| [Print](javascript:window.print()) | [Close](javascript:parent.window.close()) |

**Course Transcript**

Advanced Querying in Java Hibernate

**Advanced Querying**

| [1. Course Introduction](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a03_it_enus/output/html/course_transcript.html#t2) |

| [2. Projections and Aggregation](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a03_it_enus/output/html/course_transcript.html#t6) |

| [3. Named Queries](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a03_it_enus/output/html/course_transcript.html#t10) |

| [4. Advanced HQL](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a03_it_enus/output/html/course_transcript.html#t14) |

| [5. Interesting Queries in HQL](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a03_it_enus/output/html/course_transcript.html#t18) |

| [6. Sub-Queries in Criteria](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a03_it_enus/output/html/course_transcript.html#t22) |

| [7. Interesting Queries in Criteria](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a03_it_enus/output/html/course_transcript.html#t26) |

**SQL Manipulation**

| [1. Stored Procedures](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a03_it_enus/output/html/course_transcript.html#t31) |

| [2. Identity Management within XML](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a03_it_enus/output/html/course_transcript.html#t35) |

| [3. Commenting SQL](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a03_it_enus/output/html/course_transcript.html#t39) |

| [4. Using @Sort for Java-based Sorting](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a03_it_enus/output/html/course_transcript.html#t43) |

| [5. Using @JoinTable](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a03_it_enus/output/html/course_transcript.html#t47) |

| [6. Using many-to-many Relationships](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a03_it_enus/output/html/course_transcript.html#t51) |

| [7. The @Transient Annotation](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a03_it_enus/output/html/course_transcript.html#t55) |

**Debugging and Performance**

| [1. Debugging in Hibernate](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a03_it_enus/output/html/course_transcript.html#t60) |

| [2. Improving General Performance](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a03_it_enus/output/html/course_transcript.html#t64) |

| [3. Improving Query Performance](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a03_it_enus/output/html/course_transcript.html#t68) |

| [4. Read Only Objects](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a03_it_enus/output/html/course_transcript.html#t72) |

Course Introduction

Learning Objective

*After completing this topic, you should be able to*

* *start the course*

**1. Introduction to the course**

The Hibernate Query language enables advanced querying of Hibernate data objects, including detailed searches across multiple tables, classes, joins, and select. Tony Lowe is an experienced software engineer and instructor and in this course he will explore advanced HQL querying, manipulation of SQL databases and data structures, as well as debugging and improving performance in Java Hibernate. Let's begin.

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a03_it_enus/output/html/course_transcript.html#top)

Projections and Aggregation

Learning Objective

*After completing this topic, you should be able to*

* *apply aggregations and projections to a query within a database in Java Hibernate*

**1. Applying aggregations and projections**

To analyze database data, we often need to use advanced SQL to categorize or limit data being returned by a query. Criteria objects provide services to add projections which modify or limit the data being returned. Let's take a look at those. If you've been working with databases for a while, you've been using projections and aggregations, even if you didn't know you were. They're basically a fancy set of terms that are features in the database that allows you to summarize data, analyze data, group data, and otherwise come up with interesting query results. So in Criteria objects I can set projections, or one, or a number of them in order to be get some interesting results. So there's a series of them right here, and we're going to run through one after the other. So the first simple projection that we can say is, I'm going to do a criteria search on Story, and my Projection is going to say return to me the rowCount(). So simply, it's just going to tell me how many rows did I get back from this. So let me go ahead and run it – let's actually run all of my tests right here – but we'll walk through them one at a time. And so the very first result we see up top, is a 100 stories. So my test generates, randomly, a 100 stories and throws them in the database, and that's what comes back here. That's what I expected to come back here.   
*An instance of Java - AllCommon/src/hibernate/query/StoryCriteria.java - Eclipse. In the Package Explorer, hibernate.query is expanded and StoryCriteria.java is selected. The second pane has two tabs:    
QueryingMain.java and StoryCriteria.java. The StoryCriteria.java tabbed page is open. The third pane is the Console tabbed pane and the fourth pane is open on the Declaration tab.  
  
The presenter focuses on the StoryCriteria.java tabbed pane, and highlights this code:  
{  
       Criteria c = session.createCriteria(Story.class);  
       c.setProjection(Projections.rowCount());  
       List results = c.list();  
  
       System.err.println ("\n!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!  
       System.err.println ("The count of stories is " + results.get(0)  
  
The presenter selects the QueryingMain.java tabbed pane, clicks the Run button, and selects the StoryCriteria.java tab. The Console pane fills with output data and the presenter scrolls to the section which displays: The count of stories is 100.*   
  
So the next test I'm doing here is I'm saying, I don't want the story count for all stories, I don't want everything that's in the database, I want a specific one inside of there. So I can add that projection, as well as other restrictions, I can put inside of there. So I'm going to say give me a calendar date that's three days ago and so show me all the stories' rowCount, again we are searching on the story. We adding the projection for rowCount, but search that only for stories that's date is between three days ago, the date I set up here, and today. So the story's date needs to between those guys, and then go through and list that stuff. And then it's going to get the count of all the stories in the last three days. So when it ran that, this next line here – the count of all the stories, you can see in the last three days, it was 21. They got 21 stories in the last three days. And so here's the different things that were being spit out from that, the different sequels being spit out. The first one was just simply select count(\*), and the next one was select count(\*) where the storyDate was between those two dates I had passed in. So the logging caught up to the error logging that was going on inside of there.   
*In the StoryCriteria.java tabbed page the presenter scrolls to the section of code:  
  
Calendar cal = calendar.getInstance();  
cal.add(Calendar.Date, -3);  
  
c = session.createCriteria(Story.class);  
c.setProjection(Projections.rowCount());  
c.add(Restrictions.between("storyDAte", cal.fetTime(), new Date  
results = c.list();  
  
System.err.println ("\n!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!  
System.err.println ("The count of stories in the last three days  
  
The presenter scrolls to the part of the output in the Console, which displays:   
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!  
The count of stories in the last three days is 21  
Hibernate: /\* criteria query \*/ select count(\*) as y0\_ from Story this\_  
Hibernate: /\* criteria query \*/ select count(\*) as y0\_ from Story this\_ where this\_.storyDate between ? and ?  
Hibernate: /\* criteria query \*/ select count(\*) as y0\_, this\_.storyDate as y1\_ from Story this\_group by this\_.storyDate  
Hibernate: /\* criteria query \*/ select avg(this\_.pageNumber) as y0\_from Page this\_The count of stories in the last three days is 21.  
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!  
  
The presenter points to the row of exclamation marks when referring to the error logging.*   
  
So again, that's a really simple projection we can put on top of there. Now projections aren't always that simple, sometimes we want more than that. And so in this third example here, we're going to go through and we're going to say, I have a list of projections. As we have two projections, so we don't normally…we don't only just want a rowCount. We want to know, what's the count of stories on each date. And so in SQL I would normally be using a group by, and so the group by is shown as a Projections.groupProperty. So we're saying this is going to add in what's essentially a group by storyDate, and I'm going to go up and list that set of projections. And so this is the next query inside of there. So collects count(\*), group by the storyDate – that's what we added right inside of here – and when we go off and run that one, the third set is there, we're saying, "How many stories are on each day?" Now when I get the answer back here, since we have multiple projections, I'm getting back an array of objects, which is the results set that's coming back. And I get first the date, and then the count on that date, that's what "group by" does behind the scenes. And so we can see on the 15th, I had seven stories, the 16th and 17th, I had ten stories, and so on, and so forth. Each day is going on and adding more stories.   
*The presenter scrolls to the section of code:  
  
c = session.createCriteris(Story.class);  
ProjectionList projections = Projections.projectionList();  
projections.add(Projections.rowCount());  
projections.add(Projections.groupProperty("storyDate"));  
  
c.setProjection(projections);  
results = c.list();  
  
He highlights the output in the Console:  
  
Hibernate: /\* criteria query \*/ select count(\*) as y0\_, this\_.storyDate as y1\_ from Story this\_group by this\_.storyDate  
  
In the StoryCriteria.java pane the presenter scrolls to the section of code:  
  
System.err.println ("\n!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!  
System.err.println ("How many stories on each date?");  
for (Object o : results)  
{  
       Object[] items = (Object[]) o;  
       System.err.printIn (items[1] + " => "  + items[0]0;  
}  
  
He highlights the output in the Console:  
How many stories on each date?  
2014-02-15 00:00:00.0 => 7  
2014-02-16 00:00:00.0 => 10  
2014-02-17 00:00:00.0 => 10  
2014-02-18 00:00:00.0 => 6  
2014-02-19 00:00:00.0 => 12  
2014-02-20 00:00:00.0 => 14  
2014-02-21 00:00:00.0 => 9  
2014-02-22 00:00:00.0 => 11  
2014-02-23 00:00:00.0 => 6  
2014-02-24 00:00:00.0 => 15*   
  
So all those features you can do in SQL, all those different types of queries, the Criteria objects allows you do with these projections and these aggregations. The last example inside of here that we can talk about is going beyond that, and we're going to apply one of these database functions to get some answers out of that. And so we're going to do a criteria now instead on the Page.class. And we're going to get a list of projections again. And the list of projection we're going to add in here, is we want the average of the page number out there. There's a column in the Page.class with the page number, and each page numbered 1, 2, 3, 4, 5, whatever the page number is. And by adding them all up and dividing by the number of pages, we can see the average number of pages in the stories in our database. And so we create this list of projections, we set that into our criteria, on the pages, and then we list the results. And we're just getting one number back and the number we are getting back here is the average number of pages, in this case 1.6 pages. So there's just over a page and a half per story. So some stories have one page, some have two, some have three – in my sample here – and that's how we end up with this average page count.   
*The presenter scrolls to the section of code:   
  
c = session.createCriteria(Page.class);  
projections = Projections.projectionList();  
projections.add(Projections.avg("pageNumber"));  
  
c.setProjection(projections);  
results = c.list();  
  
System.err.println ("\n!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!  
System.err.println ("Average Page Number " + results.get(0));  
  
He highlights the output in the Console:  
  
Average Page Number 1.6462*   
  
Now we're not limited to the things you've seen here, there's all sorts of these features inside of the projections. The projections you can see, there's aliases, and rowCounts, and other counts. Min and max is common ones to choose out there. There's sum. All the different things you can do inside of the database are available, for the most part, inside of here. There's also other restrictions outside of there, so we have, you know, we have as we saw up above, between, but we also have like, empty, and notEmpty, and not, and matching, and greater than, and less than, and disjunctions, and all sorts of things out there that we can choose. Conjunctions as well, and disjunctions, that we can choose in order to be able to use these different aggregations and projections to be able to get exactly the results we want. We're not limited to just simply dealing in mapped objects. With criteria searches, we can get absolutely anything we want to out of the database.   
*The presenter highlights a part of the code, avg, and a pop-up scroll box appears with various projections listed, such as:  
alias(Projection projection, String alias): Projection  
rowCount() : Projection - Projections  
count(String propertyName): CountProjection  
countDistinct(String propertyName): CountProjection  
min(String propertyName): AggregateProjection  
max(String propertyName): AggregateProjection  
  
The presenter highlights the word 'between' in the code and a pop-up scroll box appears with various projections listed, such as:  
like(String propertyName, Object value): Criterion  
isNotEmpty(String propertyName): Criterion  
conjunction(): Conjunction*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a03_it_enus/output/html/course_transcript.html#top)

Named Queries

Learning Objective

*After completing this topic, you should be able to*

* *externalize HML by using annotations or an external XML file in Java Hibernate*

**1. Using named queries**

One of the big complaints in working with JDBC, was managing the query strings that are coming in there. Because Java programmers aren't always the best queriers, and sometimes we need to externalize that to DBAs, to help us optimize our queries. Well moving to HQL, sometimes we fall back into that habit, where we just put all of our queries as Java strings, embedded in our Java code. And we're not stuck with that. Here we have three examples of calling different features, we're going to get the StoryCount, the PageCount, and then some StoryCountBy some topic. We have three different methods that we're going to call, and these methods are going to externalize the HQL, or SQL, (SQL), entirely from the code. So what I'm doing inside of this method is, I'm going to a session and asking for what's called a NamedQuery. And in this case it's going to say give me "countStories", it doesn't matter – it could return a "storyCount," it could return a "resultSet" – anything that's a valid query is for game here. And there's two places I can store these. So this query in particular, is stored as part of the story definition up here, "countStories" is a @NamedNativeQuery.   
*An instance of Java is open with four panes displayed. The first pane is Package Explorer, where hibernate.domain is expanded and Page.java is selected. The second pane has four files on different rabs: NamedStatement, NamedStatement, Story.java and Named.hbm.xml. The third pane is the Console pane and the fourth pane is open on the Search tab.  
  
In the NamedStatement file, the code is:  
{  
       factory = ConfigHelper.getSessionFactory();  
         
       session = factory.getCurrentSession();  
       session.getTransaction().begin();  
         
       BulkInsert adder = new BulkInsert(session);  
       adder.createABunchForSearching();  
         
       NamedStatements ns = new NamedStatements9session);  
         
       System.err.printIn ("Stories: " + ns.getStoryCount());  
       System.err.printIn ("Pages: " + ns.getPageCount());  
       System.err.printIn ("Stories about Dog " + ns.getStoryCountByTopic("Dog"));  
         
       TestHelper.cleanup(session);  
       session.getTransaction().commit();  
    } catch (Throwable t)  
    {  
       t.printStackTrace();  
    } finally  
    {  
        // To kill the run else Hibernate leaves it active.  
        factory.close();  
        System.exit(0);  
    }      
  
The presenter selects the second instance of NamedStatement, which has these methods:  
  
public NamedStatements(Session session)  
{  
       this.session = session;  
}  
     
public int getStoryCount()  
{  
       List results = session.getNamedQuery("countstories").list();  
       BigInteger count = (BigInteger) results.get(0);  
       return count.intValue();  
}  
  
public int getPageCount()  
{  
       List results = session.getNamedQuery("countPages").list();  
       BigInteger count = (BigInteger) results.get(0);  
       return count.intValue();  
}  
  
The presenter highlights the method:  
public int getStoryCount()  
{  
       List results = session.getNamedQuery("countstories").list();  
       BigInteger count = (BigInteger) results.get(0);  
       return count.intValue();  
}  
  
He opens the Story.java file and highlights the line:  
  
@NamedNativeQuery(name = "countStories", query="call countStories()")*   
  
Now there's two types of queries: there's @namedQueries, which are HQL queries, and there's @NamedNativeQueries, which are Hibernate adapters, to go to @nativeSQLQueries. So in this case, I'm actually calling it "stored procedure," just to show you the full extent of what can be done in sort of these, named SQL queries. I could have, "select count from story," that's perfectly valid. I can also do a @StoredProcedure, or any call inside of here. So going back to our source code, when this gets called, it's going to go and look up this configured NamedQuery. It's going to execute that, and return the list() back from executing – in this case, the @StoredProcedure – and give you the results back. So there's zero embedded HQL, inside of my code. Now the other calls are very similar. In this case I'm getting a NamedQuery called "countPages." Even better, because to some degree, this doesn't help, it's still embedded inside of my source code, even though it's an annotation. Even better, I can mix that up with NamedQueries outside, in an external mapping file. And this is brilliant because, I can hand this XML file to my DBAs, and say "here are some queries, edit them all you like, change them, make them perfect." And as long as I can go and tweak them a little bit to make them HQL-friendly, or Hibernate-friendly, we're good to go.   
*The presenter selects the first instance of NamedStatement and highlights "countstories" in the method:  
  
public int getStoryCount()  
{  
       List results = session.getNamedQuery("countstories").list();  
       BigInteger count = (BigInteger) results.get(0);  
       return count.intValue();  
}  
  
He switches to the Story.java file and highlights "call countStories" in the line:  
  
@NamedNativeQuery(name = "countStories", query="call countStories()")  
  
The presenter selects the first instance of NamedStatement and highlights "list()" in the method:  
{  
       List results = session.getNamedQuery("countstories").list();  
       BigInteger count = (BigInteger) results.get(0);  
       return count.intValue();  
}  
  
He then highlights "countPages" in the method:  
public int getPageCount()  
{  
       List results = session.getNamedQuery("countPages").list();  
       BigInteger count = (BigInteger) results.get(0);  
       return count.intValue();  
}  
  
The presenter switches to the Story.java file and points out the line:  
@NamedNativeQuery(name = "countStories", query="call countStories()")  
  
He opens the Named.hbm.xml file where the line is highlighted:  
<sql-query name="countPages">  
select count(\*) from Page  
</sql-query>*   
  
So in this case I have a "countPages" one, that's externalized entirely from Java, it's an XML file. And it's just a straight SQL command, you can see it's a SQL query, inside of my mapping. So it's a SQL command that goes through and loads (\*) from the page count. And It'll get that one there. The third example here we can point to, is an externalized query, where we're doing another NamedQuery, categorized by "topic", but we're having parameters involved. And in this case we're choosing to use an HQL query, we're not limited to it, but it makes it a little easier, because instead of having to deal with parameter locations, we can deal with the actual names. So the name is "topic", and I'm again, putting in the...percentage signs around – to do wildcard searches inside of there – but that comes from this <query> definition, outside of here. So notice the differences, the first one is SQL, it's a </sql-query>, the second one is HQL, we just call it <query. And so we have an HQL query…select\*…select count(\*)...excuse me...from Story where title is like :topic. And the topic is what's getting passed in as, from the source code, as the parameter. It's the setString that's going on inside of here, just like we did in the normal quote, quote, quote, HQL, where we actually do it in line. So really all we're doing is saving ourselves from having to embed our HQL, or our SQL inside of here. So again let's get this guy running, let's just see that it all works, just like we would expect it to.   
*The presenter focuses on the line:  
<sql-query name="countPages">  
select count(\*) from Page  
</sql-query>  
  
He switches to the second instance of NamedStatement and focuses on the method:  
  
public int getStoryCountByTopic(String topic)  
{  
       Query q = session.getNamedQuery("countStoriesByTopic");  
       q.setString("topics", "%" + topic + "%");  
       List results = q.list();  
         
       Long count = (Long) results.get(0);  
       return count.intValue();  
}  
  
The presenter opens the Named.hbm.xml file and highlights the section:  
<query name="countStoriesByTopic">  
       select count(\*) from Story where title like :topic  
</query>   
  
The presenter switches to the second tabbed instance of NamedStatement and highlights the line:  
q.setString("topics", "%" + topic + "%");  
  
He switches to the first instance of NamedStatement and clicks the Run button. The Console displays the output.*   
  
And so when I go and run this guy, you can see I have a 100 stories in my database, that generated 203 pages, and there's 22 stories about dogs out there in the database. And so each of these queries allows me to make a separation between my Java code, and my database, a little further by externalizing that either as an annotation, or in my opinion, even better as an external XML file.   
*In the Console, the presenter highlights the lines:  
  
Stories: 100  
Pages: 203  
Stories about Dog: 22  
  
He switches to the Story.java file for annotations and then he switches to the Named.hbm.xml file for external XML files.*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a03_it_enus/output/html/course_transcript.html#top)

Advanced HQL

Learning Objective

*After completing this topic, you should be able to*

* *use advanced querying in Hibernate and databases*

**1. Using advanced HQL**

As SQL has an impressive depth of features, so HQL has been built to model those features, to be able to give you access to so many different features that are out there. So I have a little test here, that's going to allow us to just play a little bit with HQL, like we could play with SQL. So I have a database with stories and pages out there, and I want to do some searches into the story. So I want to see where I'm going to select into the Story class. So basically I need to put the items I want out there, so let's say, I want to get the storyDate, and the count of stories on that date, from the Story class, okay. In order to get this done, I have to take advantage of the SQL, and also an HQL group by feature. And I have to group by the date of the story. And so this query right here, is very specifically built, to take advantage of the Story name as a class, not as a table – the storyDate attribute of a class, in both cases here – not necessarily the column name. The column could be named anything, but we're looking at this from a Java standpoint. And so the "s" here points to the object of a story. The group by then has to be very careful to pick an attribute...   
*An instance of Java is open with three panes displayed.   
The QueryingMain.java file is open. Below that the Console tab is selected.  
  
The QueryingMain.java file contains the code:  
  
private void playWithGQL(Session session)  
{  
       Query q = session.createQuery("");  
         
       List list = q.list();  
         
       printResults(list);  
}  
  
The presenter updates the line:  
  
Query q = session.createQuery("");  
  
The updated line is:  
  
Query q = session.createQuery("select s.storyDate, count(s) from Story s group by s.storyDate");*   
  
...and then use that attribute consistently throughout. So I'm going to basically get a list of all the dates and the number of stories on that date running this query. So let's give that a shot…beautiful. So I have a list of dates, and a list of the number of stories on each date. Now just to show you, let's say I did something wrong here, let's say I didn't put s.storyDate, let's see if that would still work. See the flexibility inside of there? Okay, in this case it's smart, why? Because the storyDate is only part of the story, it's able to interpret that out. It's not going to be all that picky inside of there. Now we can advance past group by. SQL has, and thus HQL has, the idea of a having clause. So a where clause in SQL, and in HQL, searches before I apply a group by, it limits before I do a group by. A having clause comes after the grouping to say, "okay you do the grouping, and then let's check something here." So I'm going to say – you know, I only really care about very, busy news days. And so I want to say, "I want to check on stories having a count, of let's say, greater than ten outside of there."   
*The presenter runs the code and the output displays in the Console:  
  
2014-02-15 00:00:00.0 => 14  
2014-02-16 00:00:00.0 => 8  
2014-02-17 00:00:00.0 => 12  
2014-02-18 00:00:00.0 => 6  
2014-02-19 00:00:00.0 => 9  
2014-02-20 00:00:00.0 => 11  
2014-02-21 00:00:00.0 => 9  
2014-02-22 00:00:00.0 => 12  
2014-02-23 00:00:00.0 => 8  
2014-02-24 00:00:00.0 => 11  
  
In the QueryingMain.java file, he removes the "s" from the line of code:  
  
Query q = session.createQuery("select s.storyDate, count(s) from Story s group by s.storyDate");  
  
The changed code is:  
Query q = session.createQuery("select s.storyDate, count(s) from Story s group by storyDate");  
  
He runs the changed code and the output does not change. The presenter updates the line to:  
  
Query q = session.createQuery("select s.storyDate, count(s) from Story s group by storyDate having count (\*) > 10");*   
  
So let's check out this query, and see how it goes. So again, I'm going to get the count just like I did before. You see, I'm building up my query, I got this working, I got the stories there, and now I'm going to add in the having clause to make sure it's more interesting from beyond. Now in this case, you can see I've limited there. I'm only getting the 15th, 17th, 19th, 20th, and 24th of the dates, instead of all of them, because I only care about ones that have stories with more than 10 on a date. I can do all sorts of stuff from here, I could again have a where clause, where title like…let's put another percentage in there...I think that's the right syntax, let's give it a shot. Oh "IndexOutOfBounds" array, why? Because I kept the having clause. There's probably no days with 10 stories about dogs on them inside of there. So I have to be careful inside of my query, you can see I get certain errors that come up, because it probably came back with nothing inside of there. But I could add all these items inside of there. If I do something wrong, like let's say I put the wrong thing out here, so I called this to be storyTitle instead, or storyTile, there you go...I spelled it wrong there. That's where we come back with HQL errors.   
*The presenter runs the code and the output is:  
  
2014-02-15 00:00:00.0 => 11  
2014-02-17 00:00:00.0 => 14  
2014-02-19 00:00:00.0 => 11  
2014-02-20 00:00:00.0 => 11  
2014-02-24 00:00:00.0 => 13  
  
In the QueryingMain.java file, he updates the line of code to:  
  
Query q = session.createQuery("select s.storyDate, count(s) from Story s where title like '%Dog%' group by storyDate having count (\*) > 10");  
  
He runs the code and the output displays:  
  
java.lang.IndexOutOfBoundsException: Index: 0, Size: 0  
  
In the QueryingMain.java file, the presenter changes the line of code to:  
  
Query q = session.createQuery("select s.storyDate, count(s) from Story s where storyTile like '%Dog%' group by storyDate having count (\*) > 10");  
  
He runs the code.*   
  
This is the good and bad news about working in HQL. It'll be very clear to us here...Unknown column "storyTile" in 'where clause.' This message is coming back to me though from my SQL, it's not going back to me from the HQL. The HQL just kind of passed it along. So when I said "storyTile," am I trying to get at the story that's in my object, or in my database? So just as a warning, as you're getting into advanced querying, you can do all these great things. You can use length functions, and max functions, and group bys, and all this sort of stuff that's out there. But you have to be very disciplined in building these things, because sometimes you'll get errors from Hibernate, sometimes you'll get errors from the database. And in this case, we're seeing a straight up database error, it's actually coming straight out of mysql.jdbc driver, that's coming here. It's not saying this inside of my database. And so occasionally you'll run into points where you're debugging Hibernate in HQL, and how it's generating your SQL. That's okay, it's no big deal, I just keep working. I rollback, I figure out how it's going, I find out what's working, I go back to the query that's working, and then I go from there. In this case, that query wouldn't have worked for me because I'm not data rich enough to come up with it. I have a good query out there, I just don't have 10 stories about dogs, in that case. So I'd have to look at a different way, but it will work. So go crazy, play with this stuff, and you come up with some amazing HQL queries.   
*The output the presenter displays:  
  
Caused by: com.mysql.jdbc.exceptions.jdbc4.MySQLSyntacErrorException: Unknown column 'storyTile in 'where clause'  
  
In the QueryingMain.java file, the presenter changes the line of code to:  
  
Query q = session.createQuery("select s.storyDate, count(s) from Story s group by storyDate having count (\*) > 10");  
  
He runs the code and scrolls to the output:  
  
2014-02-18 00:00:00.0 => 14  
2014-02-19 00:00:00.0 => 12  
2014-02-20 00:00:00.0 => 13  
2014-02-24 00:00:00.0 => 11*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a03_it_enus/output/html/course_transcript.html#top)

Interesting Queries in HQL

Learning Objective

*After completing this topic, you should be able to*

* *perform Hibernate Query Language (HQL) queries and searches across multiple classes, joins, and multiple selects*

**1. Performing queries in HQL**

HQL by no means is limited in how interesting the queries can be. And so, I want to spend a little time looking through a few what I find to be interesting queries and, more importantly, the capabilities HQL provides us. So the first query here in front of us, is a query that allows us to select across multiple classes. You might get the impression that I can only return one class based off of the mapping structure, but that's entirely not true. I can get any combination of things out there – it just comes back to us as an array of objects instead of as a straight class. So in this case, I'm selecting both the title of the story and the count(p) that's coming back from that story; and so I'm getting that from the Story class and the Page class. Notice that I'm saying class and not table – this is HQL as a reminder – and so where the page, dot, story.id, so the p is the page and it has a method inside there, getStory, which then has a method .getId = s.id. So from the Story class, I'm going to get the id. I'm going to group that by the page's story.id. And that's where I'm getting the count from of the pages.   
*An instance of Java is open. The QueryingMain.java file is displayed. The query is:  
{  
       String hql = "select s.title, count(p) from Story s, page p "  
                  + "where p.story.id = s.id group by p.story.id";  
                    
       Query q = session.createQuery(hql);  
         
       List list = q.list();  
         
       printResults(list);  
}*   
  
And then for that story, I'm going to go by and get the title, and I'm going to go and do that. So I'm going to **Run** this query real quick, just to go and see. And it's going to return back to us the title of the page and the number of pages…the title of the story, excuse me – and the number of pages in each story. So some pages have four…some stories have one, two, or three pages based off of my random generation here. You can see the SQL that got generated from that. And so it's actually going through and selecting pretty much the same sort of syntax we have going on out there. It's just been translated from HQL to SQL – and right there's our HQL, and the second half is the SQL out there, where it just translates most of it over. But you still have your group by; you still have your where clause, and checking the different IDs and such that's coming together. So that's one sample and it's like across multiple stories. Again, quickly, I can show you this List, and I'm going to do…run this in Debug real quick and, when I run it in Debug, it'll pop us over to the debugging mode. And we can see – **Yes**, I want to go over the Debug mode – the variable coming back to us; the list of stories that's coming back there, is now an array of objects. And the array of objects…each element inside of there comes back as being the title-comma-a number.   
*The presenter runs the query and examines the output. Part of the output is:  
  
Doughnut fist bumps Man2  
Doughnuts hugs Car 1  
Eggplant praises Woman 1  
Bamboo high fives Mayor 3  
  
The presenter highlights a line in the output:  
  
Hibernate: /\* select s.title, count(p) from Story s, Page p where p.story.id = s.id group by p.story.id \*/ select story0\_title as col\_0\_0\_, count(page1\_.id) as col\_1\_0\_ from Story story0\_cross join Page page1\_ where page1\_.storyId=story0\_.id group by page1\_.storyId  
  
He clicks the Debug button and a Confirm Perspective Switch pop-up asks, Do you want to open this perspective now? The presenter clicks Yes.  
  
The interface now displays the Java - Eclipse instance:  
Debug - AnnotationsIntro/src/hibernate/QueryingMain.java - Eclipse  
  
The desktop is divided into five panes. The first pane is open on the Debug tab, the second pane is open on the Variables tab, the third pane is open on the QueryingMain.java tab, the fourth pane is open on the Outline tab, and the fifth pane is open on the Console tab.  
  
In the Variables pane the presenter scrolls through the rows of Name and Value columns. He highlights the row called List with the value:   
ArrayList<E> (id+43)  
  
He expands the list node and it shows the elementData subnode with the value:  
  
Object[100] (id=53)  
  
The elementData node has subnodes such as:  
  
[0] with a value of Object[2] (id=55)  
[1] with a value of Object[2] (id=56)  
  
When the line [0] with a value of Object[2] (id=55) is highlighted, a line below displays the code:  
  
[Dog congratulates Mayor, 3]*   
  
And so I just need to treat that accordingly. I get a string and then a number and so I can still manage that within my HQL query quite easily. So let me go ahead and let this play out and finish, and go from there. So very nicely done query here across multiple tables; returned something other than just one of the classes that's coming in. Let me steal another query I have preformed here. So I'm going to come up here; we'll dump this one, and we'll pop this one in its place. Now this one requires a little bit extra, and I'll show you here what I mean here…oops, that goes here. So I'm selecting, in this case, from the Story class, where I want to search in the story classes inside the pages. And so the pages here have words – in the page, obviously – and I want to search those words for a :topic. I want to search about whatever's going to be here and, well, actually when we search on here, the second is Cat. And so as I'm adding in my parameter here – and we've seen this in queries before, hopefully – it is a colon to indicate it's data coming in…that's going to be passed in. And that's been set here as a String, and the topic which is matching…you see the topic matching here. Matching the title here, is Cat. And I have to apply inside of here the % sign, appropriately, as the wildcard.   
*The presenter leaves the Debug tab and switches back to the instance of Java - AnnotationsIntro/src/hibernate/QueryingMain.java - Eclipse.  
  
He clicks the Run and the Console shows part of  the previous list:  
  
Doughnut fist bumps Man 2  
Doughnuts hugs Car 1  
Eggplant praises Woman 1  
Bamboo high fives Mayor 3  
  
The presenter clicks the Breakpoint in the line which displays:  
  
       printResults(list);  
  
A Quickfix Warning icon displays in line:  
       List list = q.list();  
  
In the QueryingMain.java file, the presenter scrolls to a few lines with the code:  
  
//  String hql = "select distinct s from Story s join s.pages as p where p.words like :topic";  
//  q.setString("topic", "%Cat%);  
//  String hql = "from Story where id in (select story.id as storyId from Page group by story.id having count(\*) = 3)";  
  
He copies two of the lines and pastes them into his query. The full query is now:  
  
private void playWithGQL(Session session)  
{  
       String hql = "select distinct s from Story s join s.pages as p where p.words like :topic";  
         
       Query q = session.createQuery("");  
       q.setString("topic", "%Cat%);  
         
       List list = q.list();  
         
       printResults(list);  
}*   
  
So in this case, I can **Run** my query again. Let's go and save it and **Run** it, and I'm truly returning a list of story this time. So I'm getting back what I would expect, a list of story, but the important thing is I'm doing a join inside of here. I know this relationship; when you're restoring a page, I need to join those together to get it back and, furthermore, I have to add the distinct inside of here to make sure. If I don't have the distinct; if I pull that out real quick and **Run** it again – instead of one story being…one story object being returned for each hit, I get one story object to be returned per pages hit. So you can see, there's three pages in this story; I'm getting three of them back. That's why I have the distinct inside of here – to keep it nice and clean. So, that's another option inside of here; I can do joins inside of any tables. I can do left joins, right joins, outer joins, inner joins, or I can even let the HQL kind of sort of what's needed inside of here. So here you can see, the join is happening in the SQL as an inner join. I let HQL determine that it needed to be an inner join in this case. I could've modified that – I could've specified it myself had I so chosen. So I've one more choice here I want to go and show you; let me copy this guy and drop it inside of here. Our third sample…we don't need the parameter anymore. Now notice I don't have a select inside of here, because I just went from Story. I know I'm getting story objects again.   
*The presenter runs the query and a Save and Launch pop-up dialog box appears.   
  
The Select resources to save section includes a QueryingMain.java checkbox. There are Select All and Deselect All buttons, as well as an Always save resources before launching checkbox, which is not selected. The presenter selects the checkbox and clicks OK.  
  
The presenter scrolls to the section of output in the Console:  
  
39 - 2014-02-20 - Cat praises Dog  
47 - 2014-02-17 - Cat promotes President  
48 - 2014-02-24 - Cat fist bumps Mayor  
56 - 2014-02-15 - Cat praises Mayor  
59 - 2014-02-21 - Cat hugs Dog  
60 - 2014-02-23 - Cat promotes Hedgehog  
  
In the QueryingMain.java file, the presenter removes the word distinct from the query line:  
  
      String hql = "select distinct s from Story s join s.pages as p where p.words like :topic";  
  
He runs the query and scrolls to the section of output which displays three of the same lines:  
  
100 - 2014-02-15 - Cat praises Mayor  
  
The presenter replaces the word distinct in the query line and it is now:  
  
     String hql = "select distinct s from Story s join s.pages as p where p.words like :topic";  
  
The presenter runs the query and scrolls to the output line:  
  
Hibernate: /\* select distinct s from Story\_s join s.pages as p where p.words like :topic \*/ select distinct story0\_.id as id1\_3\_, story0\_.creationDate as creation2\_3\_, story0\_.lastModifiedDate as lastModi3\_3\_, story0\_.storyDate as storyDat4\_3\_, story0\_.title as title5\_3\_ from Story story0\_ inner join Page pages1\_ on story0\_id+pages1\_storyId where...  
  
In the QueryingMain.java file, the presenter copies and pastes this line into the query:  
  
String hql = "from Story where id in (select story.id as storyId from Page group by story.id having count(\*) = 3)";  
  
He removes the line:  
       q.setString("topic", "%Cat%);  
  
The query is now:  
private void playWithGQL(Session session)  
{  
       String hql = "from Story where id in (select story.id as storyId from Page group by story.id having count(\*) = 3)";  
         
       Query q = session.createQuery("");  
                 
       List list = q.list();  
         
       printResults(list);  
}*   
  
And so I want to select where the id is in some groups. So what am I searching for? Notice I'm doing a subquery here; I have a basic query, I was getting stories from a subquery. And so from the subquery, I'm going off and getting the story.id – as the storyId – from the pages, where the pages group together have a count(\*) = 3. So I only want stories that are three pages long in this case. I could say the pages is greater than three or less than three, or greater than equal three, one-page stories – whatever I'm looking for. But that's what I'm doing here; is I'm doing a subquery that's identifying the pages that are…the stories that have three pages, and then I'm going to go select those stories – so I'll go along and **Run** this query. And it's not going to get a very big subset here. Just a handful. It's a bit more than a handful, but maybe about a third of this, all the hundreds stories that are created have three stories associated with them. So within my HQL, I can do all sorts of interesting queries; including multiple tables, resultsets that don't have anything to do with a single class but, instead, have multiple data types associated with that. I can do joins between classes; I can even do subqueries and multiple selects and things like that. As long as I adhere to something that produces proper SQL, I can do it within HQL, and HQL will translate that into the SQL for me.   
*The presenter focuses on the line:  
  
       String hql = "from Story where id in (select story.id as storyId from Page group by story.id having count(\*) = 3)";  
  
He runs the query and he examines the section of the output:  
  
75 - 2014-02-15 - Hedgehog high fives Hedgehog  
77 - 2014-02-15 - Eggplant fist bumps Mayor  
83 - 2014-02-15 - Hedgehog high fives Hedgehog  
85 - 2014-02-15 - Doughnut promotes Car  
86 - 2014-02-15 - Woman praises Woman  
91 - 2014-02-15 - Doughnut congratulates Dog  
97 - 2014-02-15 - Doughnut hugs Bamboo  
98 - 2014-02-15 - Bamboo hugs Man*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a03_it_enus/output/html/course_transcript.html#top)

Sub-Queries in Criteria

Learning Objective

*After completing this topic, you should be able to*

* *use Hibernate Criteria queries as inputs to other queries, in order to construct DetachedCriteria using Restrictions and Projections*

**1. Using sub-queries in criteria**

Generally speaking, as I set up Criteria queries, I'm querying into a specific class. There's often a need – as I query into a class though – to use data from another query – a subquery – to customize the search I'm doing, based off the data returned from the other query. Often I want to use my query based off of something else that's in the database already. In Criteria objects, the way I manage that is by introducing this idea of a Detached query. A DetachedCriteria then allows me to set up a query for a class – in this case, the Page.class, that's not attached to a session yet. I can't execute the Criteria; the Criteria normally is something I can execute immediately – as soon as I'm ready to use that. But what I can do, is I can set it up and decorate it with all the different restrictions and projections and things like that, that I want to have – but then, later on, attach it to a query. Now here you can see I have a pageSearch; I'm searching in the pages where I'm doing a search where the topic that I'm being passed in is going to match inside of…somewhere inside the words, or the page, ANYWHERE. So basically, I'm searching a story for a phrase or a word that appears on one of the pages inside of the story.   
*An instance of Java - Eclipse is open. In the Package Explorer, the StoryCriteria.java file is selected and displayed:  
  
{  
       DetachedCriteria pageSearch = DetachedCriteria.forClass(Page.class)  
       pageSearch.add(restrictions.like("words", topic, matchMode.ANYWHERE  
       pageSearch.setProjection(Projections.property("story.id"));  
         
       Criteria storyQuery = session.createCriteria(Story.class);  
       storyQuery.add(Subqueries.propertyIn("id", pageSearch));  
//       storyQuery.setResultTransformer(Criteria.DISTINCT\_ROOT\_ENTITY);  
  
       List list = storyQuery.list();  
       return list;  
}*   
  
And so the Restrictions.*like* is being added to the Detached pageSearch; and I'm actually throwing a Projection out there as well where, what I want to do, is I want it to return the *property* – not the whole page, but just the story.id out there. And so, notice this is not the database column name; it's the attribute story. There's a getter and setter out there in the Page class called getStory, setStory. And on that Story class, there's an attribute for id – and that's what's being returned inside of here. And this Projection is really important because, otherwise, I'm returning the whole class and that throws things off. So on little or no attached criteria, you can set up the attached Criteria any time you want to and, once you get a session, attach it and execute it. There's not a value inside of that, outside of just presetting it up – maybe some performance enhancement. But where it really, really is valuable is now when I have my session, I can create a Criteria search on the Story class, and I can add a restriction to that criteria based off of the subquery.   
  
And the subquery is saying: Hey, go execute a subquery, get the results back, and then do something with it. And so here – I can show you real quick, there's a lot of different things so…the subquery is equal to; the subquery all are equal to, the subquery is greater than, the subquery is less than, or in, or like, or not, or whatever the different features you want. There's a lot of features inside of there and, what I'm saying is – the subquery here of the IDs that are coming back, I want to know if that is in the property id. And so if any page contained the story.id that comes back to me matches my id, I want to load that story. And so that's based off the pageSearch that's coming in. So let me go ahead and execute this query real quick, with the subquery involved, and I get back a whole series of stories, and you can see there's multiples – this guy's come back three times, this guy's come back three times, this guy's come back three times, this one's come back two times….there'll be somewhere…no, there's just two and three. So there's no one-page stories out there, because the reason this is coming back multiple times, is because each of these has multiple pages.   
*The presenter highlights propertyIn in the line:  
  
       storyQuery.add(Subqueries.propertyIn("id", pageSearch));  
  
A pop-up box appears with values such as:  
  
eq(Object value, detachedCriteria dc): Criterion - Subquery  
eqAll(Object value, DetachedCriteria dc): Criterion - Subquery  
exists(DetachedCriteria dc): Criterion - Subquery  
ge(Object value, DetachedCriteria dc): Criterion -Subquery  
  
The presenter opens the QueryingMain.java file and clicks Run.  
  
The presenter focuses on three instances of each of the following lines in the Console:  
  
97 - 2014-02-19 - Cat fist bumps Dog  
94 - 2014-02-24 - Dog hugs Woman  
91 - 2014-02-17 - Bamboo praises Dog  
  
There are two instances of the line:  
  
75 - 2014-02-21 - Dog promotes President*   
  
And the word that's being searched on is showing up, you know, Dog in this case, it's showing up on multiple pages. So what I have to remember inside of here, is just to add in my ResultTransformer. And making I want a DISTINCT page that's coming back from here, I can go **Run** this one more time. And what I'm going to get back is the DISTINCT pages. So I'm getting one story, even though it was found on multiple pages. So the subqueries is not out of the question when I'm using Criteria objects. I just have to create it as essentially what it is; a query on its own, and then attach it into the Criteria of the main query I'm doing. And this actually makes it – I think – very manageable and very usable; and makes a lot of sense on how I would structure the object tree, because that's essentially how I'm structuring the query to begin with. So use subqueries all you like, by taking advantage of DetachedCriteria.   
*The presenter switches to the StoryCriteria.java file and focuses on the line:  
  
//       storyQuery.setResultTransformer(Criteria.DISTINCT\_ROOT\_ENTITY);  
  
He removes the forward slashes, switches back to the QueryingMain.java file, and clicks Run.  
  
He scrolls through the output to the section which reads:  
  
8 - 2014-02-20 - Eggplant hugs Dog  
13 - 2014-02-17 - Dog congratulates Car  
26 - 2014-02-17 - Dog promotes Car  
27 - 2014-02-24 - Mayor praises Dog  
37 - 2014-02-24 - Dog hugs Car  
40 - 2014-02-20 - Dog fist bumps Mayor  
44 - 2014-02-17 - Dog fist bumps Car  
47 - 2014-02-16 - Woman high fives Dog  
48 - 2014-02-19 - Dog high fives Doughnut  
63 - 2014-02-23 - Dog hugs Dog  
67 - 2014-02-18 - Car congratulates Dog  
68 - 2014-02-16 - Woman praises Dog  
73 - 2014-02-20 - Dog promotes President  
74 - 2014-02-23 - Bamboo congratulates Dog  
79 - 2014-02-22 - Dog praises Hedgehog  
85 - 2014-02-24 - Man praises Dog  
87 - 2014-02-20 - Cat fist bumps Dog  
95 - 2014-02-22 - Dog high fives Bamboo  
99 - 2014-02-16 - Hedgehog high fives Dog  
  
The presenter switches back to the StoryCriteria.java file and focuses on the lines:  
  
       DetachedCriteria pageSearch = DetachedCriteria.forClass(Page.class)  
       pageSearch.add(restrictions.like("words", topic, matchMode.ANYWHERE  
       pageSearch.setProjection(Projections.property("story.id"));*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a03_it_enus/output/html/course_transcript.html#top)

Interesting Queries in Criteria

Learning Objective

*After completing this topic, you should be able to*

* *use Hibernate's Criteria query API and conditional logic to conduct detailed data searches and queries across multiple tables*

**1. Conducting queries in Criteria**

So much of our job is putting together queries and the queries can get pretty interesting, pretty quick. And so here's an example of a query in Criteria objects that's searching on both some topic for a title and a date in the story. And so as I assemble my Criteria object, I'm searching on the Story.class. I am going to add some Restriction where the title is going to be Like some topic that was passed in as a parameter. And then I'm going to add in something where the date happens on a particular DATE. And so this very simple test – I can run it and I'm going to get back the results of all items that were by a Dog on the 17th, in this case. And so you can see a very limited subset of the data coming back. Now the thing to note about this query is I could also want to query on either topic or calendar or Date independently. I might not want to do both of them, so it's possible – within Criteria objects – to scale up queries very nicely using conditional logic.   
*An instance of Java - Eclipse is open.  
  
The StoryCriteria.java file is displayed:  
  
// Used in Interesting Queries in Criteria  
public List<Story> allOnATopicAndDate(String topic, Date day)  
{  
       Criteria c = session.createCriteria(Story.class);  
       c.setReadOnly(true);  
         
       c.add(Restrictions.like("title", topic, MatchMode.ANYWHERE));  
         
       Calendar fromDate = Calendar.getInstance();  
       fromDate.setTime(day);  
       fromDate.set(Calendar.HOUR\_OF\_DAY, 0);  
       fromDate.set(Calendar.MINUTE, 0);  
       fromDate.set(Calendar.SECOND, 0);  
       fromDate.set(Calendar.MILLISECOND, 0);  
       c.add(restrictions.erq("storyDate", fromDate.getTime()));  
       return c.list();  
}  
  
The presenter opens the QueryingMain.java file and clicks Run.   
  
In the Console, the output is displayed. The presenter navigates to the section:  
  
45 - 2014-02-17 - Dog fist bumps Car  
45 - 2014-02-17 - Dog fist bumps Car  
45 - 2014-02-17 - Dog fist bumps Car  
67 - 2014-02-17 - Dog high fives Car  
98 - 2014-02-17 - Dog fist bumps Woman  
  
In the QueryingMain.java file the following is shown:  
  
// Interesting Queries in Criteria  
//      driver.run(TestDriver.CRITERIA\_TOPIC\_AND\_DATE);  
//      driver.run(TestDriver.SELECTIVE\_CRITERIA\_DATE);  
//      driver.run(TestDriver.SELECTIVE\_CRITERIA\_TOPIC);  
//      driver.run(TestDriver.SELECTIVE\_CRITERIA\_BOTH);  
//      driver.run(TestDriver.CRITERIA\_ASSOCIATIONS);  
  
// Query By Example  
//       driver.run(TestDriver.BY\_EXAMPLE);  
  
// Projections and Aggregation  
//       driver.run(TestDriver.PROJECTION);  
  
The presenter removes the forward slashes from the beginning of the line:  
  
driver.run(TestDriver.SELECTIVE\_CRITERIA\_DATE);  
  
He then switches to the StoryCriteria.java file.*   
  
So I have another method here which does the same thing except for, in this case, I could choose to either ignore topic or ignore day and send a null and then my query could operate just fine. So if the day is not equal to null, I'll search on the day; if the topic is not equal to null, I'll search on the topic. So now I can **Run** a test where I am just sending in a day, and I can run my second query here...or the second method I'm showing you, and you can see, I get all the stories that happened on the 23rd. Or I could **Run** a test that just sends in the topic. And as I just send in the topic, I'm going to get a list of all of the stories as they come back relating to a Woman. So here's the interesting thing though; the exact-same method, you know, this method right here where I can pass in both – well, I types in…I passed in one or I passed in the other. I can now pass in both of them and, while passing in both of them…   
*The StoryCriteria.java file displays the code:  
  
// Used in Interesting Queries in Criteria  
public List<Story> allOnATopicAndDate(String topic, Date day)  
{  
       Criteria c = session.createCriteria(Story.class);  
       c.setReadOnly(true);  
         
       if (day != null)  
       {  
               Calendar fromDate = Calendar.getInstance();  
               fromDate.setTime(day);  
               fromDate.set(Calendar.HOUR\_OF\_DAY, 0);  
               fromDate.set(Calendar.MINUTE, 0);  
               fromDate.set(Calendar.SECOND, 0);  
               fromDate.set(Calendar.MILLISECOND, 0);  
               c.add(restrictions.erq("storyDate", fromDate.getTime()));  
       }  
         
       if (topic != null)  
       {  
               c.add(Restrictions.like("title", topic, matchMode.ANYWHERE  
       }  
       return c.list  
}  
  
The presenter switches to the QueryingMain.java file and focuses on the line:  
driver.run(TestDriver.SELECTIVE\_CRITERIA\_DATE);  
  
He clicks Run and scrolls to the section of the output that lists stories dated on the twenty third.  
  
In the QueryingMain.java file, the presenter replaces the forward slashes in the beginning of the line:  
//      driver.run(TestDriver.SELECTIVE\_CRITERIA\_DATE);  
  
He removes the forward slashes from the beginning of the line:  
  
driver.run(TestDriver.SELECTIVE\_CRITERIA\_TOPIC);  
  
He clicks Run and scrolls to the section of output which lists stories including the word Woman, for example:  
  
3 - 2014-02-23 - Man hugs Woman  
4 - 2014-02-18 - Woman high fives Man  
17 - 2014-02-23 - Woman promotes Mayor  
  
The presenter switches to the StoryCriteria.java file and focuses on the line:  
public List<Story> allOnATopicAndDate(String topic, Date day)  
  
He switches to the QueryingMain.java file and removes the forward slashes at the beginning of the line:  
//      driver.run(TestDriver.SELECTIVE\_CRITERIA\_BOTH);  
  
The presenter clicks Run.*   
  
I'll run that one more time, I can do query on both of them without having to write multiple methods. So here I have all…everything that relates to a Woman on the 15th. And so this query really makes it easy, and I don't have to mess with APPENDs and ANDs and ORs and all that sort of stuff that I would if I was doing SQL or HQL, because it's all generated for me. And it just takes all the restrictions I add and it sorts them as appropriate; it adds the criteria between them as appropriate. And I can do that myself or I can let it automatically do that, depending on if I choose how the ANDs and ORs are working. Now that's not all we can do for interesting criteria; this is interesting, this allows us to building up, but the other thing we really need to be able to do is to do searches into associations and into other parts of my class. And so one of the things that we've been running into here…as you can see, my results have been returning multiples because, as it comes in, it's actually returning one story for each page it finds. The first story has two pages, the second story has three pages, the third story has one page. And so we're getting that in there; so I've got to do something to clean that up. And so to have an interesting query, I want unique distinct values that are coming back. So in my third query here, I'm going to do a search into, not my title but, actually, my PageText just as another alternative to show you here. And so I create a Criteria where I'm searching on the Story...   
*The presenter scrolls to the section of output:  
  
8 - 2014-02-15 - Cat promotes Woman  
8 - 2014-02-15 - Cat promotes Woman  
48 - 2014-02-15 - Woman fist bumps Cat  
48 - 2014-02-15 - Woman fist bumps Cat  
48 - 2014-02-15 - Woman fist bumps Cat  
95 - 2014-02-15 - Woman hugs President  
  
He switches to the StoryCriteria.java file and focuses on the section:  
  
               Calendar fromDate = Calendar.getInstance();  
               fromDate.setTime(day);  
               fromDate.set(Calendar.HOUR\_OF\_DAY, 0);  
               fromDate.set(Calendar.MINUTE, 0);  
               fromDate.set(Calendar.SECOND, 0);  
               fromDate.set(Calendar.MILLISECOND, 0);  
               c.add(restrictions.erq("storyDate", fromDate.getTime()));  
  
The presenter scrolls to another section:  
  
// Used in Interesting Queries in Criteria  
public List<Story> allOnATopicAndDate(String topic, Date day)  
{  
       Criteria c = session.createCriteria(Story.class);  
       Criteria pageSearch = c.createCriteria("pages");  
       pageSearch.add(Restrictions.like("words", topic, MatchMode.ANYWHERE));  
       c.setResultTransformer(Criteria.DISTINCT\_ROOT\_ENTITY);  
         
       return c.list();  
}*   
  
...but I'm actually creating additional criteria inside of there where I'm searching into the pages. And so the Story has a list of pages that's associated with it. I'm going to search into those pages and, within that pageSearch criteria, I'm going to add a restriction where my words in the page are like the topic. So like the search I was doing up above with like words in the title, I want to search for what's in the pages – and the field in the database is called words for the pages. And then past that, I need to add what's called ResultTransformer. And the ResultTransformer here is just a fancy way of saying make it distinct. So my Criteria is saying DISTINCT\_ROOT\_ENTITY, but it's important in the Java world that we say it's the root entity. I'm not looking for DISTINCT pages, I'm looking for a DISTINCT story. And so I don't want to return three stories back when there's three pages. I want to return one story back. So let me go and **Run** my final little test case here. So I'm going to search in all the pages now and I want to return back a DISTINCT story for each one that makes a hit.   
*The presenter switches to the QueryingMain.java file and replaces the forward slashes in the beginning of the line:  
  
driver.run(TestDriver.SELECTIVE\_CRITERIA\_BOTH);  
  
He removes the forward slashes from the beginning of the line:  
//      driver.run(TestDriver.CRITERIA\_ASSOCIATIONS);  
  
He clicks Run.*   
  
And now notice here, I'm searching for all stories that involve Cars and I'm only getting back one story Id per page that hits that – the page that hits Cars. These stories definitely have more than one page – not all of them, but most of them are going to have between one and three pages; that's what's randomly generated inside of there. But it's really important to my Criteria to be able to assemble up the object in an interesting way to be able to do that. And again, in Criteria I can do that without having to understand SQL; without, you know, the syntax, the exact flow, or without having to assemble SQL or HQL with ANDs and ORs. I let Hibernate do that for me, by just simply decorating this network of objects that's going to come up with a query.   
*In the Console, the presenter scrolls to the section:  
  
3 - 2014-02-20 - Hedgehog fist bumps Car  
7 - 2014-02-24 - Woman high fives Car  
9 - 2014-02-18 - Man high fives Car  
17 - 2014-02-21 - President fist bumps Car  
20 - 2014-02-23 - Car high fives Car  
24 - 2014-02-18 - Car fist bumps Cat  
27 - 2014-02-24 - Mayor praises Car  
28 - 2014-02-15 - Eggplant promotes Car  
  
He switches to the StoryCriteria.java file and focuses on the section:  
  
       Criteria c = session.createCriteria(Story.class);  
       Criteria pageSearch = c.createCriteria("pages");  
       pageSearch.add(Restrictions.like("words", topic, MatchMode.ANYWHERE));  
       c.setResultTransformer(Criteria.DISTINCT\_ROOT\_ENTITY);*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a03_it_enus/output/html/course_transcript.html#top)

Stored Procedures

Learning Objective

*After completing this topic, you should be able to*

* *call a Hibernate stored procedure by creating a NamedQuery and using IN and OUT parameters*

**1. Calling stored procedures**

Using Hibernate does not mean abandoning the use of stored procedures. Hibernate allows interaction with calls to stored procedures and lets you keep the ability to map the results. While most of our updates, and inserts, and deletes are provided by Hibernate, there's still sometimes a need to use existing or create new stored procedures – and Hibernate doesn't exclude you from doing that. It's actually not too bad calling and using stored procedures when you have them and when you need them, and they can solve so many great problems. So in the first example we'll show here, I can actually call a stored procedure by just simply creating a namedQuery. And so over my Story class, I have an annotation out here for a namedQuery, and one of my namedQuerys is called countStories, which calls this stored procedure called countStories. And a stored procedure...we can actually go and look at here, very simply does a SELECT count(\*) FROM story. Not a very interesting procedure; it has no parameters coming in and just returns whatever the resultset is. In this case, the resultset is just one number that's coming back. So in calling that stored procedure, it really looks just like any other namedQuery I could be doing, whether it's SQL, or HQL, or whatever. And so from the Big query, I can do a list and, from that list, I can get the first element. I could also call get unique here and that becomes a BigInteger, which becomes the count and I can return that – quick and simple and easy. So let's go ahead and execute that, just to see that guy **Run**.   
*An instance of Java - Eclipse is open. Three files are displayed on tabbed pages: StoredProc.java, QueryingMain.java, and Story.java. The Console pane is also available.  
  
The presenter focuses on the StoredProc.java file, with the code:  
  
public int storyCount()  
{  
       Query q - session.getNamedQuery("countStories");  
       BigInteger count = (BigInteger) q.list().get(0);  
       return count.intValue();  
}  
  
He switches to the Story.java file and focuses on the section:  
({  
       @NamedNativeQuery(name = "resetStoryequence", query="ALTER TABLE Story AUTO\_INCREMENT = 1"),  
       @NamedNativeQuery(name = "countStories", query="call countStories()")  
})  
@NamedStoredProcedureQueries  
  
From the taskbar, the presenter opens the minimized procs file in Notepad and focuses on the section:  
  
delimiter #  
create procedure countStories()  
proc\_main:begin  
SELECT count(\*) FROM story;  
end proc\_main #  
delimiter ;  
  
He navigates back to the StoredProc.java file in Java and focuses on:  
  
       Query q - session.getNamedQuery("countStories");  
       BigInteger count = (BigInteger) q.list().get(0);  
       return count.intValue();  
  
The presenter switches to the QueryingMain.java file and clicks Run.*   
  
So I'm doing the query; I'm calling the stored procedure, calling that proc, and I'm getting back a 100. So you can see it called this namedQuery to countStories inside of here – very simple and easy. Now that only works for the simplest and easiest of the stored procedures, particularly when there's no parameters involved. So now we're going to do a stored procedure call where there's a parameter that's going to be sent in. It's an *IN* parameter. Now this can also be done with a namedQuery, but I'm going to show you how it's done as a stored procedure call. So from this session, I'm going to get a NamedProcedureCall. I'm going to set up this NamedProcedure out there and that also happens over here inside of our annotations. It can happen in XML as well, but here we're showing the annotations. So that NamedProcedure is calling countStories, and the important thing here is I'm setting up the parameters which is the topic as an *IN* parameter that's a String coming in. So I'm going to search for all the stories about a specific topic, so I can show you that stored procedure over here. So I'm getting a count from it, where the title is like the topic that's being passed in; very simple IN parameter coming into my stored procedure. So if I **Run** this example now…so let me go ahead and change my TestDriver and go to my second example and execute this one...   
*The presenter examines a section of the output:  
  
Call a stored proc to get Story count  
Hibernate: /\* named native SQL query coutStories \*/ call countStories()  
100  
  
In the StoredProc.java file, the presenter scrolls to the section:  
  
public int storyCountOnTopic(String topic)  
{  
       ProcedureCall q = session.getNamedProcedurecall("countStoriesAbout");  
       ParameterRegistration<String> r = q.getParameterRegistration("topic");  
       r.bindValue(topic);  
         
       ProcedureOutputs out = q.getOutputs();  
         
       Output output = out.getCurrent;  
       // This could be any list from any query, in this case we are just getting the count  
       List list = ((ResultSetOutput) output).getResultList();  
       BigInteger count = (Biginteger) list.get(0);  
       return count.intValue();  
}    
  
He switches to the Story.java file and examines the sections:  
  
@NamedtoredProcedureQuery(name = "countStoriesAbout",  
                         procedureName="countStoriesAbout",  
                         parameters=  
                               {@StoredProcedureParameter(name="topic", mode-ParameterMode.IN, type=String.class)}  
                         ),  
  
From the taskbar, the presenter opens the minimized procs file in Notepad and scrolls to the section:  
  
create procedure countStoriesAbout(IN topic Varchar(255))  
proc\_main:begin  
       SELECT count(\*) FROM story where title like CONCAT('%', topic, '%');  
end proc\_main #  
delimiter;  
  
He navigates to the QueryingMain.java file, with the code:  
  
// Stored Proc  
       driver.run(TestDriver.STORED\_PROC);  
//               driver.run(TestDriver.STORED\_PROC\_PARAMS);  
//               driver.run(TestDriver.STORED\_PROC\_OUT\_PARAMS);  
  
The presenter adds two forward slashes at the beginning of the line:  
       driver.run(TestDriver.STORED\_PROC);  
  
He removes the forward slashes at the beginning of the line:  
//               driver.run(TestDriver.STORED\_PROC\_PARAMS);  
  
The presenter clicks Run.*   
  
...this again calls a stored procedure; it's now calling a stored procedure with a parameter inside of there. And it's getting the results back of 14. So there's 14 stories that had the topic that matched what was going on inside of there. Again the code is slightly more complicated, because now I have to create this ProcedureCall instead, and I have to look up the parameter binding, out of the query. Again not all that terrible, but I got to know the name of the parameter binding based off of what I've mapped in the story. So I've mapped the name of topic inside of here, and then I can go through and call it. So it's slightly more involved; so I'm binding the value in here, but I'm still getting back the same results of a number and mapping that off and getting that and returning it back out of there. So the last trick we can show inside of here, is when there is out parameters involved. So in this query, I want to return to you both the count and the list of stories. So how do we make that happen? Well again, I have a namedStoredProcedure here, countStoriesAboutandCount. I want to get both the list of stories and the count. So I have a parameter that goes in, which is the topic – again, the same as we had before, an *IN* mode of a String. And now I have a parameter, returnCount, which is now coming back out.   
*In output displays in the Console. The presenter examines the sections:  
  
Call a stored proc to get Story count  
Hibernate: {call countStoriesAbout(?)}  
Building Return [isResultSet=true, updateCount=-1, extendedReturn=false  
14  
  
In the Storedproc.java file, the presenter highlights the lines:  
  
        ProcedureCall q = session.getNamedProcedurecall("countStoriesAbout");  
       ParameterRegistration<String> r = q.getParameterRegistration("topic");  
  
He switches to the Story.java file and highlights the line:                                {@StoredProcedureParameter(name="topic", mode-ParameterMode.IN, type=String.class)}  
  
Then he switches to the StoredProc.java file and highlights the line:  
       r.bindValue(topic);  
  
The presenter focuses on the section:  
       Output output = out.getCurrent;  
       // This could be any list from any query, in this case we are just getting the count  
       List list = ((ResultSetOutput) output).getResultList();  
       BigInteger count = (Biginteger) list.get(0);  
       return count.intValue();  
  
He switches to the Story.java file and examines the section:  
  
@NamedStoredProcedureQuery(name = "countStoriesAboutandCount",  
                         procedureName="countStoriesAboutandCount",  
                         parameters=  
                               {@StoredProcedureParameter(name="topic", mode-ParameterMode.IN, type=String.class),  
                                @StoredProcedureParameter(name="returnCount", mode-ParameterMode.OUT, type=Integer.class),  
                               },  
                         resultClasses={Story.class}  
                         )*   
  
So I'm returning a count as an *OUT* parameter and then the resultset is going to come back as the default resultset coming from the stored procedure – as would normally operate within a stored procedure. So again I'm going to update to my third solution here. I'm going to go ahead and **Run** this, and then we'll look through that code. But in this third sample, I am getting out a resultset which returns all the story titles that's coming out of there. And separately, I have a count telling me there's 18 of them. Now if I look at the stored procedure here; you can see I have an IN variable for the topic, I have an OUT variable for that returnCount. I'm selecting the count(\*) into the returnCount here. So that's stored procedure logic inside of here, and then I'm selecting everything FROM the story and returning that as the main one here. So I'm saving the count, returning that as a parameter, and then going separately. Basically this is to show you how OUT procedures work. So within our source code for that then, I have the namedProcedure...   
*In the QueryingMain.java file, the presenter replaces the forward slashes at the beginning of the line:  
                  driver.run(TestDriver.STORED\_PROC\_PARAMS);  
  
He removes the forward slashes at the beginning of the line:  
  
//               driver.run(TestDriver.STORED\_PROC\_OUT\_PARAMS);  
  
The presenter clicks Run and the examines the output in the Console. It displays a list of 18 story titles. On a separate line it displays: The count was 18.  
  
From the taskbar, the presenter opens the minimized procs file in Notepad and focuses on the section:  
  
delimiter #  
create procedure countStoriesAboutAndCount(IN topic Varchar(255), OUT returnCount int)  
proc\_main:begin  
SELECT count(\*) into returnCount FROM story where title like CONCAT('%', topic, '%');  
SELECT \* FROM story where title like CONCAT('%', topic, '%');  
end proc\_main #  
delimiter;  
  
He navigates to the StoredProc.java tab.*   
  
...I bind my topic going in the same way I did, I set the Value of topic going in – based off of what was passed in to me. And then I'm going get the Outputs from this. I'm going to go through and, you can see, ask for the OutputParameter called returnCount and then I can get the ResultList for everything else that came back from that stored procedure. So Hibernate allows you to fully bind IN parameters, OUT parameters, no parameters – however you want to with stored procedures – and gives you multiple ways of doing that, either very simply as a named call or, more involved, as a stored procedure call. And as I said, these take advantage of the annotations in this case, but you can do the same thing through XML bindings.   
*In the StoredProc.java file, the presenter examines the code:  
  
public List<Story> outParams(String topic)  
{  
       ProcedureCall q = session.getNamedProcedureCall("countStoriesAboutAndCount");  
       ParameterRegistration<String> r = q.getParameterRegistration("topic");  
       r.bindValue(topic);  
         
       ProcedureOutputs out = q.getOutputs();  
         
       int count = (int) out.getOutputParameterValue("returnCount");  
       System.err.printIn("The count was " + count);  
         
       List list = ((ResultSetOutput) out.getCurrent()).getReultList();  
         
       return list;  
}  
  
He then switches to the Story.java file.*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a03_it_enus/output/html/course_transcript.html#top)

Identity Management within XML

Learning Objective

*After completing this topic, you should be able to*

* *use a basic, composite, or foreign key ID to manage XML identity mapping in Java Hibernate*

**1. Managing XML identity mapping**

As every class in Hibernate is required to have an ID, I have to have a robust set of choices on setting the ID. The most basic ID is from the ID tag, and the ID tag in the XML mapping allows me to point to a single column in the database that represents the ID. There's a lot of options I can do on that, such as setting a generator or managing it myself, it doesn't matter, but it still ties to a single ID. I can't map a class without my ID, but in this case I'm limited to a single column. Now obviously, there are tables that I need something more interesting, something more complex than that. So for instance, I have a class here, no particular business context, but it has a remote ID, as a separate object, that it wants to use as an ID, because there are some – more than one – field that's going to represent how to identify this class, this table. And so if I look at the ID object, in this case we might be identifying a person, based off the lastFour of their Social Security Number, and a zip code, as an example. Again, it's up to your business, but these could be two reasonable fields by which I am going to identify a person. So thus when I'm creating rows in the database, or when I'm searching on people in the database, I want to search on these two columns instead of just one. And so, in my mapping of my embedded ID class here, I have to find a way to tie to more than one column.   
*An instance of Java - Eclipse is open.   
  
Various files are open, including: Identity.hbm, EmbeddedID.java, MyID.java, EmbeddedID.hbm, RemoteID.java, and RemoteID.hbm. The Identity.hbm tabbed pane is open. The Console pane is also available.  
  
The presenter focuses on a section of code in the Identity.hbm file:  
  
<hibernate-mapping package="hibernate.domain.identifier.generator">  
       <class name="Identity" table="Generic\_auto">  
               <id name="id">  
                       <generator class="identity"/>  
               </id>  
               <property name="text"/>  
       </class>  
</hibernate-mapping>  
  
The presenter switches to the EmbeddedID.java file, which has the code:  
  
public class EmbeddedID  
{  
       private MyID id;  
         
       private String name;  
         
       public MyID getId()  
       {  
               return id;  
       }  
         
       public void setId(MyID id)  
       {  
               this.id = id;  
       }  
  
The presenter highlights the line:  
  
       private MyID id;  
  
He switches to the MyID.java file with the code:  
  
public class MyID implements Serializable  
{  
       private String lastFour;  
       private String zip;  
         
       public String getLastFour()  
       {  
               returnlastFour;  
       }  
       public void setLastFour(String lastFour)  
       {  
               this.lastFour = lastFour;  
       }  
  
The presenter focuses on the lines:  
  
       private String lastFour;  
       private String zip;  
  
He then switches to the EmbeddedID.hbm file.*   
  
The way around that is, instead of using the ID tag, I'm now using the composite-ID tag. Again, I can name this as being whatever I want to; it's called ID, which harkens back to the name of the attribute, and the getters and setters inside of my code. That's what this name represents here; ID. And then I can point to an external class – a separate class that I'm going to map the ID class within. I don't have to map it separately – I don't have to create another separate class definition for this – I can define it right here as being the identifier that does the mapping for me. And in here, I'm choosing to put the fully qualified path name inside of this mapping, inside of the class. And so for this, we saw in the Java class we have lastfour and zip, and those are the two things that are going to map inside of there. And so the lastfour needs to match – the name of this needs to match – the getter or setter/attribute name that comes from my Java object. It doesn't have to match what's in the database though; in the database the column name is ssn. And...so that's perfectly okay, I can still do those alternative mappings. In the second example, zip lines up, and so column is optional here. But I go ahead and put it so you can see the name of the attribute is zip, the name of the column is zip.   
*In the EmbeddedID.hbm file, the code is:  
  
<hibernate-mapping package="hibernate.domain.identifier.generator">  
       <class name="EmbeddedID" table="ComplexPerson">  
               <composite-id name="id" class="hibernate.domain.identifier.MyID">  
                       <key-property name="lastFour" column="ssn"/>  
                       <key-property name="zip" column="zip"/>  
               </composite-id>  
                 
               <property name="name" column="name"/>  
        </class>  
</hibernate-mapping>  
  
The presenter highlights the line:  
  
               <composite-id name="id" class="hibernate.domain.identifier.MyID">  
  
He switches to the EmbeddedID.java tab and highlights the line:  
  
       public MyID getId()  
  
He switches to the EmbeddedID.hbm file and highlights the line:  
               <composite-id name="id" class="hibernate.domain.identifier.MyID">  
  
Then he opens the MyID.java file and highlights the lines:  
       private String lastFour;  
       private String zip;  
  
The presenter goes back to the EmbeddedID.hbm file and highlights the line:  
                       <key-property name="lastFour" column="ssn"/>  
  
He returns to the MyID.java file and highlights the lines:  
       private String lastFour;  
       private String zip;*   
  
So the net result of this is any time I would do an operation with the ID normally, for getters or setters or what not, instead of passing in a long or a string that I used to, I'm now expecting to be passed, in a whole, my ID object with both a lastfour digits and a zip that's being used as far as that ID key. And so the ID object essentially becomes the key for this class. So I can go through and I can run my solution here, and show you. I have an example that sets up an embedded object, sets the lastfour and a zip code, and it's going to go and insert that into the database. And it's gone off and done it. So you see, when it does it, it's inserting into name, social security number, and zip. It doesn't represent that Java class and the database at all, because that's a Java mapping thing. So as the database goes, it's just is a database with columns. And so if we look at the database, you can see, I have social security number, zip, and the name of the person. The next option inside of there is, let's say, I don't have a single key or even a compound key, let's have a foreign key that's involved inside of there. So I'm done with my embedded ID example here, we're going to move over to our remote ID. So in this example, I have a separate object that is my partner.   
*The presenter opens the IdentityMain.java file and highlights the code line:  
  
showEmbedded(session);  
  
The examines the code section:  
  
private void showEmbedded(Session session)  
{  
       EmbeddedID person = new EmbeddedID();  
       MyID id = new EmbeddedID();  
       id.setlastFour("1234");  
       id.setZip(90210");  
       person.setID(id);  
       person.setName("My Person:);  
         
       session.persist(person);  
}  
  
The presenter clicks Run and in the Console, examine the output. He focuses on the line of code:  
  
Hibernate: insert into ComplexPerson (name, ssn, zip) values (?,?,?)  
  
From the taskbar, he opens the Command Prompt - mysql - u root - p where he points out the section which has a table displayed. The table has four columns and one row:  
  
remoteId: NULL  
ssn: 1234  
zip: 90210  
name: My Person  
  
The presenter closes the Command Prompt and the EmbeddedID.java, MyID.java, and EmbeddedID.hbm.xml tabs. The RemoteID.java tab remains open.*   
  
I'm tied to that object, and my Identity is determined by that object. And so in this case, I'm tying to just some other generic class, and that's the class that will get loaded up with this. And so if I look at my example for the remote ID, again, I'm creating my partner class, and I'm setting an ID in the partner class of whatever attributes that partner class might have. In this case, I'm just Identifying it as being my partner object. And so when I go and set up my object, I need to first set that partner. And that partner needs to be saved in the database, so that way its ID is established, and then my foreign key can tie to that. And so I'm going to set up whatever other data I have, whatever other data I need in my object, save the partner object, then save my own object. And so the way this all happens is through, again, the same mechanism of a composite-ID. And so the composite-ID is pointing to my partner class out there. And it's saying, "hey, for that partner class, I need you to go into that object, and ask that object for its ID." And so the class partner I'm going to go through is the same thing that we saw in our code, that partner class, the NoGenerator class in this demo. But it's just another object, another class, that's out there. And inside of that class, the key property that we need inside of there is that ID property.   
*The RemoteID.java tab displays the code:  
  
package hibernate.domain.identifier;  
  
import hibernate.domain.identifier.generator.NoGenerator;  
  
public class RemoteID  
{  
       private NoGenerator partner;  
         
       private String name;  
       private String ssn;  
       private String zip;  
         
       public String getName()  
       {  
               return name;  
       }  
         
       public void setName(String name)  
  
He scrolls to code:  
  
private void showRemote(Session session)  
{  
       NoGenerator partner = new NoGenerator();  
       partner.setId(1);  
       partner.setText("partner object");  
         
       RemoteID sample = new RemoteID();  
       sample.setPartner(partner);  
       sample.setName("My Sample");  
       sample.setSsn("none");  
       sample.setZip("none");  
         
       session.persist(partner);  
       session.persist(sample);  
}  
  
He then switches to the RemoteID.hbm.xml tab which contains the code:  
  
<hibernate-mapping package="hibernate.domain.identifier">  
       <class name="RemoteID" table="ComplexPerson">  
               <composite-id name="partner" class="hibernate.domain.identifier.generator.NoGenerator">  
                       <key-property name="id" column="remoteId"/>  
               </composite-id>  
                 
              <property name="zip"/>  
              <property name="ssn"/>  
              <property name="name"/>  
       </class>  
</hibernate-mapping>*   
  
And the column that I'm pointing to in my database is called remoteID. We saw that a second ago, we'll see it in here again. But it's going to go to partner class, it's going to call getID, and it's going to insert that into my column, remote ID, along with the other mapping that's going on. So let's execute our sample of this guy. And so if we go through and we comment out the other example, and run this example, we will get the net results. And so it's inserted first into the generic table, the parent ID, and then it's inserted into the table I'm working on, the person table here, the other ID. And if I look at the database, you can see we start off with the partner object being inserted. We've ignored the other data items from this, but we have the partner object inserted with its primary key of one. And then my object, again, it's a second object I'm putting in here. I'm not worried about the social security number and zip in this time, I'm just reusing the table for the example. But I have a remoteID, and the remoteID is a foreign key over to this guy, which is one. And so my sample here, in this case, is creating an ID that's remotely set. So within Hibernate, I can either use a basic ID, a composite ID, or a ID that's a foreign key, but I still am meeting the mapping demand of having a unique ID for every class.   
*The presenter switches to the IdentityMain.java file code and adds two forward slashes at the beginning of the line:  
  
showEmbedded(session);  
  
The presenter removes the forward slashes at the beginning of the line:  
//      showRemote(session);  
  
He clicks Run and scrolls to the section of output which displays the lines:  
  
Hibernate: insert into Generic (text, id) values (?, ?)  
Hibernate: insert into ComplexPerson (zip, ssn, name, remoteId) values (?, ?, ?, ?)  
  
From the taskbar the presenter opens an instance of Command Prompt - mysql - u root - p where, at the command prompt, he enters:  
source showGenerics.txt;  
  
The output shows two tables with a few lines of output in between the two. The first table has four columns and one row. The column contains the categories and row contains the values:  
  
id: 1  
text: partner object  
aDate: NULL  
sampleMoney: NULL  
aBoolean: NULL  
  
The output lines between the tables displays:  
  
1 row in set <0.00 sec>  
Empty set <0.00 sec>  
Empty set <0.00 sec>  
  
The second table contains four columns and one row. Each column contains a category and each row contains a value.  
  
remoteID: NULL.  
ssn: 1234.  
zip: 90210.  
name: My Person.  
  
The last line of output is:  
2 rows in set <0.00 sec>*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a03_it_enus/output/html/course_transcript.html#top)

Commenting SQL

Learning Objective

*After completing this topic, you should be able to*

* *enable SQL commenting in Java Hibernate for table creation and search queries*

**1. Enable SQL commenting**

As developers, letting Hibernate manage our queries, and manage even our table creation, has some great advantages. But we don't want to take away some of the advantage that we have of natively creating them over on our database, using DDL scripts and such. So one of the limitations can be the loss of comments to our SQL, and we don't want to do that, but we don't have to sacrifice that within our native environment. So if I show you here, our database has no tables inside of it. You can see when I do a show tables command, there's nothing there. If I go back to my code, we can see in my story class for my table tag – and note here the table tag must be the hibernate table tag, not the JPA table tag. I'm now applying this to the table story. And I can add in a comment here, so this is My STory table comment, and I actually have misspelled this somewhat intentionally so we can see it's not generated, it's something that's my text that's showing up here. And so when I go and run my test, it's going to go through first and create the tables, as we can see from the script here. It's creating the table page, it's creating the table story, it's creating a foreign script…a foreign key between them. And it's gone through here, and now when I do a show tables you can see it's created my page and story table. And so if I look at the story table here...   
*An instance of Java - Eclipse is open.   
  
Four files are open: QueryingMain.java, Story.hbm.xml, CommandLineMain.java, and Story.java. The CommandLineMain.java files is displayed. The Console pane is also displayed.  
  
From the taskbar the presenter opens an instance of Command Prompt - mysql - u root - p. He runs the command: show tables;  
  
The output is:  
Empty set <0.02 sec>  
  
In Java, he selects the Story.java file, which contains the code:  
  
import org.hibernate.annotations.Table;  
  
@Entity  
@Table(appliesTo="Story", comment="My STory table comment")  
@NamedNativeQueries  
  
The presenter switches to the CommandLineMain.java tab and clicks Run.  
  
The presenter scrolls to the section of output:  
  
Hibernate: create table Page (id bigint not null auto\_increment, creationDate datetime, htmlincluded bit not null, lastModifie...  
Hibernate: create table Story (id bigint not null auto\_increment, creationDate date, lastModifiedDate datetime, storyDate date...  
Hibernate: alter table Page add constraint FK\_7qm1qw26wvu... foreign key (storyId) references Story (id)  
  
The presenter opens the Command Prompt and runs the command:  
  
show tables;  
  
The table has one column with the heading; Tables\_in\_bignews. The first row shows page and the second row shows story. The output line below the table is: 2 rows in set <0.00 sec>  
  
At the command prompt he runs the command: describe story;  
  
The output is a table with five columns and five rows. The columns are: Field, type, Null, Key, Default, and Extra.  
  
The table reads, for example:  
  
Field: id  
Type: bigint <20>  
Null: NO  
Key: PRI  
Default: NULL  
Extra: auto\_increment  
  
The last line of output after the table is: 5 rows in set <0.00 sec>*   
  
...I can see it's set up the table how I needed to. And it's set it up as I've done in DDL scripts by hand before, but now I've done it straight through Hibernate. Now I've not lost the comments, I don't see the comments here, but I can see the comments by doing a show create table for story. And this is my SQL syntax. But we can see here, in the table script, Hibernate created the table with My STory comment as part of this story creation of the story table. This shows the annotation approach towards that, where we can put it as part of the Hibernate table tag. If you're using XML mapping, you can do that through the comments, and the angle bracket comment class, inside of your story mapping. And so you can do something along the lines of this here. We can add in a comment towards the story, and we can get that same comment as we saw inside of our annotations approach. So we're not limited, whether we do annotations or we do it through XML mapping. Now the other half of this comes into...when we're doing SQL query. And so if we go to a different example here, we have a SQL query out here and we can go ahead and run our SQL query, and we'll get the standard execution of what we have going on here. But now, inside of our Hibernate config – if we hop over to that...   
*Next in the command prompt, he runs the command:  
  
show create table story;  
  
The output is:  
  
--  
--  
:Table : Create Table  
--  
--  
--  
:story : CREATE TABLE 'story' <  
'id' bigint<20> NOT NULL AUTO\_INCREMENT,  
'creationDate 'date DEFAULT NULL,  
'lastModifiedDate' datetime DEFAULT NULL,  
'storyDate' datetime DEFAULT NULL,  
'title' varchar<255> DEFAULT NULL,  
PRIMARY KEY <'id'>  
>ENGINE=InnoDB DEFAULT CHARSET=utf8 COMMENT='My STory table comment' :  
--  
--  
--  
1 row in set <0.00 sec>  
  
In the Story.java tabbed pane the presenter focuses on the lines:  
@Entity  
@Table(appliesTo="Story", comment="My STory table comment")  
@NamedNativeQueries  
  
The presenter switches to the Story.hbm.xml file and highlights the line:  
  
<comment>Comment my Story SQL Creation</comment>  
  
The presenter switches to the QueryingMain.java tab and scrolls to the next example displayed:  
  
private void playWithHQL(Session session)  
{  
       Query q = session.createQuery("select distinct s from Story s join s.pages as p where p.words like :topic");  
       q.setString("topic", "%Dog%");  
       q.setComment("I am searching!!");  
         
       List list = q.list();  
         
       printResults(list);  
}  
  
The presenter runs the query and examines a section of the output:  
  
62 - 2014-02-19 - Dog congratulates Mayor  
71 - 2014-02-19 - Dog high fives Doughnut  
73 - 2014-02-15 - Dog praises Woman  
89 - 2014-02-19 - Doughnut hugs Dog  
  
In the Package Explorer pane the presenter double-clicks the hibernate.cfg.xml file.*   
  
...we've not only turned on to track our SQL, but we've also turned on a Hibernate config option to track SQL comments. We say use\_sql\_comments is true, and as we look through our stack trace, you can see on all of our insert statements, it gives a very generic, "hey, I'm inserting data to this Hibernate package, domain package class called page," inside of here for all of our comments. And then when we do our query, it says "I am searching!!" and so the "I am searching!!" on our query comes from how we specify that query in our work; and our query is specified here. So just a generic, good old query, where I'm doing a distinct on page looking for words, nothing very interesting there, or maybe very interesting. But what we're talking about here is this setComment, and this can be done either in a HQL query for a Hibernate createQueryCall, or a criteria object search where I do session.createCriteria, both of them have the method on there to set comment.   
*In the hibernate.cfg.xml file, the presenter scrolls to specific lines:  
<!-- Echo all executed SQL to stdout -->  
<property name="show sql">true</property>  
  
<!-- Enables adding of comments to SQL when generated -->  
<property name="hibernate.use\_sql\_comments">true</property>  
  
In the Console, the presenter scrolls to the line:  
Hibernate: /\* insert hibernate.domain.Page \*/ insert into Page (creationDate, htmlIncluded, lastModifiedDate, pageNumber, stor...  
  
Then he highlights the line:  
Hibernate: /\* I am searching!! \*/ select distinct story0\_.id as id1\_1\_, story0\_.creationDate as creation2\_1\_, story0\_.lastModi..  
  
The presenter opens the QueryingMain.java file and highlights the line:  
       Query q = session.createQuery("select distinct s from Story s join s.pages as p where p.words like :topic");  
  
He also highlights the line:  
       q.setComment("I am searching!!");*   
  
And so we see here the text "I am searching!!" shows up as the comment here. So the advantage of this is my database administrators, as they go back and look at the logs, have the ability to see the intent of the developers. I can say what am I searching on, what is my goal here, and I can even tag these appropriately. They can search through logs quicker by putting special tags inside of there that might enable my debugging to go much quicker, or my data analysis to go much quicker. So Hibernate gives us this great advantage to keep comments, whether it's in our table creation for DDL, or if it's within our search queries to be able to know the intent, not only of what the query is doing, but what we're hoping the query was doing, or any other metadata we wish to add.

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a03_it_enus/output/html/course_transcript.html#top)

Using @Sort for Java-based Sorting

Learning Objective

*After completing this topic, you should be able to*

* *use Java-based methods to arrange search results from a database using Hibernate*

**1. Practising @Sort for Java-based Sorting**

As I'm getting data back from the database, I of course have the option to order that data coming back, using Order By clauses, or what not in my queries. Hibernate doesn't limit you to that though, because Java is just as equally good at sorting, as the database is. So in this case, I have an example where I have a newspaper that loads back a bunch of stories. So if I run my demo for here, it's going to get the stories back, and they're just going to be in a hodgepodge of order, whatever they happen to be coming back, from the query. But I know traditionally, when I look at a newspaper I want to see the stories by a date. If I'm going to the web site of a newspaper, I want to look at the most recent stuff first, not just a random order of how they come up. So I have the option inside of Hibernate, to make that sort happen on the Java side, and I can do that through the @Sort annotation. Now the @Sort annotation, actually is deprecated, and so the @Sort annotation was replaced by two other annotations: @SortComparator, and @SortNatural. Which is a very logical way of sorting, because it bases that on the Java sorting approach. So in Java, I use the Comparator, and Comparable interface. The Comparator is between two objects, the Comparable is an object that is comparing in inside of itself.   
*An instance of Java - Eclipse is open.   
  
There are four open files: NewspaperMain.java, Newspaper.java, Newspaper.hbm.xml, and StoryByDate.java. The Newspaper.java file is open. It contains the code:  
  
@Basic  
private String edition;  
  
@Temporal(TemporalType.DATE)  
private Date releaseDate;  
  
@manyToMany  
@JoinTable(name="includedStories"'  
          joinColumns={@JoinColumn(name="newspaperId:)},  
          inverseJoinColumns={@JoinColumn(name="storyId")}),  
private Set<Story> stories;  
  
public long getId()  
{  
       return id;  
}  
  
public void setId(long id)  
  
The presenter clicks Run and the output displays in the Console as an extensive list of stories in no particular order.  
  
In the Newspaper.java file, the presenter types @Sort and a pop-up box appears with two available options: @SortComparator, and @SortNatural.*   
  
So if I have an object that is comparating inside of itself, I have the comparable implementation, I can do Natural. And if I do Natural, I've nothing else to do, it's going to use the Comparator naturally in the object to sort my set, my collection, my list, whatever's coming up here. A bit more interesting, I'm going to show you the @SortComparator, in which case I can change the sorting order by applying a external comparison. So I have a class: StoryByDate, that I can set to be the Comparator here. And so here is my Comparator, it's going to compare two stories, and it's going to order them appropriately – and we'll come back to that in a second. But now once I've set my @SortComparator out here, by just doing that simple annotation I can re-run my solution, and as I get stories back, notice they're now going to be all sorted in some sort of date order. They're going to come up, starting here and going from high to low, in the date order. Now the ordering is based off of the solution I put inside of my Comparator.   
*The presenter selects @SortNatural and then updates it to the @SortComparator. He types () and a pop-up menu appears where he selects value and the presenter continues to type the added line. The updated section code is now:  
  
@ManyToMany  
@JoinTable(name="includedStories"'  
          joinColumns={@JoinColumn(name="newspaperId:)},  
          inverseJoinColumns={@JoinColumn(name="storyId")}),  
@SortComparator(value=StoryByDate.class)             
private Set<Story> stories;  
  
The presenter switches to the StoryByDate.java file, which has the code:  
  
package hibernate.domain;  
  
import java.util.Comparator;  
  
public class StoryByDate implements Comparator<Story>  
{  
  
@Override  
public int compare(Story o1, Story o2)  
{  
       int answer = o2.getStoryDate().compareTo(o1.getStorydate());  
       if (answer == 0)  
       {  
               return (int) (o2.getId() - o1.getId());  
       } else  
       }  
               return answer;  
       }  
  
The presenter switches to the NewpaperMain.java file and clicks Run. The Console displays the output, with articles sorted by date.*   
  
So my Comparator – in this case – is comparing two story dates. So hey, I got your StoryDate, I got my StoryDate. The first thing I… the first choice I'm making it right at this Comparator is, do I want it to be in ascending order, or descending order? Again, this is Java solutioning, it's not like the database. I can just simply change the order by switching the order in which I compare them. If I do o2 first, I get descending, if I get o1 first, I get ascending. The other little trick inside of here is, since I'm in a set, I have to actually compare deeper than just the dates. Because the set is going to use this comparator to whether or not I'm going to keep it there. So if I comment out a little bit of code here, excuse me, and I go and I run my solution, I actually end up with a tiny little list of stories that gets returned. I only have a handful of stories here, and notice it's one story per day. Since I'm dealing with a set, I have to be careful how I'm using my Comparator. And so what this extra code inside of here is doing, is it's not only sorting by date, but it's additionally sorting by ID. So if my date comparison returns back zero, as these two items happened on the same date, then I'm going to sort them by ID to ensure they all show up. So by keeping this code in my Comparator, (I apologize I hit the wrong button there again), by keeping this code in the Comparator...   
*The presenter switches to the StoryByDate.java file and updates a few lines by adding forward slashes at the beginning of the lines. The updated section is:  
@Override  
public int compare(Story o1, Story o2)  
{  
       int answer = o2.getStoryDate().compareTo(o1.getStorydate());  
//        if (answer == 0)  
//        {  
//                return (int) (o2.getId() - o1.getId());  
//        } else  
//        }  
//                return answer;  
//        }  
  
He switches to the NewpaperMain.java file and clicks Run. He scrolls through the output which shows a short list of stories.  
  
The presenter goes back to the StoryByDate.java file and updates the code to:  
@Override  
public int compare(Story o1, Story o2)  
{  
       int answer = o2.getStoryDate().compareTo(o1.getStorydate());  
       if (answer == 0)  
      {  
               return (int) (o2.getId() - o1.getId());  
       } else  
       }  
               return answer;  
       }  
  
He switches to the NewpaperMain.java file, runs the code, and scrolls to the output which shows all the stories by id.*   
  
Then it's going to end up with a final solution that sorts by date, and it keeps all the stories included inside of there…I forgot to save it in additionally. And so I get all the solution down, and out from that. The last note inside of there is, you're not limited to annotations. You can also do the same thing in stories…in XML version of the stories, by adding a @Sort to your set list map, or whatnot, inside of there when you're mapping collections. And so we can add the @Sort order, and we can add the Comparator inside of there, or we can set it to be unsorted, or we can set it to be natural sorted. Those are all options inside of there. So on the Java side, I can still do sorting, I can still manage the presentation or the ordering of my data pulled from the database, without having to resort to Order By or other clauses in the database.   
*The presenter switches to the Newspaper.hbm.xml file and highlights the line:  
  
       <set name="stories" table="IncludedStories" sort="hibernate.domain.StoryByDate">*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a03_it_enus/output/html/course_transcript.html#top)

Using @JoinTable

Learning Objective

*After completing this topic, you should be able to*

* *create a many-to-many relationship using @JoinTable annotation in Java Hibernate*

**1. Adopting @JoinTable annotations**

One essential design approach inside of databases, is the old-fashioned LinkTable, and so here's an example, we can look at where we have a newspaper class, which defines some edition of a newspaper that's going to come out. And that needs to link to all the different stories that are going be a part of that edition. And so, between the newspaper class and the story class, I have the includedstories table here, which points to which edition of the newspaper it's going to point to, and which edition of the story it's going to point to. So it's going to tie stories to the newspaper. And so this is your traditional many-to-many relationship, where there are databases or objects are involved. And it's just a very temporary tie between these – we don't have to have the story a part of the newspaper – and the newspaper story could be part of many editions. We're not limited either way, but we want to tie those together. So Hibernate gives us a very elegant way to be able to link between these different arrangements, because in Java, in your object model, I would just simply have a list, or a set, or a map, or some collection of stories, that I would tie to. I don't need a LinkTable inside of there, that is already a temporary relationship between those objects. And so in Hibernate, I can define it as being a many-to-many relationship, using additionally the @JoinTable tag the annotation inside of here to describe that relationship.   
*An instance of Command Prompt - mysql - u root - p is open. It displays three tables. Each table has six columns: Field, Type, Null, Key, Default, and Extra.  
  
The first table is the output for the command: describe newspaper;  
  
The table has three rows. For row one:  
  
Field: id.  
Type: bigint<20>  
NULL: NO.  
Key: PRI.  
Extra: auto\_increment  
  
The line of output after the table is: 3 rows in set <0.01 sec>  
  
The second table is the output for the command: describe story;  
  
The table has five rows. The output for row one is:  
  
Field: id.  
Type: bigint<20>  
NULL: NO.  
Key: PRI.  
Extra: auto\_increment  
  
The last line of output after the table is: 5 rows in set <0.00 sec>  
  
The third table is the output for the command: describe includedstories;  
  
The table has two rows. The output for row one is:  
  
Field: newspaperid.  
Type: bigint<20>  
NULL: NO.  
Key: PRI.  
Extra: blank  
  
The line of output after the table is: 2 rows in set <0.02 sec>  
  
The presenter opens Java-Eclipse. Two files are open: two NewspaperMain.java and Newspaper.java. The Newspaper.java file is selected. It contains the code:  
  
@Basic  
private String edition;  
  
@Temporal(TemporalType.DATE)  
private Date releaseDate;  
  
@ManyToMany  
@JoinTable(name="includedStories",  
          joinColumns={@JoinColumn(name="newspaperId")},  
          inverseJoinColumns={@JoinColumn(name="storyId")})  
private Set<Story> stories;  
  
public long getId()  
{  
       return id;  
}*   
  
So old-fashioned Hibernate I might be required to actually create a third-party class, to link these two other classes – the newspaper and the story – together. But now I can define the includedStories table as part of this @JoinTable. And then within my queries then, I can define which direction each of these goes, so my @JoinColumn pointing back to myself as an object is the "newspaperId". The inverseJoinColumn pointing to the stories I want to link to, is the joinId. And so we can see, these are the columns from our database. This is the newspaper I'm trying to load from, this is the stories I'm trying to load to. And so any number of stories, can be part of any number of editions, and vice versa, within this table. So executing this solution, going through and being able to run our little test here, it'll go through and load up a bunch of stories, put them into the newspaper. And so if we look through our collection here, we're inserting a collection into the different included stories that's going on here. We're going to include that in, and then we're going to select those back in. So all the editions that are part of...all the stories that are part of this edition...get loaded and shown to us.   
*The presenter clicks Run, switches to the NewspaperMain.java file, clicks Run again, and examines the output in the Console. He highlights the line:  
  
Hibernate: /\* insert collection row hibernate.domain.Newspaper.stories \*/ insert into includedStories (newspaperId, storyId)  
  
He scrolls through the output where an extensive list of stories appears.*   
  
Now the more interesting side here, the demo is going to look as we go back to the database, and so I can do a select here, on all of the newspaper stories. So I just added a story into the newspaper here, and as I added the story into the newspaper, I also get this LinkTable populated. So all those stories that are already out there and populated, I'm loading them up. I'm putting them into my newspaper object and that code is pretty simple inside of here. So I'm doing a query to load all the stories, and then I'm setting inside of the newspaper object, all those stories to link inside of there. And then Hibernate's kind enough for us to go and populate that entire LinkTable. So I can manage my many-to-many relationships, whether it's one directional or bidirectional, inside of Hibernate using both the many-to-many tag inside of the annotation, and the @JoinTable to make those things happen simply and easily, and not having to create a third-party table.   
*The presenter returns to the Command Prompt and runs the command:  
  
select \* from newspaper;  
  
The output is a table with three columns and one row:  
id: 1.  
edition: Hot off the Presses  
releaseDate:2014-02-24.  
  
The line of output below the table is: 1 row in set <0.00 sec>  
  
Next he runs the command:  
  
select \* from includedstories;  
  
The output is a table with two columns and 100 rows. Each row has a value of 1 in the newspaperId column. The storyId column has rows with values from 1 to 100.  
  
The line of output below the table is: 100 rows in set <0.00 sec>  
  
The presenter goes back to the NewpaperMain.java file and scrolls to the section of code:  
  
Query q = s.createQuery("from Story");  
List<Story> stories = (List<Story>) q.list();  
paper.setStories(stories);  
  
In the Newspaper.java file, the presenter highlights the line:  
  
@JoinTable(name="includedStories",  
  
He hovers over @JoinTable where a pop-up displays:  
  
@ javax.persistence.JoinTable  
  
@Target(value={FIELD,METHOD})  
@Retention(value=RUNTIME)  
  
Note: This element neither has attached source nor attached Javadoc and hence no Javadoc could be found.  
Press F2 for focus*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a03_it_enus/output/html/course_transcript.html#top)

Using many-to-many Relationships

Learning Objective

*After completing this topic, you should be able to*

* *set up a many-to-many relationship in Hibernate*

**1. Setting up many-to-many relationships**

Within our database design, we often have the need to link together two different classes or two different tables to be able to have those data rows interconnect, but not necessarily have it be part of the same class. That's where link tables come in. So we have a design here where we're talking about linking newspapers to stories. And so stories might appear in many editions of a newspaper and a newspaper edition obviously has many different stories. And since it's such a temporary relationship, we're going to create this link table here where from our stories and our newspapers, were going to use the link table to tie them together. So inside here we're going to say which edition, in which newspaperId does it point to or which stories does the newspaper point to. And they go back and forth, and neither table and neither class is going to be affected by this. Now the Java way of representing this relationship is really easy. I can just create inside of my Newspaper class, a set, a list, a map, some collection of stories that I'm going to tie to. And it can be one directional or it can be both directional…both directions. The newspaper can keep track of the stories and the stories can choose to keep track of the newspaper or not. In our model here, we don't care about it. We're looking at a single direction of this relationship.   
*An instance of Command Prompt - mysql - u root - p displays.  
  
The first command is:  
  
describe newspaper;  
  
The output is a table with six columns and three rows. The values for the first row are:  
  
Field: id.  
Type: bigint<20>.  
NULL: NO.  
Key: PRI.  
Default: Null.  
Extra: auto\_increment.  
  
The line of output after the table is: 3 rows in set <0.01 sec>  
  
The next command is:  
  
describe story;  
  
The output is a table with six columns and five rows.  
The values for the first row are:  
  
Field: id.  
Type: bigint<20>.  
NULL: NO.  
Key: PRI.  
Default: Null.  
Extra: auto\_increment.  
  
The last line of output after the table is: 5 rows in set <0.01 sec>  
  
The third command is:  
  
describe includedstories;  
  
The output is a table with six columns and two rows.  
The values for the first row are:  
  
Field: newspaperId.  
Type: bigint<20>.  
NULL: NO.  
Key: PRI.  
Default: Null.  
Extra: blank.  
  
The line of output after the table is: 2 rows in set <0.01 sec>  
  
The presenter navigates to the instance of Java - Eclipse that is open with three files: NewspaperMain.java, Newspaper.java, and Newspaper.hbm.xml. The Newspaper.java file is selected. It contains the code:  
  
package hibernate.domain;  
  
import java.util.Collection;..  
  
public class Newspaper  
{  
       private long id;  
       private String edition;  
       private Date releaseDate;  
       private Set<Story> stories;  
         
       public long getId()  
       {  
               return id:  
       }  
         
       public void setId(long id)*  
  
But the question is inside of our XML mappings of Hibernate, how do I represent that link table? Now the old way, you might be required to actually create a third-party table to represent just this included story table and that doesn't make sense. So what we do instead in Hibernate, is we can map that set as part of our XML mappings. And so we can map the name here stories to point to this method here, this attribute here, stories and then the get story and Set<Story> that's part of the Java class. Now inside of the XML mapping we can then tie this to a table IncludedStories. And so we don't have to create another class to represent it, we're going to represent this table as part of this relationship between these two objects. The important thing though is we need to map keys each direction. They most likely have a foreign key relationship, not required to, but they might have that, and they probably should have that. But the key column here represents as part of my class, the Newspaper class, what is my ID that, that included story tables points to, so that's the newspaperId. I am the newspaper, I have the ID and there's my newspaperId, that's tied into that.   
*The presenter opens the Newpaper.hbm.xml file with the code:  
  
<hibernate-mapping package="hibernate.domain">  
       <class name="Newspaper" table="Newspaper">  
               <id name="id" column="id" type="long">  
                       <generator class="native"/>  
               </id>  
               <property name="edition"/>  
               <property name="releaseDate" type="date"/>  
                 
               <set name="stories" table="IncludedStories">  
                       <key column="newspaperId"/>  
                       <many-to-many column="storyId" class="Story"/>  
               </set>  
       </class>  
</hibernate-mapping>*   
  
And then on the other class, I'm using the many-to-many XML tag here to represent...this is the other class I'm going to point to. And it's pointing to the column storyId, which we saw over here in our database, in our schema over here for our database. And that is pointing to the Java class="story", now notice in this case, I could put the whole package out here. I don't have to put the whole package out here because I defined the package; all the stuff lives in up here as part of my XML mapping. So I could say hibernate.domain.story here or I can take advantage of the fact that it's already defined for me. So this mapping right here will then allow me to go through and run my test. And so when I go and load up that newspaper story, I get all the stories that are associated with that. And in this case, the data if I look at it, I can look at the newspaper, there's my hot off the press's edition of my newspaper, and I've linked to all one hundred stories that's in my database right now.   
*The presenter switches to the NewspaperMain.java file, clicks Run, and scrolls through the output which lists multiple stories. The presenter maximizes the Command Prompt instance and runs the command:  
  
select \* from newspaper;  
  
The output is a table with three columns: id, edition, and releaseDate.  
  
The data for row one is:  
  
id: 1.  
edition: Hot off the Presses.  
releaseDate: 2014-02-24.  
  
The line of output below the table is: 1 row in set <0.00 sec>  
  
The presenter runs the command:  
  
select \* from includedstories;  
  
The output is a table with two columns: newspaperId and storyId. There are 100 rows each with the value of 1 in the newspaperId column. The storyId column has rows with values from 1 to 100. The line of output below the table is: 100 rows in set <0.00 sec>*   
  
So for newspaperId, number one, I point to story number 1, 2, 3, 4 all the way down to 100. And so Hibernate's giving us a very slick and easy with the set tag and the <many-to-many> tag to link up our example here. Now we're not limited to just the set tag here. We can use the Hibernate map tag, the list tag, the bag tag, all these different options inside of here. And you can see the bag, map, list. All these different things can take advantage of the key column to point to myself and the <many-to-many> column to point to some other class and be able to manage this relationship. And so I can do that with just simple mappings, not having to create extra classes or extra annotations or extra XML mappings or anything inside of there, just simply using the <many-to-many> tag.   
*The presenter returns to the Newpaper.hbm.xml file in Java and adds a line of code above the line:   
              <set name="stories" table="IncludedStories">  
  
He types <bag, then deletes it and types <map, deletes <map, and types <list  
  
The presenter deletes the line.*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a03_it_enus/output/html/course_transcript.html#top)

The @Transient Annotation

Learning Objective

*After completing this topic, you should be able to*

* *set up the @Transient annotation within Java Hibernate*

**1. Setting up @Transient annotations**

When mapping through annotations, there's one little tweak that has to be understood. It doesn't exist in XML mapping, but you need it, and you need to understand it as far as annotations go, and it's the @Transient field. So our @Transient field here refers to an attribute and a class that is not supposed to be persistent in the database. And you would think, well the Hibernate will just ignore if it's not out there, but that's not the way it works. So for instance if I take out @Transient on my countNotPersisted, it's some count that has to do with real-time execution, I don't need to save it, I don't care about it, it has to do with each execution for whatever reason. If I go through and run this, if I go through and run the main application here, it's going to fail and you can see it says Unknown column 'countNotPersisted' in 'field list'. Because it has gone through and attempted to map that over, and insert those values with something called countNotPersisted. I didn't map it, I didn't tell it was a basic attribute, but Hibernate kind of insists on doing it anyway.   
*An instance of Java - Eclipse is open with three files: GenericMain.java, Generic.java, and hibernate.cfg.xml.  
  
The Generic.java file contains the code:  
  
private long id;  
  
@Basic  
private String text;  
  
@Basic  
private Date aDate;  
  
@Basic  
private double sampleMoney;  
  
@Basic  
private boolean aBoolean;  
  
@Transient  
private int countNotPersisted;  
  
public long getId()  
{  
  
The presenter removes the line:  
@Transient  
  
The presenter switches to the GenericMain.java tab, clicks Run, and scrolls through the output in the Console. He highlights the line:  
  
Caused by: com.mysql.jdbc.exceptions.jdbc4.MySQLSyntaxErrorException: Unknown column 'countNotPersisted' in 'field list'  
  
The presenter scrolls to another section of the output and highlights the line:  
  
\* insert hibernate.domain.Generic \*/ insert into Generic (aBoolean, aDate, countNotPersisted, sampleMoney, text, id) values (?, ?, ?, ?, ?...*   
  
Even stranger behavior is I have my auto table management turned off. If I turn it on and allow it to be in create mode or even update mode for that matter. And I run this guy again, it does something a little bit crazier, it will actually go through your table. And it will do an alter of your existing table, we can actually go through and see it right here. It will update the table to include that extra column. So it didn't fail this time, but if I go to my database and I show the table, it's actually gone through and added in a column countNotPersisted. And it's put in that default value of 0 for one object and the value we set into it of 5 for the other object. So in order for me to have everything shiny and happy and work properly, I absolutely must go through in my mapping and mark any attributes that I don't want to have to saved in the database as being @Transient when dealing with annotations. Again from doing XML mappings, it doesn't get that intrusive into my object.   
*The presenter switches to the hibernate.cfg.xml file which has the code:  
  
       <property name="current\_session\_context\_class">thread</property>  
         
       <!-- Disable the second-level cache -->  
       <property name="cache.provider\_class">org.hibernate.cache.internal.NocacheProvider</property>  
         
       <!-- Echo all executed SQL to stdout -->  
       <property name="show\_sql">true</property>  
         
       <!-- Enables adding of comments to SQL when generated -->  
       <property name="hibernate.use\_sql\_comments">true</property>  
         
       <!-- Option to sync the Hibernate schema to your database -->  
<!--            <property name="hbm2ddl.auto">create</property> -->  
  
       <!-- Used to save occasionally when doing large inserts of data -->  
       <property name="hibernate.jdbc.batch\_size">10</property>  
         
       <mapping class="hibernate.domain.Story"/>  
  
The presenter highlights the line:  
<!--            <property name="hbm2ddl.auto">create</property> -->  
  
The presenter  updates the line by tabbing the line to line up with the rest of the code. He switches to the GenericMain.java tab, which displays:  
{  
       Generic g = new Generic();  
       g.setId(1);  
       g.setText("Full Object");  
       g.setaBoolean(true);  
       g.setaDate(new Date());  
       g.setSampleMoney(199.99);  
       g.setCountNotPersisted(5);  
       s.save(g);  
         
       g.setText("Update Full Object");  
       s.update(g);  
}  
  
for (Object o : s.createCriteria(Generic.class).list())  
{  
       Generic g = (generis) o;  
       System.err.printIn(g.getId() + " " + g.getText());  
}    
  
The presenter clicks Run and scrolls through the output in the Console:  
  
Hibernate: /\* criteria query \*/select this\_.id as id1\_0\_0\_, this\_.asBoolean as aBoolean2\_0\_0\_, this\_.aDate as aDate3\_0\_0\_, this\_.countNot...          
0 Simple Object  
1 Update Full Object  
  
The presenter scrolls to a different section in the GenericMain.java file:  
  
Session s = factory.getCurrentSession();  
s.beginTransaction();  
  
{  
       Generic g = new Generic();  
       g.setText("Simple Object");  
       s.save(g);  
}  
  
{  
       Generic g = new Generic();  
       g.setId(1);  
       g.setText("Full Object");  
       g.setaBoolean(true);  
       g.setaDate(new Date());  
       g.setSampleMoney(199.99);  
       g.setCountNotPersisted(5);  
       s.save(g);  
  
The presenter maximizes the instance of Command Prompt - mysql - u root - p and runs the command:  
  
source showGenerics.txt;  
  
The output is a table with six columns: id, aBoolean, aDate, countNotPersisted, sampleMoney, and text. There are two rows.  
  
The values for row 1 are:  
  
id: 0.  
aBoolean: blank.  
aDate: NULL.  
countNotPersisted: 0.  
sampleMoney: 0.  
text: Simple Object.  
  
The  lines of output after the table are:  
  
2 rows in set <0.00 sec>  
Empty set <0.00 sec>  
Empty set <0.00 sec>  
Empty set <0.00 sec>*   
  
It will just skip anything that's out there, there is no notion of @Transient inside of XML mappings. But as far as we get into annotations we absolutely, absolutely must deal with that, otherwise we end up updating our database and whatnot. And so again, if we rollback, we put everything back the way we started with, the we wanted to begin with. Because let me tell you, the tendency in Hibernate after sometime, if you don't understand something, is just change things until the error goes away. Because Hibernate can be frustrating like that. But the proper answer here for our solution is just simply to make the attributes that are not being used @Transient and that will keep them and our database happy as we go through and do our solutions. So again you can see, we are back to not having a extra column here and everything is persisting just fine, by the simple use of the proper @Transient.   
*The presenter navigates to the hibernate.cfg.xml file and updates the line so it no longer lines up with the rest of the code:  
  
<!--            <property name="hbm2ddl.auto">create</property> -->  
  
The full code is:  
       <property name="current\_session\_context\_class">thread</property>  
         
       <!-- Disable the second-level cache -->  
       <property name="cache.provider\_class">org.hibernate.cache.internal.NocacheProvider</property>  
         
       <!-- Echo all executed SQL to stdout -->  
       <property name="show\_sql">true</property>  
         
       <!-- Enables adding of comments to SQL when generated -->  
       <property name="hibernate.use\_sql\_comments">true</property>  
         
       <!-- Option to sync the Hibernate schema to your database -->  
<!--            <property name="hbm2ddl.auto">create</property> -->  
  
       <!-- Used to save occasionally when doing large inserts of data -->  
       <property name="hibernate.jdbc.batch\_size">10</property>  
         
       <mapping class="hibernate.domain.Story"/>  
  
He switches to the Command Prompt and runs the command:  
  
source setup.txt;  
  
The output displays multiple lines of Query OK, how many rows are affected, and the number of seconds it took for each query.  
  
The presenter switches back to the GenericMain.java file in Java and clicks Run.  
  
The output lines are:  
  
Hibernate: /\* criteria query \*/select this\_.id as id1\_0\_0\_, this\_.asBoolean as aBoolean2\_0\_0\_, this\_.aDate as aDate3\_0\_0\_, this\_.countNot...          
0 Simple Object  
1 Update Full Object  
  
The presenter switches back to the Command Prompt and runs the command:  
  
source showGenerics.txt;  
  
The output is a table with five columns and two rows. The columns are: id, text, aDate, sampleMoney, and aBoolean.  
  
The values for row 1 are:  
  
id: 0.  
text: Simple Object.  
aDate: NULL.  
sampleMoney: 0.00.  
aBoolean: blank.  
  
The  lines of output after the table are:  
  
2 rows in set <0.00 sec>  
Empty set <0.00 sec>  
Empty set <0.00 sec>  
Empty set <0.00 sec>*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a03_it_enus/output/html/course_transcript.html#top)

Debugging in Hibernate

Learning Objective

*After completing this topic, you should be able to*

* *debug exception messages within Java Hibernate*

**1. Debugging within Java Hibernate**

Hibernate helps to abstract many of the details of the database interactions away from everyday operations. But what do we do when it's not working? We need a few ideas where to start when things are going wrong. The purpose of Hibernate is to abstract you away from so much of the details of dealing with the database, but that might actually cause a little bit of issues while you're developing it. Because now you're trying to make everything work the way you want it to work, but at a distance. And so the first, really tough phase of getting your Hibernate perfect is getting your mappings together. And so as you're going through and you're setting up your code, and you run your code, you'll have points where you'll just have mysterious exceptions that pop up. And so debugging these exceptions can be tough, because in this case, I have an error that's showing up in my SQL side, it's saying, "this story\_page doesn't exist". What does that mean? Where does it come from? And so it becomes difficult to search, in some cases because if it's not a Hibernate exception...   
*An instance of Java - Eclipse is open and three files are selected: CustomSQL.java, CommandLine., and Story.java.  
  
The Story.java tab contains the code:  
  
                                @StoredProcedureParameter  
                               },  
                       resultClasses={Story.class}  
                       )        
})          
public class Story  
{  
       @Id  
       @GeneratedValue  
       private long  id;  
         
       @Basic  
       private String title;  
         
       @Basic  
       private Date storyDate;  
         
       @Temporal(TemporalType.DATE)  
       private Date  creationDate;  
         
       @Temporal(TemporalType.TIMESTAMP)  
       private Date  lastModifiedDate;  
         
       @OneToMany(cascade=CascadeType.ALL)  
       //JoinColumn(name= "storyId", updatable=false)  
       private List<Page> pages = new ArrayList<Page>();  
         
       public void addPage(Page page)  
       {  
               page.setStory(this);  
               page.setPageNumber(pages.size() + 1);  
               pages.add(page);  
       }  
  
The presenter switches to the CommandLine... file, which has the code:  
  
package hibernate;  
  
import hibernate.crud.Create;..  
  
public class CommandLineMain  
{  
       private SessionFactory factory;  
         
       private Create          create;  
       private Read            read;  
       private Update          update;  
       private Delete          delete;  
         
public static void main(String[] args)  
{  
       new CommandLineMain().run();  
}  
  
public CommandLineMain()  
{  
       factory = ConfigHelper.getSessionFactory();  
       read = new Read(factory);  
       create = new Create(factory, read);  
       update = new Update(factory, create);  
       delete = new Delete(factory, create);  
}  
  
private void run()  
{  
       try  
       {  
               factory.getCurrentSession().beginTransaction();  
  
He clicks Run and switches to the Story.java file. The Console displays the output as an extensive list of exceptions. The presenter highlights the line:  
  
at:com.mysql.jdbc.exceptions.jdbc4.MySQLSyntaxErrorException: Table 'bignews.story\_page' doesn't exist*   
  
Doing a search on a SQL error isn't going to get you questions back from Hibernate. So part of it's going to be really paying attention to your resources, and understanding the fields. In this case I have a one-to-many relationship where I didn't define a JoinColumn, and so when I don't have a JoinColumn, it assumes there's a linkTable that's named from one class name, to the other class name –from story to page – and that's where it's coming up with that. So I actually have to add inside of here, the @JoinColumn, to tell this that, "hey, I'm not using a linkTable, I'm using a joinColumn instead". And that's going to have that pop up that way. And so putting that back in…then everything is going to run fine in this case. Now there's other cases where you get Hibernate exceptions, and you can go from that exception…you'll see now it's all working, it's all happy…there's other cases where you can get Hibernate exceptions, so you can from that exception directly and search it, Google, Bing…whatever your favorite search engine is. And it'll come up with a lot of interesting options. Now some of the options, some of the pages you'll hit, won't have anything interesting. Some of them will, and it's really a matter of passing through and learning more. The biggest thing is just learning more and more about these tags, and what they're trying to get you to do. And that'll help you understand what the net result is, and what's coming out from there.   
*In the Story.java file, the presenter removes the forward slashes from the line:  
  
       //JoinColumn(name= "storyId", updatable=false)  
  
An error icon appears in the line and the presenter adds the @ sign. The updated section is:  
  
       @OneToMany(cascade=CascadeType.ALL)  
       @JoinColumn(name= "storyId", updatable=false)  
       private List<Page> pages = new ArrayList<Page>();  
  
The presenter switches to the CommandLine...file and clicks Run. He scrolls through the output in the Console and examines the section:  
  
!!!  
Insert the story to read  
Hibernate: /\* insert hibernate.domain.Story \*/ insert into Story...  
Hibernate: /\* insert hibernate.domain.Page \*/ insert into Page...  
Hibernate: /\* insert hibernate.domain.Page \*/ insert into Page...  
!!!  
Call Get  
Hibernate: select story0\_id as id1\_3\_0, story0\_.create...  
  
!!!  
Show the story we just read  
  
No story loaded  
Test Complete  
!!!*   
  
There is sometimes, you know for instance the bidirectional relationship, if you don't map it properly, if you don't have your tags set up properly, then you'll have to do things like add updatable=false on one point. Because it's a bidirectional relationship, I only want it to be managed by one side or the other, as an instance here. My Story class shouldn't be updating the id's of the Page class in this case. So past that, sometimes I get my Hibernate all working fine and I'm still having issues in the database. One great example I came across with while playing in Hibernate, was dealing with stored procedures around...or even just straight queries...around custom SQL. So in my test where I'm doing custom SQL, in this specific example, I want to go off and run it – and this one's all up and working right now – but it's setting a series of parameters…I need to actually go through and refresh my database for this one. A great example there of getting an interesting null pointer exception – we'd have to go sort out where that's coming from, but anyway – when I'm doing this, it will go through and it sets up JDBC-like queries with a bunch of question marks inside of it. And Hibernate does not specify, or allow you to predict, or allow you to specify, the order of these parameters.   
*In the CommandLine file, the presenter scrolls to a section of code:  
  
//  Read Demo  
       read.run(Read.GET\_BY\_ID);  
//        read.run(Read.GET\_MISSING);  
//        read.run(Read.LOAD\_BY\_ID);  
//        read.run(Read.LOAD\_MISSING);  
//        read.run(Read.LOAD\_TO\_EXISTING);  
//        read.run(Read.REFRESH);  
//        read.run(Read.SHOW\_LAZY\_LOADING);  
  
  
//  Update Transient Demo  
//      update.run(Update.UPDATE\_FROM\_SCRATCH);  
//      update.run(Update.UPDATE\_WITHOUT\_ALL);  
  
  
//  Update Persistent Demo  
//      update.run(Update.LOAD\_AND\_UPDATE);  
//      update.run(Update.LOAD\_AND\_CALL\_UPDATE);  
//      update.run(Update.LOAD\_AND\_MERGE);  
  
  
//  Delete Demo  
//      delete.run(delete.DELETE\_A\_PAGE);  
  
He opens the CustomSQL.java file and scrolls through the query:  
{  
       factory = ConfigHelper.getSessionFactory();  
       Session s = factory.getCurrentSession();  
       s.beginTransactional();  
         
       StoryCustomStatements story = new StroyCustomStatements(...  
       story.setTitle("Insert using custom statements");  
       story.setLastModifiedDate(newDate());  
       story.setStoryDate(new Date());  
         
       Long id = (Long) s.save(story);  
         
       s.getTransaction().commit();  
       s = factory.openSession();  
       s.beginTransaction();  
         
       story = (StoryCustomStatements) s.get(StoryCustomStatement...  
       System.err.printIn(story.getTitle() + " " + story.getLast...  
       s.evict(story);  
         
       StoryCustomStatements updatedStory = new StoryCustomState...  
       updatesStory.setId(new Long(1));  
       updatedStory.setTitle("Update the story now");  
       updatedStory.setStoryDate(story.getStoryDate());  
         
       s.update(updatedStory);  
       s = factory.openSession();  
       s.beginTransaction();  
         
       story = (StoryCustomSatatements) s.get(StoryCustomStatemen...  
       System.err.printIn(story.getTitle() + " " + story,getLast...  
         
       s.delete(story);  
         
       s.getTransaction().commit();  
}catch (Throwable t)  
}  
  
The presenter clicks Run and he examines a section of the output code in the Console:  
  
java.lang.NullPointerException  
       at hibernate.CustomSQL.run(CustomSQL.java:55)  
       at hibernate.CustomSQL.main(CustomSQL.java:17)  
  
The presenter opens an instance of Command Prompt -mysql -u root -p where he runs the command:  
  
source setup.txt  
  
The output is:  
Query OK, 15 rows affected <0.53 sec>  
  
Query OK, 1 row affected <0.00 sec>  
  
Connection id: 545  
Current database: BigNews  
There are multiple rows showing Query OK with the number of rows affected and the time in seconds.  
  
The presenter switches to the CustomSQL.java file and clicks Run. He scrolls to the output:  
  
Hibernate: call specialStoryUpdateAnnotations(?, ?, ?, ?, ?)*   
  
What's happening here is I'm doing a custom update statement, so it's passing every single one of the attributes of the class over to my stored procedure in this case. It could also be a query, with a bunch of question marks. The issue is, as I'm writing it, I don't know which one's which. I don't know the order of parameters, and I don't know the types of parameters, as they're coming inside of there. And the only thing that's somewhat consistent is the id tends to come last inside of here, if there's an id involved. So I need to, basically be aware of what they are, and I can't do that, even right now, as I have in my solution, I have the SQL logging turned on, it doesn't get me all the way so, even though I have "show\_sql" >true. So in order to be able to debug this, the only real choice I have is to go turn on the full logging, to get in my case, log4j hooked up. And then I can turn on the trace logging here, of the types. And so the types are saying, "I want to also log the data that's involved inside of here." So if I go back...and let me actually just refresh my database to be safe...   
*From the Package Explorer, the presenter opens the hibernate.cfg.xml file. It contains the code:  
  
       <property name="connection.pool\_size"  
  
       <!-- SQL dialect -->  
       <property name="dialect">org.hibernat...  
  
       <!-- Enable Hibernate's automatic ses...  
       <property name="Current\_session\_conte...  
  
       <!-- Disable the second-level cache  
       <property name="cache.provider\_class"  
  
       <!-- Echo all executed SQL to stdout  
       <property name="show sql">true</prope...  
  
       <!-- Enables adding of comments to SQ...  
       <property name="hibernate.use\_sql\_com...  
  
       <!-- Required to sync the Hibernate s...  
<!--            <property name="hbm2ddl.auto">va...  
  
       <!-- Used to save occasionally when d...  
       <property name="hibernate.jdbc.batch\_...  
         
       <mapping class="hibernate.domain.Stor...  
       <mapping class="hibernate.domain.Page...  
       <mapping class="hibernate.domain.Stor...  
       <mapping class="hibernate.domain.Stor...  
       <mapping class="hibernate.domain.Stor...  
       <mapping class="hibernate.domain.Stor...  
       <mapping class="hibernate.domain.Publ...  
       <mapping class="hibernate.domain.News...  
  
He also opens the log4j.properties file from the Package Explorer. The code is:  
  
# Direct log messages to stdout  
log4j.appender.stdout=org.apache.log4j.Console...  
log4j.appender.stdout.Target=System.out  
log4j.appender.stdout.layout=org.apache.log4j...  
log4j.appender.stdout.layout.ConersionPattern...  
  
# Root logger option  
log4j.rootlogger=INFO, stdout  
  
# Log everything. Good for troubleshooting  
#log4j.logger.org.hibernate=INFO  
#log4j.logger.org.hibernate.SQL=trace  
#log4j.logger.org.hibernate.engine.query=trace  
#log4j.logger.org.hibernate.type=trace  
#log4j.logger.org.hibernate.jdbc=trace  
  
The presenter removes the # from the line:  
  
#log4j.logger.org.hibernate.type=trace  
  
He switches to the Command Prompt and runs the command:  
  
source setup.txt  
  
The output displays multiple rows showing Query OK with the number of rows affected and the time in seconds.*   
  
...and I'm going to go and run my little test case here again. And as I run it now with this full logging turned on, now I can actually see as it came through and had the ?, ?, ?, ?, it's binding in [TIMESTAMP], [TIMESTAMP], [TIMESTAMP], [VARCHAR], [BIGINT].  
So here's the id; in this case this is the title, this is the date of…that I'm using in a useful way. And then there's two other dates here, and if I go look at my stored procedure, I've actually updated this and I had to do this several times. I had to move around these parameter several times, where I got them where I wanted them to come in. And they have to be aligned, so that way they line up properly to the types that they're coming, and the order they're coming in. I don't know the algorithm by which Hibernate does that, I probably could the pull source code and figure that out. But in this case, I can just do it from the other end and just look at the data that's coming out and try and match to that.   
*The presenter navigates to the CustomSQL.java file, clicks Run, and scrolls to the output in the Console:  
  
Hibernate: call specialStoryUpdateAnnotations(?, ?, ?, ?, ?)  
07:04:31,000 TRACE BasicBinder:69 = binding parameter [1] as [TIMESTAMP] = [null]  
07:04:31,000 TRACE BasicBinder:69 = binding parameter [2] as [TIMESTAMP] = [null]  
07:04:31,000 TRACE BasicBinder:81 = binding parameter [3] as [TIMESTAMP] = [2014-02-26 00:00:00.0]  
07:04:31,000 TRACE BasicBinder:81 = binding parameter [4] as [VARCHAR] = [Update the story now]  
07:04:31,000 TRACE BasicBinder:81 = binding parameter [5] as [BIGINT] = [1]  
  
From the taskbar, the presenter maximizes an instance of procs in Notepad. It contains the code:  
  
drop procedure if exists specialStoryUpdate;  
  
delimiter #  
create procedure specialStoryUpdate(IN title VARCHAR(100),  
                                   In storyDate DATE,  
                                   In creationDate TIMESTAMP,  
                                   In lastModifiedDate TIMESTAMP,  
                                   In id int)  
proc\_main:begin  
       update story s set s.title = title, s.storyDate = storyDate, s.lastModifiedDate = CURRENT\_TIMESTAMP  
               where s.id = id;  
  
end proc\_main #  
delimiter ;*   
  
And so there's another example of the way I can put things together. The last big thing that comes up in Hibernate, is there's so many different little tweaks to how I manage the session. You have to really be aware of when am I accessing the session, when am I calling getCurrentSession, when am I calling openSession, when am I beginning the transaction, when am I committing the transaction. And, like in this example, I begin a transaction, I commit a transaction, then I open a new session. So I have to be careful that I do it consistently, within one method it's not so bad. But most of our designs are going to have multiple methods, so you have to have good strategies in managing the transaction. That's the last big area of issues that might come up. Because Hibernate will do all sorts of crazy things like, it will allow you to load something, commit it, and then if I don't manage the session properly, when I go...I'm sorry...save something, commit it and when I go to load it, it's not there. Because I am in a different session. Remember the session also provides a first-tier cache.   
*The presenter switches to the open CustomSQL.jav file.*   
  
Or the other thing that I've seen is I save something and it's successful, but I switch sessions and I don't commit the first one and thus it doesn't get saved in my database, when it's all said and done. So the program runs successfully, but my database isn't being updated. Hibernate's perfectly okay with you not committing a transaction, and essentially rolling it back when it exits. So you have to be careful of that as well. The last big thing is, in some cases Hibernate applications, as you go and run them, they won't terminate. They'll actually stick around and keep up and running, so you have to be very careful in your exception handling. You basically always want to surround your Hibernate work in some sort of exception handling, because otherwise there might be unreported exceptions that basically keep the Hibernate engine running, even though your application stops, and so your command line doesn't actually stop. On a server it's a little bit different, because the server apps tend to keep running, so you might not even notice; but for command line in particular, you want to be careful of that. So Hibernate can be a fantastic framework, a few tips on debugging can make it a little bit easier to work with. So look up some of these other tools, and some of these other techniques, and you can do a lot with Hibernate.   
*The presenter opens the DEBUG tab which displays:  
  
<terminated>CustomSQL [Java Application]  
       <terminated, exit value:0>C:\Program Files\Java\jre7\bin\javaw.exe (Feb26, 2014, 7:04:17 AM)  
  
He switches back to the Java - Eclipse instance where the CustomSQL.java tab remains open.*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a03_it_enus/output/html/course_transcript.html#top)

Improving General Performance

Learning Objective

*After completing this topic, you should be able to*

* *improve the overall performance of Java Hibernate*

**1. Improving overall performance**

By abstracting you away from the database, sometimes Hibernate does indeed affect performance. Now that can be because of Hibernate solution, in the way it's implemented, but it also can be because you haven't done the things you need to do to really tweak Hibernate to run properly, which you would normally do if you were building it by hand. So the first thing that I'll say is it's okay not to worry about prematurely optimizing your solution, it's not necessarily going to be slow, it might be okay. Most of the times we want to optimize the areas that are slow, the areas we really do need to address. And so it's okay to go through and tweak it later on, you just don't want to wait too long and then you want to update it based off of real needs. So we want to target our solutions, we want to recognize where we might have a slow query or a slow part of the model, or maybe there are wide spread problems on how we're doing some of the, the behaviors we have and so we want to get details, we're going to baseline what is happening right now. How much time it's taking, and then measure as we're doing performance improvements from there. So some of the general options we can do to improve performance in Hibernate is caching, one of them right there is caching. So caching allows us to, in the session, basically keep track of data that I'm using right there in that behavior.   
*Heading: Recognizing Issues.  
  
For premature optimization remember that not everything starts out perfect, and that is OK. And most times we only need to optimize when there are defined issues. For targeted solutions recognize if the problem is specific, for example; a slow query or a part of the object model; or if the problems are widespread. Get details by baselining the performance data and measuring how changes improve performance.  
  
Heading: General options.  
  
Caching is configured in session for small work and a secondary cache can be used for widespread data and queries. Caching also trades off resources for more reactive timing.*   
  
But by applying a secondary cache, I can provide widespread caching to be able to basically not have to hit the databases often. I'm trading off some of the runtime resources. I'm adding more memory and doing more processing in Java as opposed to having to hit the database. And so that caching can let me store data that's used frequently only on the server side and thus not have to hit the database, to decrease the performance on the database, and spend the time making that call. Past that another big area that can be an issue is fetching strategies. As I'm using collections, I have to choose the right data structure, should I be using sets, or lists, or maps, and I want to check how I'm going to link and load those appropriate objects. And so a lot of that comes down to lazy loading. I want to make sure that I'm loading the data I need when I need it, I don't want to have to make extra queries to the database for data I knew I needed right away. I don't want to have to make big returns, result sets back to me, if I'm not going to be using that data. And so I can set fetching strategies that are high level and then be able to customize those based off of query needs, but in general I can say okay be lazier, or be more eager, in our fetching and change it from there. And that can be done through config, meaning I don't have to change my code as much as well.   
*Heading: General options.  
  
Fetching strategies involves collections, which include what data structures we choose to use and how we link and load associated objects. Lazy loading has many alternatives. If it is not used, it can be slow on gathering too much data, and if used in the wrong place, it can lead to too many queries.*  
  
So past that I want to make sure I'm doing some of the things that just help databases in general. I want to, in my collection mapping, ensure that when I'm doing foreign key relationships that I'm using an index as well, this is called an indexed collection. And so as I'm doing those links between associated objects, I'm doing those searches on an index and not just a general table scan for that work. The indexes will also help within queries, I want to make sure I'm applying an index on things I'm querying on. I want to query mostly in indexes as I can and I also want to manage table sizes and accounts. Hibernate can allow you to do archiving fairly easily; I have those objects mapped and I can do queries to move those around. It's not about loading them into Java and putting them some place else, I can do subselects inside of Hibernate and I can move those things using batch job solutions that are provided within Hibernate. And so that'll keep my search space down and help me to basically keep my searches up quicker if possible. And so within that fetching strategy then, we can do lazy loading by default, but we need to make sure we don't need that extra data. If we know we're going to use that extra data we want to pull that up immediately, we want to configure it through the lazy load or eager load as we go and then we can tweak that strategy as it goes along.   
*Heading: Use Index Collections.  
  
Past a Foreign Key use index: this improves search speed on loading objects and this can be configured if Hibernate is creating the table, otherwise you must add this to the database DDL. Indexes help in queries too - don't forget to optimize them where appropriate. Table sizes count, though you may need an archiving strategy or some way to keep search spaces down.  
  
Heading: Fetching Strategy.  
  
With lazy loading, by default, objects are not loaded until used as this saves extra data when not needed and can reduce the number of sub-queries. We can configure not to lazy load if we know we are using data. We can tweak that strategy as there are multiple strategies available in how we access data and we can turn on and off strategies globally or within specific queries.*   
  
Using a fetching profile we can actually turn on and off the fetching strategy based off of the queries we have going on. So generally we can lazy load but if we know we're going to need some detailed data for a report or such, we can turn off that strategy and then reduce the number of hits into the database. So the biggest key that enables a lot of this performance strategy is monitoring. If we look at our session factory, there's actually some really basic built-in features as far as that goes. We can actually get individual data from the session factory and we can externalize that data as well using MBeans. So we will take a peek here in a minute add some of this data, it's pretty interesting but we can see statistics of how Hibernate is doing certain activities. We can try and get timing features of how long it's taking to do certain activities from the code we're already running; what better place to get it from if we know how to access that. So within that we will come up to our solutions here, and we can go to a piece of code and right here before we finish up our work inside of our session we can go to our factory, and we can ask it to get the statistics. And from there, we can have it log out a summary.   
*Heading: Monitoring  
  
Built in features are that the session factory keeps some basic statistics while we can get individual ones or log them as we wish. Externalized features involve Mbeans giving real time access to Java data, where we can configure Hibernate to externalize its statistics, thus testing or real time seeing how it is performing.  
  
The presenter opens an instance of Java - Eclipse, where three files are open:  CustomSQL.java, CommandLine.., and Story.java.  
  
The CustomSQL.java file contains the code:  
  
       story = (StoryCustomStatements) s.get(StoryCustomStatement...  
       System.err.printIn(story.getTitle() + " " + story.getLast...  
       s.evict(story);  
         
       StoryCustomStatements updatedStory = new StoryCustomState...  
       updatesStory.setId(new Long(1));  
       updatedStory.setTitle("Update the story now");  
       updatedStory.setStoryDate(story.getStoryDate());  
         
       s.update(updatedStory);  
       s = factory.openSession();  
       s.beginTransaction();  
         
       story = (StoryCustomSatatements) s.get(StoryCustomStatemen...  
       System.err.printIn(story.getTitle() + " " + story,getLast...  
         
       s.delete(story);  
         
       s.getTransaction().commit();  
}catch (Throwable t)  
}  
  
The presenter opens the CommanLine file and scrolls to the section of code:  
//  Delete Demo  
//        delete.run(Delete.DELETE\_A\_PAGE);  
//        delete.run(Delete.DELETE\_PARENT\_ONLY);  
//        delete.run(Delete.DELETE\_TRANSIENT);  
//        delete.run(Delete.DELETE\_PERSISTENT);  
  
       System.err.printIn("Test Complete");  
       System.err.printIn("!!!!!!!!!!!!!!!!!  
//        TestHelper.cleanup(factory.getCurrentSession());  
  
//        Session s =factory.openSession();  
//        s.beginTransaction();  
//        s.getTransaction().commit();  
       factory.getCurrentSesion().getTransaction().commit();  
         
   } catch (Throwable t)  
   {  
       t.printStackTrace();  
       factory.getCurrentSession().getTransaction().rollback(  
   } finally  
   {  
       // To kill the run else Hibernate leaves it active.  
       factory.close();  
  
He removes the lines:  
//        Session s =factory.openSession();  
//        s.beginTransaction();  
//        s.getTransaction().commit();  
  
He replaces the lines with:  
factory.getStatistics().logSummary();  
  
The full page displays as:  
  
//  Delete Demo  
//        delete.run(Delete.DELETE\_A\_PAGE);  
//        delete.run(Delete.DELETE\_PARENT\_ONLY);  
//        delete.run(Delete.DELETE\_TRANSIENT);  
//        delete.run(Delete.DELETE\_PERSISTENT);  
  
       System.err.printIn("Test Complete");  
       System.err.printIn("!!!!!!!!!!!!!!!!!  
//        TestHelper.cleanup(factory.getCurrentSession());  
  
factory.getStatistics().logSummary();  
       factory.getCurrentSesion().getTransaction().commit();  
         
   } catch (Throwable t)  
   {  
       t.printStackTrace();  
       factory.getCurrentSession().getTransaction().rollback(  
   } finally  
   {  
       // To kill the run else Hibernate leaves it active.  
       factory.close();*   
  
So in our test case right here, I'm going to through before it finishes up, and run this test case and spit out all the statistics that come up from that. And so this very simple little call allows me to see, and my test case isn't doing, anything very interesting, but what am I doing within the second level cache? How well am I using that? How well am I using my entities? How am I loading and fetching them? Collections...my associations that are going on inside of there. So again my test case isn't doing anything interesting and it ran very, very quick. But as we're doing more interesting work and things might be slower, I could see perhaps I'm not hitting my cache as much as I would like, perhaps I'm doing too much loading and unloading of collections. Whatever the performance issue might be, these statistics will allow me to see that. And if I don't have a server solution that I can monitor this externally using JMX, I can at least use this to log and see it on my own. So again a lot of options for being able to baseline and monitor performance. And that's the key around Hibernate, is understanding where you issues lie and what options you have to optimize that. Don't just say hey, I mapped it and I'm done, Hibernate doesn't always try and do the best performance it just tries to get your features and functionalities done for you and it's up to you to tweak it to due performance.   
*The presenter clicks Run and examines some of the output:  
  
No story loaded  
Test Complete  
07:23:34,652 INFO ConcurrentStatisticsImpl:720 - HHH000239: Second level cache puts: 0  
07:23:34,652 INFO ConcurrentStatisticsImpl:721 - HHH000237: Second level cache hits: 0  
07:23:34,652 INFO ConcurrentStatisticsImpl:722 - HHH000238: Second level cache misses: 0  
07:23:34,652 INFO ConcurrentStatisticsImpl:724 - HHH000080: Entities updated: 0  
07:23:34,652 INFO ConcurrentStatisticsImpl:725 - HHH000078: Entities inserted: 0  
07:23:34,652 INFO ConcurrentStatisticsImpl:726 - HHH000076: Entities deleted: 0  
07:23:34,652 INFO ConcurrentStatisticsImpl:728 - HHH000033: Collections loaded: 0  
07:23:34,652 INFO ConcurrentStatisticsImpl:729 - HHH000036: Collections updated: 0  
07:23:34,652 INFO ConcurrentStatisticsImpl:730 - HHH000035: Collections removed: 0  
07:23:34,652 INFO ConcurrentStatisticsImpl:731 - HHH000034: Collections recreated: 0  
07:23:34,652 INFO ConcurrentStatisticsImpl:732 - HHH000032: Collections fetched (minimize this):...*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a03_it_enus/output/html/course_transcript.html#top)

Improving Query Performance

Learning Objective

*After completing this topic, you should be able to*

* *improve the Query performance within Java Hibernate*

**1. Improving Query performances**

Most of your performance issues, anytime you're dealing with database, tend to come around specific queries or sets of queries, around problematic tables or associations between tables or things like that. So in Java we have two sides to the equation that can affect performance. We have to understand our Java object model. Our database model might not be our Java object model and that's okay, we want our Java object model to represent the application at hand. A database represents all data for all time, a Java model represents how I'm using that data right now. And so I want to map that model according to how I'm going to use it, but that mapping is going to drive how the database performs. And so I have to choose how I'm going to create my Java objects in the way that is optimized for my Java solution but also allows good performance from the database. On the database side, well the data lives there, and most of the work is done there. So we have to do both working efficiently within the database and you don't want to ask database to massive, massive amounts of processing. But also we want to limit the number of interactions with the database, because going from a Java solution to the database can be a very slow action comparatively to other things. We don't want to hit the database too often but we also don't want to do too much work while we're at the database. So as far as the Java model goes, one key thing is we build that Java model upfront and tend to fall in love with it.   
*Heading: Two Sides to the Equation.  
  
In Java, how the object model is defined determines what is loaded. The mapping is driven by the object model chosen as this may be the easier of the two to tweak. In the database model the data lives here and most of the work is done here. We need to both do the work efficiently on the database and limit the number of interactions to the database.*   
  
We think that once we've built some Java classes it's a great model and I don't want to touch it, if I touch it I've to change a lot of lines of code if I change the way something is represented, oh my gosh, look at all the retesting I have to do. Well you have to understand even though I have a great Java model it still might need tweaking, it still might need additions or rearranging or changing the data structures around for performance. And so I don't want to be stuck in a slow moving application just because I made a choice early on, on how I'm going to represent my model. Remember process are cheap the other thing we tend to do is think we have to have one ubiquitous model. I have a class mapping from this table to this class and thus it's out there, it must be that representation. And that's not true, it's neither true in Hibernate nor is it true in the real world. Data is represented in object models very differently over time, each use case has its own representation. So I want to make special use classes that matches that representation. I want to load the data I need for this use case. If I'm looking at a newspaper on a web site I don't see every page of every story right there in front of me. I don't see every word of every story, I often start with just the headlines and the first few words.   
*Heading: Target the Java Model.  
  
Don't fall in love because while you may have a great Java model, it may still need tweaking or additions for performance. Classes are cheap – add special use classes to match the data needed. Represent data to match how it is used and don't load more data than you need. Mapping is easy – don't worry about more, as this gives flexibility for tweaking.*   
  
And so I can load that data accordingly and doing that mapping is easy. I can add multiple mappings of any class out there for the different use cases I need at a very inexpensive cost of a couple of extra Java classes, and maybe a few extra methods inside of my data access object. Now on the flipside, I still have to be aware of database performance. So I want to make sure anytime I'm searching in Java, if I can get a primary key I want to search on it. I might need to cache IDs for later use. I put them in a session, I put those in some sort of thread local, some place where I might not need that data now, but I know I need at least the IDs to go look up later on. And particularly when I'm using web sites; if I'm going to put something on the web site that's going to get past to the user anyway it's going to be an index of sorts. I might as well put that primary key out there on the web site. When they pass it back to me I can then search on that primary key when I'm doing my querying. If I can't use a primary key, I at least want to try and query on indices. This is just normal SQL logic, normal SQL rules that make sense. And it's going to help our Hibernate performance as well. And as in other queries I want to watch how I'm doing joins. As I am associating classes back and forth it's a very cheap operation.   
*Heading: Database Tips.  
  
Use the primary key where you can – store it for later use and use it to index objects on web sites or such. Otherwise, query on an index – this helps in normal SQL and will help in Hibernate. Watch how you join in queries – working in classes does not change how joins happen. Review the SQL and seek help from experts if needed. Tweak HQL/Criteria to align to better practices, or use custom SQL.*   
  
Doing a setter, using lists and maps is very, very quick inside of Java. It's not always quick inside of the SQL, particularly depending on how classes are aligned and what are those indices and things like that. So I want to make sure the SQL that's being generated by Hibernate aligns the best practices. I can go to those helpers, those experts, those DBAs and see, "Hey this is what's generated, should I use that?" And thus I can tweak my HQLR, my criteria objects, to be optimal performing and worst case scenario, just use custom SQL. Use name statements, use stored procedures, whatever works out there to get my solution. And so I don't always have to use a Hibernate-provided solution, I want to use Hibernate to speed up my development. I want to use all of the queries that are out there and all the rules I can overlay, the filters, the transaction helpers, the interceptors, all that stuff is great for ensuring my business rules are consistently followed through my application. But I don't want to forget those other techniques.   
  
I want to let my DBAs come up with awesome queries and use name statements or XML mapping files to include those. And as I said, stored procedures are great. If I want to keep my business rules as much as possible my Java solution, but I can also externalize some aspects of that in the stored procedures when performance, particularly when I'm not moving data outside of the database. If I have just some sort of shifting of data in the database a stored procedure is a great way to not have to make multiple transactions and unnecessarily load data into your Java application and back to the database. So querying is such a key part of our application; optimizing each query and using some of these strategies, and maybe a little bit more research into these strategies will help you make a great performing in Java Hibernate application.   
*Heading: When Not To Hibernate.  
  
Generation is good – Hibernate helps speed up development and its rules can ensure integrity in the processing of data such as filters, interceptors, and transactions. But other techniques work too, such as names statements, which lets the DGAs optimize queries and we can still map results. For stored procedures, if business rules are less involved, this can be quicker, and is particularly good when just moving data.*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a03_it_enus/output/html/course_transcript.html#top)

Read Only Objects

Learning Objective

*After completing this topic, you should be able to*

* *use the immutability feature of Java Hibernate to make objects in your database read-only*

**1. Using immutability for read only objects**

Some database tables are out there for references purposes, for certain applications. There might be a rule out there that my application – my Hibernate, my Java – is not allowed to modify the values inside that database. Traditionally, making sure that that doesn't happen is up to either database permissions, or the discipline of developers to get their work done. But Hibernate gives me a very cool feature, called immutability, to make objects that cannot be modified in the database from your Java application. Now it's not saying they're immutable in Java, it's just making sure that they're immutable in the database. So here I have a class that represents a state, you know, just like whatever you happen to live, the state that you're in has a name, and an abbreviation inside of here. So I'm going to load up the states that maybe I'm using in a drop-down list, or something along those lines. And I want to make that class immutable. So while there is getter and setter methods that change the names and the abbreviations for states, I don't want to allow those changes to be reflected into the database, and Hibernate can help me with that. So once I've made it immutable, I can go through and I'm going to show you a test here. So I'm going to load in a list of states. Now just to show you here, I've got all the states there in my database, alphabetically speaking, and I can go through and load those. And for one random state I'm going to go off and choose, I'm going try and modify that state, saying "hey, they've left the union." I'm going to set the abbreviation to XX, and just take their name out of there.   
*An instance of Java - Eclipse is open and three files are open: ReadOnlyMain.java, State.java, and State.hbm.xml.  
  
The State.jave file contains the code:  
  
package hibernate.domain;  
  
import javax.persistence.Basic;..  
  
@Entity  
@Immutable  
public class State  
{  
       @Id  
       private long id;  
         
       @Basic  
       private String name;  
         
       @Basic  
       @Column(name='abbr")  
       private String abbreviation;  
  
The presenter opens the ReadOnlyMain.java file, which contains the code:  
{  
       factory.getCurrentSession().beginTrasaction();  
       Session s = factory.getCurrentSession();  
         
       Random r = new Random();  
         
       State state = (State) s.load(State.class, new Long(r.nextInt(49) + 1));  
       System.err.printIn("Changing " + state);  
       state.setName("left the Union");  
       state.setAbbreviation("XX");  
       s.update(state);  
       System.err.printIn("Change Called");  
         
       State newState = new State();  
       newState.setId(100);  
  
From the taskbar, the presenter maximizes an instance of Command Prompt - mysql -u root -p and runs the command:  
  
select \* from state;  
  
The output displays a table with a list of states in alphabetical order. The output line after the table is: 50 rows in set <0.00 sec>*   
  
And then I'm going to try and update that state into the database, and then go see what happened; I'm going to make that change call. Now when I do that change, there's going to be some interesting results that come by. Along with that, as we're doing our test, we're also going to try and insert a new state, because I want to show you that immutability is only changing objects, it's not actually limiting you from inserting new objects. So this feature is not a 100%, but it certainly is an improvement. So let me go ahead and run my example here, and as it does its execution, we've randomly picked the state of North Dakota. Again, just to be fair, no offense to the people of North Dakota out there, and we've modified that. So if we look at our list of states, when we get down to North Dakota here, it's been replaced in Java with an XX. Again, I can't stop Java from modifying objects through Hibernate, that's not what Hibernate's scope is, but if I look at my list of states here, and if I go to the great state of North Dakota, it is not modified in the database. We have not updated anything and Java has forbid...Hibernate has forbidden that from being updated.   
*In the ReadOnlyMain.java file, the presenter examines a section of the code:  
  
       State newState = new State();  
       newState.setId(100);  
       newState.setName("Serenity");  
       newState.setAbbreviation("FF");  
  
He clicks Run and examines a section of the output in the Console:  
  
Changing ND  
Change Called  
Add a new State  
ID = 100  
Hibernate: /\* insert hiberate.domain.State \*/ insert into state (name, abbr, id) values (?, ?, ?)  
Hibernate: /\* criteria query \*/ select this\_.id as id1\_6\_0\_, this\_.name as name2\_6\_0, this\_.abbr as abbr3\_6\_0\_ from state state0\_ where state0\_.id=?  
Test Complete  
!!!  
AL, AK, AZ, AR, CA, CO,CT, DE, FL, GA, HI, ID, IL, IN, IA, KS, KY, LA, ME, MD, MA, MI, MN, MS, MO, MT, NE, NV, NH, NJ, NM, NY, NC, XX, OH, OK, OR, PA, RI, SC, SD, TN, TX, UT, VT, VA, WA, WV, WI, WY, FF,.  
  
The presenter navigates to the Command Prompt and runs the command:  
  
select \* from state;  
  
In the output table he focuses on State 34, which is North Dakota, abbreviated to ND.*   
  
Now that being said, I was able to put the great state of Serenity into my database. So if I go back and look at this, you can see I've put an ID there, and so now people can be peaceful in a state of Serenity out there within the list of states. So that might be a limitation – something to be aware of – but it is something that does provide some restrictions to your coders and to your solution. Now I'm not limited to annotations, in annotations I can say this class is immutable. I can do the same thing in XML mapping as well, by going to that class definition that we'd have for state, and setting the attribute mutable=false. This has the same net behavior that's going on within there, and will limit the net result of not being able to update the object. Insertable's a different story, that's a separate feature that's out there, but I can certainly make at least read-only objects for key things like drop-down lists, or other aspects of my code where my domain model does not want me to be able to update that, but certainly needs to be able to read that and map to that from Hibernate.   
*The presenter scrolls to State 100, which is Serenity, abbreviated to FF.  
  
The presenter switches between the tabs and opens the State.hbm.xml file. He examines the section of code:  
  
<hibernate-mapping package="hibernate.domain">  
       <class name="State" table="state" mutable="false">  
               <id name="Id:/>  
               <property name="name"/>  
               <property name="abbreviation" column="abbr"/>  
       </class>  
</hibernate-mapping>*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a03_it_enus/output/html/course_transcript.html#top)

© 2016 Skillsoft Ireland Limited