, Samuel Barber) 00:06:35, from Compact Disc
    [java] Track: "Test Tone 1" 00:00:10
    [java] Comment: Pink noise to test equalization
```

A surprising number of the queries used to retrieve objects in real applications are very simple, and criteria queries are an extremely natural and compact way of expressing them in Java. Our new tracksNoLongerThan() method is actually shorter than it was in

Example 3-10, and that version required a separate query (Example 3-9) to be added to the mapping document as well! Both approaches lead to the same patterns of underlying database access, so they are equally efficient at runtime.

In fact, you can make the code even more compact. The `add()` and `createCriteria()` methods return the `Criteria` instance, so you can continue to manipulate it in the same Java statement. Taking advantage of that, we can boil the method down to the version in Example 8-5.

**Example 8-5.** An even more compact version of our criteria query

```
public static List tracksNoLongerThan(Time length, Session session)
    throws HibernateException
{
    return session.createCriteria(Track.class).
        add(Expression.le("playTime", length)).list();
}
```

The style you choose is a trade-off between space and readability (although some people may find the compact, run-on version even more readable). Even though this is marked as an experimental API, it already looks extremely useful, and I expect to adopt it in many places.

## What about...

...Sorting the list of results, or retrieving a subset of all matching objects? Like the `Query` interface, the `Criteria` interface lets you limit the number of results you get back (and choose where to start) by calling `setMaxResults()` and `setFirstResult()`. It also lets you control the order in which they're returned (which you'd do using an `order by` clause in an HQL query), as shown in Example 8-6.

**Example 8-6.** Sorting the results by title

```
public static List tracksNoLongerThan(Time length, Session session)
    throws HibernateException
{
    Criteria criteria = session.createCriteria(Track.class);
    criteria.add(Expression.le("playTime", length));
    criteria.addOrder(Order.asc("title"));
    return criteria.list();
}
```

The `Order` class is just a way of representing orderings. It has two static factory methods, `asc()` and `desc()`, for creating ascending or descending orderings respectively. Each takes the name of the property to be sorted. The results of running this version are in Example 8-7.

**Example 8-7.** The sorted results

```
% ant qtest
```

```
...
```

```
qtest:
```

```
[java] Track: "Adagio for Strings (Ferry Corsten Remix)" (Ferry Corsten,  
William Orbit, Samuel Barber) 00:06:35, from Compact Disc
```

```
[java] Track: "Gravity's Angel" (Laurie Anderson) 00:06:06, from Compact Disc
```

```
[java] Track: "Russian Trance" (PPK) 00:03:30, from Compact Disc
```

```
[java] Track: "Test Tone 1" 00:00:10
```

```
[java] Comment: Pink noise to test equalization
```

```
[java] Track: "Video Killed the Radio Star" (The Buggles) 00:03:49, from VHS  
Videocassette Tape
```

You can add more than one Order to the Criteria, and it will sort by each of them in turn (the first gets priority, and then if there are any results with the same value for that property, the second ordering is applied to them, and so on).

## Compounding Criteria

As you might expect, you can add more than one Criterion to your query, and all of them must be satisfied for objects to be included in the results. This is equivalent to building a compound criterion using `Expression.conjunction()`, as described in Appendix B. As in Example 8-8, we can restrict our results so that the tracks also have to contain a capital "A" somewhere in their title by adding another line to our method.

**Example 8-8.** A pickier list of short tracks

```
Criteria criteria = session.createCriteria(Track.class);  
criteria.add(Expression.le("playTime", length));  
criteria.add(Expression.like("title", "%A%"));  
criteria.addOrder(Order.asc("title"));  
return criteria.list();
```

With this in place, we get fewer results (Example 8-9).

**Example 8-9.** Tracks of seven minutes or less containing a capital A in their titles

```
qtest:
```

```
[java] Track: "Adagio for Strings (Ferry Corsten Remix)" (Ferry Corsten,  
William Orbit, Samuel Barber) 00:06:35, from Compact Disc
```

```
[java] Track: "Gravity's Angel" (Laurie Anderson) 00:06:06, from Compact Disc
```

If you want to find any objects matching any one of your criteria, rather than requiring them to fit all criteria, you need to explicitly use `Expression.disjunction()` to group them. You can build up combinations

*Criteria queries are a surprising mix of power and convenience.*

of such groupings, and other complex hierarchies, using the built-in criteria offered by the Expression class. Check Appendix B for the details. Example 8-10 shows how we'd change the sample query to give us tracks that *either* met the length restriction or contained a capital A.

**Example 8-10.** Picking tracks more leniently

```
Criteria criteria = session.createCriteria(Track.class);
Disjunction any = Expression.disjunction();
any.add(Expression.le("playTime", length));
any.add(Expression.like("title", "%A%"));
criteria.add(any);
criteria.addOrder(Order.asc("title"));
    return criteria.list();
```

This results in us picking up a new version of "Adagio for Strings" (Example 8-11).

**Example 8-11.** Tracks whose title contains the letter A, or whose length is seven minutes or less

qtest:

```
[java] Track: "Adagio for Strings (ATB Remix)" (ATB, William Orbit, Samuel Barber) 00:07:39, from Compact Disc
[java] Track: "Adagio for Strings (Ferry Corsten Remix)" (Ferry Corsten, William Orbit, Samuel Barber) 00:06:35, from Compact Disc
[java] Track: "Gravity's Angel" (Laurie Anderson) 00:06:06, from Compact Disc
[java] Track: "Russian Trance" (PPK) 00:03:30, from Compact Disc
[java] Track: "Test Tone 1" 00:00:10
[java] Comment: Pink noise to test equalization
[java] Track: "Video Killed the Radio Star" (The Buggles) 00:03:49, from VHS Videocassette Tape
```

Finally, note that it's still possible, thanks to the clever return values of these methods, to consolidate our method into a single expression (Example 8-12).

**Example 8-12.** Taking code compactness a bit too far

```
return session.createCriteria(Track.class).add(Expression.disjunction().
    add(Expression.le("playTime", length)).add(Expression.like("title", "%A%"))).
    addOrder(Order.asc("title")).list();
```

Although this yields the same results, I hope you agree it doesn't do good things for the readability of the method (except perhaps to LISP experts)!

You can use the facilities in Expression to build up a wide variety of multi-part criteria. Some things still require HQL, and past a certain threshold of complexity, you're probably better off in that environment. But you can do a lot with criteria queries, and they're often the right way to go.

# Applying Criteria to Associations

So far we've been looking at the properties of a single class in forming our criteria. Of course, in our real systems, we've got a rich set of associations between objects, and sometimes the details we want to use to filter our results come from these associations. Fortunately, the criteria query API provides a straightforward way of performing such searches.

## How do I do that?

Let's suppose we're interested in finding all the tracks associated with particular artists. We'd want our criteria to look at the values contained in each `Track`'s `artists` property, which is a collection of associations to `Artist` objects. Just to make it a bit more fun, let's say we want to be able to find tracks associated with artists whose `name` property matches a particular SQL string pattern.

Let's add a new method to `QueryTest.java` to implement this. Add the method shown in Example 8-13 after the end of the `tracksNoLongerThan()` method.

**Example 8-13.** Filtering tracks based on their artist associations

```
1 /**
2  * Retrieve any tracks associated with artists whose name matches a
3  * SQL string pattern.
4  *
5  * @param namePattern the pattern which an artist's name must match
6  * @param session the Hibernate session that can retrieve data.
7  * @return a list of {@link Track}s meeting the artist name restriction.
8  * @throws HibernateException if there is a problem.
9  */
10 public static List tracksWithArtistLike(String namePattern, Session session)
11     throws HibernateException
12 {
13     Criteria criteria = session.createCriteria(Track.class);
14     Criteria artistCriteria = criteria.createCriteria("artists");
15     artistCriteria.add(Expression.like("name", namePattern));
16     criteria.addOrder(Order.asc("title"));
17     return criteria.list();
18 }
```

Line 14 creates a second `Criteria` instance, attached to the one we're using to select tracks, by following the tracks' `artists` property. We can add constraints to either `criteria` (which would apply to the properties of the `Track` itself), or to `artistCriteria`, which causes them to apply to

the properties of the Artist entities associated with the track. In this case, we are only interested in features of the artists, so line 15 restricts our results to tracks associated with at least one artist whose name matches the specified pattern. (By applying constraints to both Criteria, we could restrict by both Track and Artist properties.)

Line 16 requests sorting on the tracks we get back, so we'll see the results in alphabetical order. In the current implementation of criteria queries, you can only sort the outermost criteria, not the subcriteria you create for associations. If you try, you'll be rewarded with an UnsupportedOperationException.

*At least its error message is helpful and descriptive!*

To see all this in action, we need to make one more change. Modify the main() method so that it invokes this new query, as shown in Example 8-14.

**Example 8-14.** Calling the new track artist name query

```
...
// Ask for a session using the JDBC information we've configured
Session session = sessionFactory.openSession();
try {
    // Print tracks associated with an artist whose name ends with "n"
    List tracks = tracksWithArtistLike("%n", session);
    for (ListIterator iter = tracks.listIterator() ;
    ...
```

Running `ant qtest` now gives the results shown in Example 8-15.

**Example 8-15.** Tracks associated with an artist whose name ends with the letter n

```
qtest:
[java] Track: "Adagio for Strings (Ferry Corsten Remix)" (Ferry Corsten,
William Orbit, Samuel Barber) 00:06:35, from Compact Disc
[java] Track: "Gravity's Angel" (Laurie Anderson) 00:06:06, from Compact Disc
[java] Track: "The World '99" (Ferry Corsten, Pulp Victim) 00:07:05, from
Digital Audio Stream
```

## What just happened?

If you look at the lists of artists for each of the three tracks that were found, you'll see that at least one of them has a name ending in "n" as we requested. Also notice that we have access to all the artists associated with each track, not just the ones that matched the name criterion. This is what you'd expect and want, given that we've retrieved the actual Track entities. You can run criteria queries in a different mode, by calling `setResultTransformer(Criteria.ALIAS_TO_ENTITY_MAP)`, which causes it to return a list of hierarchical Maps in which the criteria at each

level have filtered the results. This goes beyond the scope of this notebook, but there are some examples of it in the reference and API documentation.

*You can also create aliases for the associations you're working with, and use those aliases in expressions. This starts getting complex but it's useful. Explore it someday.*

---

### TIP

If the table from which you're fetching objects might contain duplicate entries, you can achieve the equivalent of SQL's "select distinct" by calling `setResultTransformer(Criteria.DISTINCT_ROOT_ENTITY)` on your criteria.

---

## Querying by Example

If you don't want to worry about setting up expressions and criteria, but you've got an object that shows what you're looking for, you can use it as an example and have Hibernate build the criteria for you.

### How do I do that?

Let's add another query method to `QueryTest.java`. Add the code of Example 8-16 to the top of the class where the other queries are.

**Example 8-16.** Using an example entity to populate a criteria query

```
1 /**
2  * Retrieve any tracks that were obtained from a particular source
3  * media type.
4  *
5  * @param sourceMedia the media type of interest.
6  * @param session the Hibernate session that can retrieve data.
7  * @return a list of {@link Track}s meeting the media restriction.
8  * @throws HibernateException if there is a problem.
9  */
10 public static List tracksFromMedia(SourceMedia media, Session session)
11     throws HibernateException
12 {
13     Track track = new Track();
14     track.setSourceMedia(media);
15     Example example = Example.create(track);
16
17     Criteria criteria = session.createCriteria(Track.class);
18     criteria.add(example);
19     criteria.addOrder(Order.asc("title"));
20     return criteria.list();
21 }
```

Lines 13 and 14 create the example `Track` and set the `sourceMedia` property to represent what we're looking for. Line 15 wraps it in an `Example`

object. This object gives you some control over which properties will be used in building criteria and how strings are matched. The default behavior is that null properties are ignored, and that strings are compared in a case-sensitive and literal way. You can call `example's excludeZeroes()` method if you want properties with a value of zero to be ignored too, or `excludeNone()` if even null properties are to be matched. An `excludeProperty()` method lets you explicitly ignore specific properties by name, but that's starting to get a lot like building criteria by hand. To tune string handling, there are `ignoreCase()` and `enableLike()` methods, which do just what they sound like.

Line 17 creates a criteria query, just like our other examples in this chapter, but we then add our example to it instead of using `Expression` to create a criterion. Hibernate takes care of translating `example` into the corresponding criteria. Lines 19 and 20 are just like our previous query methods: setting up a sort order, running the query, and returning the list of matching entities.

Once again we need to modify the `main()` method to call our new query. Let's find the tracks that came from CDs. Make the changes shown in Example 8-17.

**Example 8-17.** Changes to `main()` to call our example-driven query method

```
...
// Ask for a session using the JDBC information we've configured
Session session = sessionFactory.openSession();
try {
    // Print tracks that came from CDs
    List tracks = tracksFromMedia(SourceMedia.CD, session);
    for (ListIterator iter = tracks.listIterator() ;
    ...
```

Running this version produces output like Example 8-18.

**Example 8-18.** Results of querying by example for tracks from CDs

```
qtest:
[java] Track: "Adagio for Strings (ATB Remix)" (ATB, William Orbit, Samuel
Barber) 00:07:39, from Compact Disc
[java] Track: "Adagio for Strings (Ferry Corsten Remix)" (Ferry Corsten,
William Orbit, Samuel Barber) 00:06:35, from Compact Disc
[java] Track: "Gravity's Angel" (Laurie Anderson) 00:06:06, from Compact Disc
[java] Track: "Russian Trance" (PPK) 00:03:30, from Compact Disc
```

---

## TIP

You might think this is something of a contrived example, in that we didn't actually have a handy `Track` object around to use as an example, and had to create one in the method. Well, perhaps... but there *is* a valuable reason to use this approach: it gives you even more compile-time checking than pure criteria queries. While any criteria query protects against syntactic runtime errors in HQL queries, you can still make mistakes in your property names, which won't be caught by the compiler, since they're just strings. When building example queries, you actually set property values using the mutator methods of the entity classes. This means if you make a typo, Java catches it at compile time.

---

As you might expect, you can use examples with subcriteria for associated objects, too. We could rewrite `tracksWithArtistLike()` so that it uses an example `Artist` rather than building its criterion "by hand." We'll need to call `enableLike()` on our example. Example 8-19 shows a concise way of doing this.

**Example 8-19.** Updating the artist name query to use an example artist

```
public static List tracksWithArtistLike(String namePattern, Session session)
    throws HibernateException
{
    Criteria criteria = session.createCriteria(Track.class);
    Example example = Example.create(new Artist(namePattern, null, null));
    criteria.createCriteria("artists").add(example.enableLike());
    criteria.addOrder(Order.asc("title"));
    return criteria.list();
}
```

Remember that if you want to try running this you'll need to switch `main()` back to the way it was in Example 8-14.

A great variety of queries that power the user interface and general operation of a typical data-driven Java application can be expressed as criteria queries, and they provide advantages in readability, compile-time type checking, and even (surprisingly) compactness. As far as experimental APIs go, I'd call this a winner.

When criteria queries don't quite do the job, you can turn to the full power of HQL, which we'll investigate in the next chapter.

*Criteria queries are pretty neat, aren't they? I like them a lot more than I expected.*