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

**Course Transcript**

Customized Mapping and Basic Querying in Java Hibernate

**Inheritance**

| [1. Course Introduction](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a02_it_enus/output/html/course_transcript.html#t2) |

| [2. Inheritance – JOINED](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a02_it_enus/output/html/course_transcript.html#t6) |

| [3. Inheritance – TABLE\_PER\_CLASS](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a02_it_enus/output/html/course_transcript.html#t10) |

| [4. Inheritance – Single Table](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a02_it_enus/output/html/course_transcript.html#t14) |

**Identity Generation and Annotations**

| [1. Identity Generation in Annotations](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a02_it_enus/output/html/course_transcript.html#t19) |

| [2. Identity Generation in XML](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a02_it_enus/output/html/course_transcript.html#t23) |

| [3. Custom SQL in Annotations](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a02_it_enus/output/html/course_transcript.html#t27) |

| [4. Create an Index in Hibernate](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a02_it_enus/output/html/course_transcript.html#t31) |

| [5. Detailed Property Mapping with Annotations](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a02_it_enus/output/html/course_transcript.html#t35) |

| [6. Lazy Loading](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a02_it_enus/output/html/course_transcript.html#t39) |

**Customizing Mappings**

| [1. Defining with the Entity Annotation](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a02_it_enus/output/html/course_transcript.html#t44) |

| [2. The ID and Advanced ID Options](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a02_it_enus/output/html/course_transcript.html#t48) |

| [3. The Table and SecondaryTable Annotations](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a02_it_enus/output/html/course_transcript.html#t52) |

| [4. The class tag Options](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a02_it_enus/output/html/course_transcript.html#t56) |

| [5. Advanced XML Table Mapping](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a02_it_enus/output/html/course_transcript.html#t60) |

| [6. Custom SQL in XML Mappings](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a02_it_enus/output/html/course_transcript.html#t64) |

| [7. Detailed Property Mapping in XML](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a02_it_enus/output/html/course_transcript.html#t68) |

| [8. Embedded Objects with @EmbedDED or the component tag](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a02_it_enus/output/html/course_transcript.html#t72) |

**Querying**

| [1. Basic Querying – HQL](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a02_it_enus/output/html/course_transcript.html#t77) |

| [2. Basic Querying – Criteria](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a02_it_enus/output/html/course_transcript.html#t81) |

| [3. Advanced Query Management](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a02_it_enus/output/html/course_transcript.html#t85) |

| [4. Query by Example](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a02_it_enus/output/html/course_transcript.html#t89) |

Course Introduction

Learning Objective

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

* *start the course*

**1. Introduction to the course**

Java Hibernate enables you to map Java objects to database tables using annotations or XML files. Tony Lowe is an experienced software engineer and instructor and in this course he will discuss identity generation and annotations, and demonstrate how to customize mapping. He will also show you basic query using HQL – an SQL-like query language designed for Hibernate. Let's take a look.

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a02_it_enus/output/html/course_transcript.html#top)

Inheritance – JOINED

Learning Objective

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

* *apply the JOINED strategy when mapping Java classes to the database using Hibernate*

**1. Applying JOINED**

Inheritance is a powerful tool in Java, but it does not have an easy and immediate solution for mapping into the database. The JOINED table strategy uses a single table to represent each class in the hierarchy. Let's see how this gets mapped by Hibernate in between our Java classes and the database. There's three Inheritance strategies and we're going to focus on the JOINED inheritance strategy for this session. So our domain model in this case uses a base class of RealEstate. It's the idea, is I have always have a super class and some base classes, but how I mapped the databases is what's going to vary. So real estate is followed up by two classes, Commercial and Residential; and so they all have a street, city, state, zip, and a value. Commercial adds in some details about commercial, the zoning, the square foot, the parking. The Residential adds bedrooms, bathrooms, and square foot. Again square foot is chosen to be separate, maybe it's measured differently or something like that. It could be common, but we're not going to do that in this case. If I look at our database design here, for this strategy, we're going to choose within our database design to have each class inside of our hierarchy, have its own table. So the realestate class ties to the realestate table, the commercial class ties to the commercial table, the residential class ties to the residential table. Again the table names don't have to match, we're just choosing that for this example to keep it simple.   
*Hibernate is open and the interface is split into the Package Explorer, a documents view, with separate tabs for each code file, and the feedback panel containing a number of tabs including Problems, Javadoc, Declaration, Search, and Console.  
  
The files that are open are RealEstate.java, Commercial.java, Residential.java, RealEstate hbm.xml, and Main.java.  
  
RealEstate.java is selected and contains the code:  
  
package hibernate.domain.individual;  
  
public class RealEstate  
{  
   private long Id;  
   private String street;  
   private String City;  
   private String state;  
   private String Zip;  
   private String value;  
     
   public long getId()  
   {  
       return id;  
   }  
   public void setId(long id)  
   {  
       this.id = id;  
   }  
   public String getStreet()  
   {  
       return street;  
  
The presenter selects the Commercial.java file. The document contains the code:  
  
package hibernate.domain.individual;  
  
public class Commercial extends RealEstate  
{  
   private String zoning;  
   private int sqft;  
   private int parkingSpaces;  
  
   public String getZoning()  
   {  
       return zoning;  
   }  
   public void setZoning(String zoning)  
   {  
       this.zoning = zoning;  
   }  
   public int getSqft()  
   {  
       return sqft;  
   }  
   public void setSqft(int sqft)  
   {  
       this.sqft = sqft;  
  
He then selects the Residential.java file and the document contains the code:  
  
package hibernate.domain.individual;  
  
public class Residential extends RealEstate  
{  
   private int bedrooms;  
   private int bathrooms;  
   private int sqft;  
     
   public int getBedrooms()  
   {  
       return bedrooms;  
   }  
   public void setBedrooms(int bedrooms)  
   {  
       this.bedrooms = bedrooms;  
   }  
   public int getBathrooms()  
   {  
       return bathrooms;  
   }  
   public void setBathrooms(int bathrooms)  
   {  
       this.bathrooms = bathrooms;  
   }  
  
The presenter launches an instance of the Command Prompt - mysql -u root -p. There are three tables: realestate, commercial, and residential.  
  
Each table has six columns: Field, Type, Null, Key, Default, and Extra. The table represents the main class in the respective JavaScript documents. The fields match the attributes assigned to the class.  
  
The realestate table contains id, street, city, state, zip, and value fields. The commercial table contains zoning, sqft, and parkingSpaces fields, as well as the realEstateId field. The residential table contains bedrooms, bathrooms, and sqft fields, as well as a realEstateId field.*   
  
And also you can see in this database schema, all of the columns match all of the attributes in the database to the classes exactly, perfectly. Again just keeping it simple. Now the advantage of the strategy is common data for streets, and cities, and zip is always in the realestate class. If I want to show you everything is being sold, and I don't care about the details of if it's commercial or residential, I can just do searches and queries into the realestate class and load that up appropriately. So it gives me a great advantage. Now the disadvantage is I have to do joins every single time I want to go off and get that information. So for this example, we're going to look at a XML mapping for how do we get this done. It can be done through annotations or XML, it doesn't really matter, but we're going to look at the XML sample. So in the XML, I map the super class as being the main class of the XML mapping. All the subclasses, whether there is one or two or a hundred, get mapped within this class as being joined–subclasses. So the base class gets mapped with an ID, it gets mapped with the attributes it would have in the realestate class. You know, everything we see here in the realestate class is being mapped over inside of my XML mapping.   
*The presenter switches back to the Commercial.java file, which is open in Hibernate.  
  
He selects the RealEstate hbm.xml file. The document contains the code:  
  
<hibernate-mapping package="hibernate.domain.individual">  
           <class name="RealEstate" table="RealEstate">  
           <id name="id" column="id" type="long">  
                  <generator class="native/>  
           </id>  
           <property name="street"/>          
           <property name="city"/>          
           <property name="state"/>          
           <property name="zip"/>          
           <property name="value"/>  
  
           <joined-subclass name="Residential" table="Residential">  
                  <key column="realEstateId"/>  
                  <property name="bedrooms"/>  
                  <property name="bathrooms"/>  
                  <property name="sqft"/>      
           </joined-subclass>  
  
            <joined-subclass name="Commercial" table="Commercial">  
                  <key column="realEstateId"/>  
                  <property name="zoning"/>  
                  <property name="sqft"/>  
                  <property name="parkingSpaces"/>      
           </joined-subclass>      
  
The presenter points out the super class, which is RealEstate. He highlights the various attributes assigned to the RealEstate class and switches to the Command Prompt to show the correlating fields in the realestate table.*   
  
So past that I get to each one of my base classes, so the joined–subclass, Residential and I am pointing to the table name here. Again only because I'm showing you the option and since it's the same, I don't have to include it here. The first thing is I have to have that key column, I can't get away from the ID. And that key column points to the foreign key relationship of each of these guys to that main table. These are inexplicitly tied together. We can't even have the subclasses without the base class. Now from there I map each one of the attributes of the subclasses as I normally would. But you can see I'm not requiring one mapping file per each class at this point. I can put the whole hierarchy in one XML mapping file. So as I go through and look at my example here, I have a little test, where I go through and I can build some Residential property. I can build some Commercial property, and I do that just treating as if they were plain old Java objects. I don't have to care about the super class. It doesn't matter if the super class is abstract or not in this instance. It's all going to get loaded the same way; that happens on the database side. But I create two classes, I persist each of them and go from there. So let me go and **Run** this sample.   
*The Realestate hbm.xml document is open in Hibernate and it displays the code:  
  
<hibernate-mapping package="hibernate.domain.individual">  
           <class name="RealEstate" table="RealEstate">  
           <id name="id" column="id" type="long">  
                  <generator class="native/>  
           </id>  
           <property name="street"/>          
           <property name="city"/>          
           <property name="state"/>          
           <property name="zip"/>          
           <property name="value"/>  
  
           <joined-subclass name="Residential" table="Residential">  
                  <key column="realEstateId"/>  
                  <property name="bedrooms"/>  
                  <property name="bathrooms"/>  
                  <property name="sqft"/>      
           </joined-subclass>  
  
            <joined-subclass name="Commercial" table="Commercial">  
                  <key column="realEstateId"/>  
                  <property name="zoning"/>  
                  <property name="sqft"/>  
                  <property name="parkingSpaces"/>      
           </joined-subclass>    
  
The presenter points out the <joined sub-class> tag assigned to the Residential and Commercial tables. He then highlights the <key column="realEstateId> tag, which links the sub-class tables to the properties in the RealEstate table.  
  
He switches to the view of the tables in the Command Prompt and highlights the realEstateId field in the commercial and residential tables.  
  
The presenter switches back to the hbm.xml file in Hibernate and points out the attributes for each of the sub-classes. He then selects the Main.java file.  
  
The document contains the code:  
  
package hibernate.inheritance;  
  
import .hibernate.domain.individual.Commercial;[]  
  
public class Main  
{  
   public static void main(String[] args)  
   {  
//        new Main().runSingleTable();  
       new Main().runJoined();  
//        new Main().runIntegrated();  
   }  
  
   public void runSingleTable()  
   {  
       SessionFactory factory = null;  
       try  
       {  
           factory = ConfigHelper.getSessionFactory();  
           Session s = factory.openSession();  
           s.beginTransaction();  
  
  
He highlights the joined() function in the line of code:  
  
        new Main().runJoined();  
  
Then he right-clicks and a shortcut menu is displayed with the options: Open Declaration and Open Implementation. He selects Open Declaration and the code is implemented.  
  
The result is displayed as:  
  
   public void runJoined()  
   {  
       SessionFactory factory = null;  
       try  
       {  
           factory = ConfigHelper.getSessionFactory();  
           Session s = factory.openSession();  
           s.beginTransaction();  
  
           Residential r = new Residential();  
           r.setStreet("1600 Pennsylvania Ave.");  
           r.setCity("Washington");  
           r.setState("DC");  
           r.setZip("20500");  
           r.setBathrooms(35);  
           r.setBedrooms(16);  
           r.setSqft(55000);  
           r.setValue(323923785);  
             
           s.persist(r);  
  
           Commercial c = new Commercial();  
           c.setStreet("1400 Defense Pentagon");  
           c.setCity("Washington");  
           c.setState("DC");  
           c.setZip("20301");  
           c.setValue(6636360000.00);  
           c.setZoning("Governmental");  
           c.setSqft(6636360);  
           c.setParkingSpaces(8770);  
             
           s.persist(c);  
             
           s.getTransaction().commit();  
       } catch (Throwable t)  
       {  
           t.printStackTrace();  
           factory.getCurrentSession().getTransaction().rollback();  
       } finally  
       {  
  
The presenter navigates to the menu options and clicks the Run Main button.*   
  
...and it's going to go off and it's going to do my inserts. And notice it's going to do two inserts each time, one to the realestate class, one to the residential, one to the realestate class, one to the commercial, it's filling in all those values. So I can look at the values inside of the database now and you can see that we have a couple of items that got put in there. And so the White House and it actually...that's an actual real estimate of the value of the White House and the Pentagon, that's a made up estimate of the Pentagon. I don't know about that one. But it's actual, put that data inside of there for the realestate table, then for the residential, you can see the White House has 16 bedrooms and 35 bathrooms, 55 000 square feet. That's the data that got put in for that object and then the commercial for the Pentagon got put in for, you know, and that's actual a good estimate of the number of parking spots at the Pentagon. So you can see our JOINED subclass solution has one table per class and it joins them together every time I want to pull data out or I'm inserting it in multiple place when I data in. It's one of the three strategies out there, but it is a very powerful strategy if that's the one you choose.   
*The presenter has run the main().joined() section of the code in the Main.java document and the results have been returned as a java object.  
  
As the code is deployed, the progress of the build and the results are displayed in the Console view. The following results display:  
  
Hibernate: insert into RealEsate (street, city, state, zip, value) values (?, ?, ?, ?, ?)  
Hibernate: insert into Residential (bedrooms, bathrooms, sqft, realEstateId) values (?, ?, ?, ?)  
Hibernate: insert into RealEsate (street, city, state, zip, value) values (?, ?, ?, ?, ?)  
Hibernate: insert into Commercial (zoning, sqft, parkingSpaces, realEstateId) values (?, ?, ?, ?)  
  
The presenter points out that each insert is done twice, once to the RealEstate class and once to each of the sub-classes.  
  
He switches to the Command Prompt to view the resulting tables in the SQL server. There are three tables. The first table has five columns that link to the RealEstate attributes, id, street name, city, state, zip, and value. The two entries in the table are 1600 Pennsylvania Ave, which has an id of 1, and 1400 Defense Pentagon, which has an id of 2.  
  
The second table contains four columns, they are realEstateid, bedrooms, bathrooms, and sqft. The realEstate Id is 1, so the table displays the bedrooms, bathrooms, and sqft  values for the matching id in the realestate table.  
  
The second table has four columns, they are realEstateId, zoning, sqft, and parkingSpaces. The realEstateId for this table is 2, so the table displays the zoning, sqft, and parkingSpaces values for the matching id in the realestate table.*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a02_it_enus/output/html/course_transcript.html#top)

Inheritance – TABLE\_PER\_CLASS

Learning Objective

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

* *configure Java Hibernate to work with your applications*

**1. Using TABLE-PER-CLASS**

Inheritance is such a key feature inside of Java, but no clear corollary exists inside of your database. And so there's three strategies by which we can map in the database, and this one we're going to look at, is having a single database table per subclass that's out there. And so in this example, I have a superclass that basically has no particular business meaning. It has some shared features, so in this case, I'm calling it my HistoryBase. So for my objects, I want to be able to keep track of the date they were created, and the date they were modified. And that's a key business feature, and I want to use inheritance as part of that. Now traditionally in mapping, especially within annotations, I have to have all that information straight in front of me. In XML, I can map this the same way, but I'm not quite as worried about the class structure, because I'm only looking at getters, and setters at that point. I'm not worried about actually the attributes, and mapping them directly in the Java class. So in this case, I don't have any entity, I don't have any Java mapping of annotations into Hibernate, inside of my base class. I'm reserving all of that for my subclasses, which then need to tie back to those up above. So here, you can see I'm creating these things as being temporal, I'm creating the attributes as being set up like there. But I don't actually tie those to a class, until I get to the subclass. And in the subclass, I'm adding the @Inheritance attribute, with the strategy of having one *TABLE\_PER\_CLASS*.   
*Hibernate is open four files are available in the document window: Main.java, HistoryBase.java, Story.java, and Page.java.  
  
HistoryBase.java is selected and displays the code:  
  
package hibernate.domain.inheritance;  
  
import java.util.Date;  
  
public class HistoryBase  
{  
   @Temporal(TemporalType.DATE)  
   private Date   creationDate;  
  
   @Temporal(TemporalType.TIMESTAMP)  
   private Date   lastModifiedDate;  
     
   public Date getCreationDate()  
   {  
       return creationDate;  
   }  
   public void setCreationDate(Date creationDate)  
   {  
       this.creationDate = creationDate;  
   }  
   public Date getLastModifiedDate()  
   {  
       return lastModifiedDate;  
   }  
   public void setLastModifiedDate(Date lastModifiedDate)  
   {  
       this.lastModifiedDate = lastModifiedDate;  
   }  
}  
  
The presenter selects the Story.java file. The document contains the code:  
  
package hibernate.domain.inheritance;  
  
import java.util.ArrayList;  
  
@Entity  
@Inheritance(strategy=InheritanceType.TABLE\_PER\_CLASS)  
@Table(name="Story")  
public class Story extends HistoryBase   
{  
   @Id  
   @GeneratedValue   
   private long   id;  
  
   @Basic  
   private String title;  
  
   @Basic  
   private Date   storyDate;  
  
   @OneToMany(cascade=CascadeType.ALL)  
   @JoinColumn(name = "storyId", nullable = false, updatable = false, insertable = true)  
   private List<Page> pages = new ArrayList<Page>();  
  
   public void addPage(Page page)  
   {  
       page.setStory(this);  
       page.setPageNumber(pages.size() + 1);  
       pages.add(page);  
   }  
  
In this section of code, he highlights the TABLE-PER-CLASS attribute that has been added to the Inheritance strategy in the following line:  
  
@Inheritance(strategy=InheritanceType.TABLE\_PER\_CLASS)  
  
He switches back to the HistoryBase.java document and points out that the creationdate attribute and lastModifiedDate attribute are temporal type attributes.*   
  
So I have a Story class, a Page class, however many classes I want to create, it doesn't matter because each class has its own database table associated with it. And so if I look at the database description you can see, story is here, it has id, title, storyDate, and then it has the common attributes: creationDate and lastModifiedDate. Similarly, page has an id, StoryId, pageNumber, words, whether it has htmlIncluded, and then it has the common elements of creationDate, lastModifiedDate. And those are a key business feature, they want to be stored in a common way, but they're going to be across many, many, many tables out there. The HistoryBase is a functional class, much more than a data storage class. It does have some data associated with it, but it wants to be sure that feature is available across all the objects in this hierarchy. So once I've set the *TABLE\_PER\_CLASS*, everything else is mapped the same way. I map it to the Table as the table name, and in this case it matches, but I can also explicitly put it out there. I have ids, I have attributes, I have associations between classes, I don't change any of that behavior. But I do not have to map either creationDate, or lastModifiedDate because they're automatically added through inheritance. The same thing here with Page, I do not have to map creationDate or lastModifiedDate. I just do the rest of the ones I care about; the bidirectional association back to story, and all the other separate fields that I need to worry about.   
*The presenter opens the Command Prompt. Two tables have been created from the Story and Page sub-classes.  
  
The command describe story outputs a table with six columns and five rows. The Field column lists the attributes: id, title, storyDate, creationDate, and LastModifiedDate.  
  
The describe page command outputs a table with six columns and seven rows. The Field column lists the attributes: id, storyId, pageNumber, words, htmlIncluded, creationDate, and LastModifiedDate.  
  
The presenter switches back to Hibernate and the Story.java document. He points out that he has mapped the sub-class to the table and named the table using the line of code:  
  
@Table(name="Story")  
  
He scrolls through the code pointing out the attributes such as id, and title. He also highlights where he has created associations between the classes in the section of the code:  
  
   @OneToMany(cascade=CascadeType.ALL)  
   @JoinColumn(name = "storyId", nullable = false, updatable = false, insertable = true)  
   private List<Page> pages = new ArrayList<Page>();  
  
The presenter selects the Page.java file and the document contains the code:  
  
package hibernate.domain.inheritance;  
  
import javax.persistence.Basic;  
  
@Entity  
@Inheritance(strategy=InheritanceType.TABLE\_PER\_CLASS)  
@Table(name="Page")  
public class Page extends HistoryBase  
{  
   @Id  
   @GeneratedValue   
   private long   id;  
  
   @OrderColumn(name="pageNumber", insertable=true, updatable=true, nullable=false)  
   private int pageNumber;  
  
   @Basic  
   private String words;  
  
   @Basic  
   private boolean htmlIncluded;  
  
   @ ManyToOne  
   private Story story;  
   @ JoinColumn(name= "storyId")  
   private Story story;  
  
He points out the attributes, as well as the many-to-oOne relationship that he is setting up with the Story table.*   
  
So when I run my sample here, I'm building a story, I'm building some pages. These guys are actually tied together, they all get saved together as one. But I'm, you see I'm saving it, I'm loading it, I'm pulling it back up, and it's all happening somewhat seamlessly, where it automatically gets those extra fields, it automatically is going to get those extra items from the other tables. So if I go and look at this here, I go to my fields and then, there you go. I have my data is out there, my creationDate and everything else is added in within that. And so this strategy allows us to consolidate things up top, but not be tied to how the solution is working down below. The Story, the Page… and it doesn't necessarily just have to be functional, but they're all storing all the data within each one of the tables, that's dedicated towards that class. So the advantage here is that I don't have joins involved, I don't have to worry about complicated selects when I'm going to go load data. The disadvantage is there's no commonality between these guys. If I'm searching for common features across classes, it doesn't exist. So you want to use this strategy where I don't have to search across classes. I don't care if these things happen to be in common, what I'm really trying to show is the representation of the values in the story. So that's this strategy, so you need to choose which one's appropriate for you, but this one is great when there is very little in common across the classes, that's shared beyond what's in each table.   
*The presenter selects the Main.java file. The document contains the code:  
  
public class Main  
{  
   public static void main(String[] args)  
   {  
//        new Main().runSingleTable();  
//        new Main().runJoined();  
       new Main().runIntegrated();  
   }  
  
   public void runSingleTable()  
   {  
       SessionFactory factory = null;  
       try  
       {  
           factory = ConfigHelper.getSessionFactory();  
           Session s = factory.openSession();  
           s.beginTransaction();  
  
The presenter navigates to the menu options and clicks Run Main.  
  
The following results are displayed in the results pane Console tab:  
  
Page 1  
  
I am the first page from inheritance  
  
Page 2  
  
I am the second page from inheritance  
  
(creation: null modified: null)  
  
The presenter switches to the Command Prompt where two new tables have been added based on the command:  
  
source showStory.txt  
  
The first table has five columns and one row:  
  
id: 1  
title: Welcome to Hibernate Inheritance  
storyDate: 2014-02-25  
creationDate: 2014-02-25 00:56:58  
LastModifiedDate: 0000-00-00 00:00:00  
  
The second table has seven columns and two rows. The first row can be read as:  
  
id: 1  
storyId: 1  
pageNumber: 1  
words: I am the first page from inheritance  
htmlIncluded: blank  
creationDate: 2014-02=25 00:56:58  
LastModifedDate: 0000-00-00 00:00:00  
  
The second row values are:  
  
id: 2  
storyId: 1  
pageNumber: 2  
words: I am the second page from inheritance  
htmlIncluded: blank  
creationDate: 2014-02=25 00:56:58  
LastModifedDate: 0000-00-00 00:00:00*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a02_it_enus/output/html/course_transcript.html#top)

Inheritance – Single Table

Learning Objective

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

* *apply the Single Table strategy as a solution for mapping into a database in Java Hibernate*

**1. Using SINGLE-TABLE**

Mapping Inheritance in Java has three different strategies and this time we're going to talk about the single table strategy. So in Java we're always going to have inheritance hierarchy. In this case, we have an abstract base class called PaymentCard. And it represents any sort of card I would use to make a payment and two different types we're representing here is a prepaid card and a credit card. And so again there's some common things that all payment cards have, they a card number, they have a pin, expiration month, year, name, stuff like that. Where credit cards have a limit and a balance that's associated with that. And a prepaid card has an available balance that you can use and pay off of that. So again the model is not terribly important, but the concept is. I have this inheritance hierarchy and I need to map that to tables. Now there's no immediate clear strategy to do that. Databases don't have the notion of inheritance the same way Java does. And so in this case we're choosing a model by which we're creating a table that has all of the columns for all of the subclasses in one table.   
*Hibernate is open and four files are available is the document view: PaymentCard.java, Prepaid.java, Credit.java, and Main.java.  
  
PaymentCard.java is selected and displays the code:  
  
package hibernate.domain.single;  
  
import javax.persistence.Basic;  
  
@Entity  
@Inheritance(strategy=InheritanceType.SINGLE\_TABLE)  
@Table(name="PaymentCard")  
public abstract class PaymentCard  
{  
   @Id  
   private int cardNumber;  
     
   @Basic  
   private int pin;  
  
   @Basic  
   private int expMonth;  
  
   @Basic  
   private int expYear;  
  
   @Basic  
   private String name;  
  
   public int getCardNumber()  
   {  
       return cardNumber;  
   }  
   public void setCardNumber(int cardNumber)  
   {  
       this.cardNumber = cardNumber;  
   }  
   public int getPin()  
   {  
       return pin;  
   }  
   public void setPin(int pin)  
   {  
       this.pin = pin;  
   }  
   public int getExpMonth()  
   {  
       return expMonth;  
   }  
   public void setExpMonth(int expMonth)  
   {  
       this.expMonth = expMonth;  
   }  
   public int getExpYear()  
   {  
       return expYear;  
   }  
   public void setExpYear(int expDate)  
   {  
       this.expYear = expDate;  
   }  
   public String getName()  
   {  
       return name;  
   }  
   public void setName(String name)  
   {  
       this.name = name;  
   }  
}  
  
The presenter selects the Credit.java file. The document displays the code:  
  
package hibernate.domain.single;  
  
import javax.persistence.Basic;  
  
@Entity  
@Inheritance(strategy=InheritanceType.SINGLE\_TABLE)  
@Table(name="PaymentCard")  
public class Prepaid extends PaymentCard  
{  
   @Basic  
   @Column(name="balance")  
   private double availableBalance;  
  
   public double getAvailableBalance()  
   {  
       return availableBalance;  
   }  
  
   public void setAvailableBalance(double availableBalance)  
   {  
       this.availableBalance = availableBalance;  
   }  
}  
  
The presenter selects the Prepaid.java file and points out the prepaid sub-class specific attributes such as available balance.   
  
The document contains the code:  
  
package hibernate.domain.single;  
  
import javax.persistence.Basic;  
  
@Entity  
@Inheritance(strategy=InheritanceType.SINGLE\_TABLE)  
@Table(name="PaymentCard")  
public class Credit extends PaymentCard  
{  
   @Basic  
   @Column(name="accountLimit")  
   private double limit;  
  
   @Basic  
   private double balance;  
  
   public double getLimit()  
   {  
       return limit;  
   }  
   public void setLimit(double limit)  
   {  
       this.limit = limit;  
   }  
   public double getBalance()  
   {  
       return balance;  
   }  
   public void setBalance(double balance)  
   {  
       this.balance = balance;  
   }  
}  
  
The presenter selects the PaymentCard.java file.*   
  
We're going to put all the data inside of there. So you can see, I have a cardNumber, I have a name, I have a pin, I have expiration month, I have an expiration year. And then I have the limit and the balance, that's inside of there. And I'm choosing this because the vast majority of the columns are in common. And really most of the extra columns aren't that offensive. I actually only have one extra column here in this design because I'm choosing to use a column for two different definitions. Balance and available balance in this case is being shared between the two classes, between Prepaid and Creditcard. And so the mapping happens at...in this case we're going to use annotations. And in annotations, again, we could use XML as well, but in annotations we're using the @Inheritance and we're setting the strategy and type inside of here. Again there's three strategies that we can use, JOINED, SINGLE\_TABLE, or TABLE\_PER\_CLASS, but we're choosing the SINGLE\_TABLE in this instance. And so we map to the table, we're going to point to PaymentCard, it just happens to be the same name in this case. And then we have the Id and all the other attributes defined as they normally would within properties. Now the two subclasses also are using inheritance with the same strategy and I need to point these guys to the same common table that's involved. It might not match any of the class names, in this case it happens to match the parent class name. But then past that, I'm mapping all the columns the same way.   
*The presenter opens the Command Prompt, which displays a table with six columns: Field, Type, Null, Key, Default, and Extra.  
  
The table has seven rows that reflect the attributes listed in the base class as well as the sub-class: cardNumber, name, pin, expMonth, expYear, accountLimit, and balance.  
  
He switches back to the PaymentCard.java file in Hibernate. He focuses on the section of code:  
  
package hibernate.domain.single;  
  
import javax.persistence.Basic;  
  
@Entity  
@Inheritance(strategy=InheritanceType.SINGLE\_TABLE)  
@Table(name="PaymentCard")  
public abstract class PaymentCard  
{  
   @Id  
   private int cardNumber;  
     
   @Basic  
   private int pin;  
  
   @Basic  
   private int expMonth;  
  
He points out the use of the @Inheritance annotation. He highlights that the strategy is being set as InheritanceType using SINGLE\_TABLE.  
  
He opens the definition panel for the InheritanceType to show that the options that include: JOINED, SINGLE\_TABLE, and TABLE\_PER\_CLASS. The presenter also highlights that it is being mapped to the PaymentCard table using the @Table annotation.  
  
He selects the Prepaid.java file and points out that the sub-classes are also making use of the SINGLE\_TABLE inheritance type but they are being mapped to the PaymentCard table, as shown in the section of code from the document:  
  
package hibernate.domain.single;  
  
import javax.persistence.Basic;  
  
@Entity  
@Inheritance(strategy=InheritanceType.SINGLE\_TABLE)  
@Table(name="PaymentCard")  
public class Prepaid extends PaymentCard  
{*   
  
Notice I don't have to map an Id in this case. Because of the inheritance, because of the strategy I'm choosing, I know the Id is from my base class. It's defined elsewhere in my strategy, I don't have to do it inside of my object here in either the Prepaid card or the Credit card. I can just tie these to what I want to, but also notice here, the balance here ties straight to the balance field and here the balance ties to the availableBalance field. I'm reusing that column across both of these. So when I go through and execute my solution, I have my SingleTable here and in my code sample here, I'm going through and I'm creating a credit card. I'm filling it with all the values I put inside of there and then I'm saving it. I'm creating a prepaid card, I am filling it with all the values I need there and saving it. I can't build a straight Credit card, it's abstract, but I can go through and run this solution and it's going to insert each of those guys to the database. Now each of them gets one insert statement, because it's all going into one table. I want you to notice something here though, if I look at the list of attributes, I have the month, the year, the name, the pin, the balance, the account type, and this thing called DTYPE that gets associated inside of there.   
*The Prepaid.java file is selected and the document displays the code:  
  
   @Basic  
   @Column(name="balance")  
   private double availableBalance;  
  
   public double getAvailableBalance()  
   {  
       return availableBalance;  
   }  
  
   public void setAvailableBalance(double availableBalance)  
   {  
       this.availableBalance = availableBalance;  
   }  
}  
  
The presenter points out that there is no id attribute in the sub-class.  
  
He clicks the Credit.java file. The document contains the code:  
  
package hibernate.domain.single;  
  
import javax.persistence.Basic;  
  
@Entity  
@Inheritance(strategy=InheritanceType.SINGLE\_TABLE)  
@Table(name="PaymentCard")  
public class Credit extends PaymentCard  
{  
   @Basic  
   @Column(name="accountLimit")  
   private double limit;  
  
   @Basic  
   private double balance;  
  
   public double getLimit()  
   {  
       return limit;  
   }  
   public void setLimit(double limit)  
   {  
       this.limit = limit;  
   }  
   public double getBalance()  
   {  
       return balance;  
   }  
   public void setBalance(double balance)  
   {  
       this.balance = balance;  
   }  
}  
  
He points out the following line of code:  
  
   @Basic  
   private double balance;  
  
He opens the Prepaid.java document and points out the line of code:  
  
   @Basic  
   @Column(name="balance")  
   private double availableBalance;  
  
He clicks the Main.java file. The document contains the code:  
  
public class Main  
{  
   public static void main(String[] args)  
   {  
       new Main().runSingleTable();  
//       new Main().runJoined();  
//        new Main().runIntegrated();  
   }  
  
   public void runSingleTable()  
   {  
       SessionFactory factory = null;  
       try  
       {  
           factory = ConfigHelper.getSessionFactory();  
           Session s = factory.openSession();  
           s.beginTransaction();  
  
The runSingleTable() function is highlighted, he right-clicks and selects Open Declaration. The page displays the code:  
  
public class Main  
{  
   public static void main(String[] args)  
   {  
       new Main().runSingleTable();  
//        new Main().runJoined();  
//        new Main().runIntegrated();  
   }  
  
   public void runSingleTable()  
   {  
       SessionFactory factory = null;  
       try  
       {  
           factory = ConfigHelper.getSessionFactory();  
           Session s = factory.openSession();  
           s.beginTransaction();  
             
           Credit cc = new Credit();  
           cc.setName("Mr. Me");  
           cc.setCardNumber(123456789);  
           cc.setPin(1234);  
           cc.setExpYear(16);  
           cc.setExpMonth(11);  
           cc.setLimit(1000);  
           cc.setBalance(236);  
             
           s.persist(cc);  
  
           Prepaid pp = new Prepaid();  
           pp.setName("Ms. Me");  
           pp.setCardNumber(987654321);  
           pp.setPin(4321);  
           pp.setExpYear(15);  
           pp.setExpMonth(4);  
           pp.setAvailableBalance(549);  
  
           s.persist(pp);  
             
           s.getTransaction().commit();  
       } catch (Throwable t)  
       {  
           t.printStackTrace();  
           factory.getCurrentSession().getTransaction().rollback();  
       } finally  
       {  
           // To kill the run else Hibernate leaves it active.  
           factory.close();  
           System.exit(0);  
       }  
  
The presenter navigates to the menu options and clicks the Run Main button. In the Console view, the output is displayed:  
  
Hibernate: insert into PaymentCard (expMonth, expYear, name, pin, balance, accountLimit, DTYPE, cardNumber) values (?, ?, ?. ?, ?...  
Hibernate: insert into PaymentCard (expMonth, expYear, name, pin, balance, , DTYPE, cardNumber) values (?, ?, ?, ?, ?, 'prepaid',...*   
  
Now in my database design, I didn't specify anything called DTYPE. I want to show you the little trick I have going on inside of here because now if I go and get all the values from the card, you can see DTYPE has been added to the description of the database field. If I look at the description again, let me go down here, if I look at the description again, it's added in DTYPE, it's a var chart inside of there. And the DTYPE represents the name of the class for the type of this. When I go through as Hibernate and read these objects out later on, I need to know what type of object I'm creating. Am I creating a credit card, am I creating a prepaid card, or whatever other fields that are out there for that. So the single class model requires you to have that extra column that tells you what type of class I'm loading. And the key to that is...the reason why that all works is because in my **hibernate.cfg**, I have my solution set up to allow it to update the table. I could choose to add a column called DType on my own. I could do that through my DDL, my schema.   
*The presenter opens the Command Prompt, the paymentcard table that is displayed does not include the DTYPE field. He presses Enter to build the new table.   
  
This table has eight columns: cardNumber, name, pin, expMonth, expYear, accountLimit, balance, and DTYPE. There are two rows representing the credit card and the prepaid card and the associated values. The card type is listed under the DTYPE column.  
  
The presenter runs the mysql> describe paymentcard command. The output displays the payment card table and this time DTYPE has been added to the Field column and is listed as a varchart in the Type column.  
  
He switches back to Hibernate and selects the hibernate.cfg.xml file in the Package Explorer. He points out a section of the code:  
  
<property name="hbm2ddl.auto">update</property>  
  
He is using this to automatically add the DTYPE column and update the value in the table to show the type of payment card.*   
  
In this case, I've let Hibernate add the column as it's desired. Either way, it works just fine but I wanted you to see that extra column magically appear, if you will, to show you that behavior inside of this strategy. So the single in table strategy works really well because it avoids all joins. It doesn't require me to join, to be able to load a single object and it allows me to search the single table to get all the information about any class in the hierarchy. It seems like a great strategy, the only disadvantage is I can end up with a lot of columns that are null. If there's a lot of columns that do not overlap between the subclasses, then they'll be empty and they won't have any definition across the different classes, and that means your table is not very normalized, and that might be undesirable. So this is one of the strategies of the three, but you need to choose what's the best for you and so consider this one as a good strategy when mapping from inheritance structures into your database.

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a02_it_enus/output/html/course_transcript.html#top)

Identity Generation in Annotations

Learning Objective

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

* *use Annotations to assign IDs to rows in Java Hibernate*

**1. Generating identities**

As every row requires unique Id, Hibernate annotations allow you to manage the way the Ids are generated. The main decision falls into which database you're using, but custom options also exist to meet nearly every need. As Hibernate has the requirement for an Id field on every class, databases also tend to have a requirement that that Id is unique. It's a primary key, it's special out there. So the basic Id field inside of Hibernate has the issue that it only uses whatever value you set as the user. That's not necessarily a problem if you want to take responsibility for that, but in my little sample program here, I can run it one time and it creates an object with Id of zero. If I try and run it a second time though, it tries to put another object of Id zero out there and I end up with the duplicate entry. Because you know, in my database, I already have an object out there of Id zero, I can't add another one. There's a quick and easy fix for that, where in my database, I can instead choose to use a database with the auto increment field. In MySQL, DB2, many databases, this is a field that will automatically insert a new Id for your table, for your row entry, based off of the previous Ids used. So 1, 2, 3, 4, 5 numbers sequentially done.   
*Hibernate is open and there are two document views open. The first one contains the IdentityMain.java document. There are a number of documents open in the second document view, including: NoGenerator.java, No GeneratorAuto.java, Identity.java, Sequence.java, TableGenerator.java, and CustomIDGenerator.java.  
  
Below this, the Console is available.  
  
The NoGenerator.java file is selected and the document displays the code:  
  
@Entity  
@Table(name="Generic")  
public class NoGenerator  
{  
   private long id;  
  
   @Basic  
   private String text;  
  
   public long getId()  
   {  
       return id;  
   }  
  
   public void setId(long id)  
   {  
       this.id = id;  
   }  
  
The presenter opens the IdentityMain.java document. It displays the code:  
  
public class IdentityMain  
{  
   private SessionFactory factory;  
  
   public static void main(String[] args)  
   {  
       new IdentityMain().run();  
   }  
     
   private void run()  
   {  
       try  
       {  
           factory = ConfigHelper.getSessionFactory();  
           Session session = factory.getCurrentSession();  
           session.beginTransaction();  
  
          NoGenerator id = new NoGenerator();  
//            NoGeneratorAuto id = new NoGeneratorAuto();  
//            Identity id = new Identity();  
//            TableStrategy id = new TableStrategy();//           CustomIDGenerator id = new CustomIDGenerator();  
  
The presenter runs the NoGenerator id = new NoGenerator(); code and the output displays in the Console:  
  
Created id is 0.  
  
He runs the code a second time and output is:  
  
Duplicate entry "0" for key "PRIMARY".  
  
The presenter switches to the Command Prompt to view the results table, which shows that there is already a row in the table that has an id of zero.  
  
He switches back to Hibernate and clicks the NoGeneratorAuto file. It displays the code:  
  
@Entity  
@Table(name="Generic\_Auto")  
public class NoGeneratorAuto  
{  
   private long id;  
   private String text;  
  
   public long getId()  
   {  
       return id;  
   }  
  
   public void setId(long id)  
   {  
       this.id = id;  
   }  
  
   public String getText()  
   {  
       return text;  
   }  
  
   public void setText(String text)  
   {  
       this.text = text;  
   }  
}  
  
He points out the @Table annotation and the  (name="Auto\_Generator") description.*   
  
So I can do that. Here's...in my second example I will go off and run here. I'm now pointing to a different table, it's a table with an automatic Id outside of there. So it says that Id created is zero. If I run it a second time, it will actually tell me again the Id is zero, but it didn't fail. So if I go look at the database, I have a different table now, but the first time it said Id 1, the second time it said Id 2 and it will keep going 3, 4, 5 as I go through in there. The issue is Hibernate is unaware of what's going on inside the table. So it never is aware of what the actual identifier is in that database. So in order to fix that we can actually attribute to Hibernate a second tag inside of here, an annotation that tells us what that strategy is for generating the value. The first strategy we'll talk about is identity. Identity is just what we talked about, the database is going to add the Id on its own, it's going to increment and associate it with the table, and what Hibernate does is after the insert query, it's going to do a check for what Id was just generated. And depending on the call, it will actually update the object with that Id, persist doesn't, but save does.   
*The presenter switches back to the IdentityMain.java document and runs the code:  
  
//           NoGenerator id = new NoGenerator();  
           NoGeneratorAuto id = new NoGeneratorAuto();  
//            Identity id = new Identity();  
//            TableStrategy id = new TableStrategy();  
//            CustomIDGenerator id = new CustomIDGenerator();  
  
The output is displayed in the Console:  
  
Created id is 0.  
  
He runs the code again and the feedback shows that the created id is zero but this time there is no error message.  
  
He switches to the Command Prompt and a new table has been created in the database. The table has two tows: the first row has been assigned an id of 1 and the second row has been assigned an id of 2.  
  
The presenter switches back to Hibernate and clicks the Identity.java file. The document contains the code:  
  
@Entity  
@Table(name="Generic\_Auto")  
public class Identity  
{  
   @Id  
   @GeneratedValue(strategy=GenerationType.IDENTITY)  
   private long id;  
  
   @Basic  
   private String text;  
  
   public long getId()  
   {  
       return id;  
   }  
  
   public void setId(long id)  
   {  
       this.id = id;  
   }  
  
   public String getText()  
   {  
       return text;  
   }  
  
   public void setText(String text)  
   {  
       this.text = text;  
  
He points out that the @Generator annotation has been added and that is describes the strategy for the GenerationType as IDENTITY.*   
  
So if I go and run this example here, I'm going to be generating that same table again. But now you can see the Id created was 3, I run it one more time, the Id created was 4. And if I look into that table, you see I've added those two additional rows to that new table. So that works really really well for most databases, not all databases though. Oracle being one known exception, there might be others as well, uses instead what's called a sequence for that. The sequence changes the order in which this happens. So in Oracle, you first go to the database and ask it for the Id and then you insert with that Id. That's the strategy that's used there and that's the sequence strategy that's chosen. Again all you have to do is point to that strategy and then you can configure your queries to go from there. It'll automatically go off and do that. I'm not using that as an example in my database, so I'm just going to show the strategy to you, I'm not going to actually run it.   
*The presenter switches back to the IdentityMain.java document and runs the code: Identity id = new Identity(); code  
  
This is from the section of the code:  
  
//           NoGenerator id = new NoGenerator();  
//            NoGeneratorAuto id = new NoGeneratorAuto();  
           Identity id = new Identity();  
//            TableStrategy id = new TableStrategy();  
//            CustomIDGenerator id = new CustomIDGenerator();  
  
The results are displayed in the Console. It shows that the created id is 3. The presenter runs the code a second time and the result is the created id is 4.  
  
He switches to the Command Prompt. A new table has been generated, with four rows. The rows are assigned an Ids of 1, 2, 3, and 4 respectively.  
  
The presenter switches to Hibernate and clicks the Sequence.java file. It contains the code:  
  
@Entity  
@Table(name="Generic\_Auto")  
public class Sequence  
{  
   @Id  
   @GeneratedValue(strategy=GenerationType.SEQUENCE)  
   private long id;  
  
   @Basic  
   private String text;  
  
   public long getId()  
   {  
       return id;  
   }  
  
   public void setId(long id)  
   {  
       this.id = id;  
   }  
  
   public String getText()  
   {  
       return text;  
   }  
  
   public void setText(String text)  
   {  
       this.text = text;  
   }  
}  
  
He points out the @GenerateValue annotation description has been modified, the strategy for the generation type has changed to SEQUENCE. He closes the document.*   
  
Another Hibernate strategy available to you is the TableGenerator. The one problem with those strategies we just talked about, is it ends up having two queries to the database every time I want to insert. The table strategy allows us instead to use a sequence generator that gives me a block of numbers that I can insert. And so in this example, I'm adding an additional annotation, which tells you what the table name for my high-load generating algorithm is. It tells you what the primary column is for that, what the name of this guy is and it matches it up to the name up here. The initial value, the allocation size. Again it's just assigning 10 Ids at a time, it could be a 100, it could be a 1000, and then what the column for the next value is. So if I run this table strategy, it's going to go through and again it's not going to use the auto increment, it's going to use a generic one. It created an Id with id 1. Now it grabbed 1, 2, 3, 4, 5, 6, 7, 8, 9 as Ids it could've used. So the second time I run it, it starts with id 10, because the assumption is it already used up one through nine.   
*The presenter clicks the TableGenerator.java file. The code is displayed:  
  
@Entity  
@Table(name="Generic")  
public class Sequence  
{  
   @Id  
   @GeneratedValue(strategy=GenerationType.TABLE, generator="generic\_sequence")  
@TableGenerator(table="HiLo",  
                                 pkColumnName="tableName",  
                                 name="generic\_sequence",  
                                 initialValue=1,  
                                 allocationSize=10,  
                                 valueColumnName="nextHi",  
                                 )  
  
He points out the strategy for this example makes use of a sequence generator. He goes on to highlight the properties assigned to the table as well as the fact that 10 Ids will be assigned at a time  
  
He switches to the IdentityMain.java document and runs the code for the example. The results are displayed in the Console tab. The first result is that the created id is 1. He runs the code a second time and the result is that the created id is 10.*   
  
So in this strategy, there might be empty unused Ids, but I can at least not have to make as many queries. Again I can run it again and it will end up with number 20 inside of here. If I look at the tables of this you can first see that I have in my initial table, 1, 10 and 20 that were inserted into the table. And now I have this high-low table here that for table Generic, that's what we base the key off of what table am I pointing to, I've gotten data from this three times. It's updated that three times and that's where it gets the 10, 20, 30. Now all the algorithm for doing that is stored inside of Hibernate. The high-low strategy is stored inside of Hibernate. As far as table goes, we just need to create, or allow Hibernate to create a table to store the table name and the next high value it's going to grab out of there. It's a nice little design that keeps it simple, that adds a certain amount of efficiency. Now the last strategy we'll talk about here is custom identifiers. You might have a special rule you need to identify with. Looking up a social security number or generating some sort of unique key or something internally, who knows what this strategy is. My strategy I'm defining in this class, you define it in your own class.   
*The presenter examines the output of the TableStrategy id = new TableStrategy(); code, which is  displayed in the Console. The results show that by using this strategy the ids are assigned in batches of 10.  
  
He runs the code for a third time and the result is that the created id is 20. He switches to the Command Prompt to view the results in the database table. He points out that the table that is generated has four rows, the rows are assigned an id of 0, 1, 10, and 20 respectively. Another table has also been generated the table only has two columns, tableName and nextHi. There is a single entry in the table, the table name is Generic and the nextHi value is 3.*   
  
Inside of here, that generic generator outside of here, I define the generic generator if I want to use it more than once. And then I have a strategy which is based off of some class I defined and then I defined and generated value, a pointer to my generic generator. The custom generator then, must implement this identifier generator interface provided by Hibernate, which just simply is saying generate a new Id. It gives you a session in case you want to go through and talk to the database, it gives you an object. In my case, I'm just simply returning a random value between one and 100. Past a hundred I'm in trouble, but this should show us a nice little simple example. So if I go through this test and run this test, we'll see there's a chance I'm going to run into it, run into a collision. But the first time I run it here, oh yes, I ran into a collision, because it picked 10 as a random number and so it collided with my existing database. So we'll run it again and the odds should be lower, we should pick up something this time up, now it created an Id of 30. I can run it again and it's going to pick another random number, 34. Again it's totally random, it just, you know luck of the draw that I hit an existing one, 59. So if I look at the values I have now, you can see I've inserted 30, 34, 59 after running into 10 which already existed. So these are strategies by which you can not have to worry about Id generation in your source code but simply let the database or some other third-party tool you create handle the Id generation and keep your code simple, yet still have unique valid Ids out there.   
*The presenter switches back to Hibernate and selects the CustomIDGenerator.java file. The document contains the code:  
  
@Entity  
@Table(name="Generic")  
public class CustomIDGenerator  
{  
   @Id  
   @GenericGenerator(name="myCustom", strategy="hibernate.domain.idenitfier.generator.CustomGenerator")         
   @GeneratedValue(generator="myCustom")  
  
He points out that he is defining the generator as myCustom generator, he defines the strategy and the generated value.  
  
He selects the CustomGenerator.java file and points out the code:  
  
public class CustomGenerator implements identifierGenerator  
{  
       private Random r = new Random();  
       @Override  
       public Serializable generate(SessionImplementor si, Object o) throws Hibernate...  
       {  
               long id = r.nextInt(100);  
               return id;  
       }  
  
}  
  
He points out that the custom generator must implement the identify generator interface, as detailed in the line:  
  
public class CustomGenerator implements identifierGenerator  
  
The presenter runs the test from the IdentifyMain.java document and the following result is displayed in the Console:  
  
Caused by: java.sql.BatchUpdateException: Duplicate entry '10' for key 'PRIMARY'.  
  
He runs the test a second time and the result in the Console shows:  
  
Created id is 30  
Hibernate: insert into Generic (text, id) values (?, ?)  
  
He runs the code a number of times and each time a random identifier is created.  
  
He switches to the Command Prompt and the database table has been updated to include the rows containing the most recently generated Ids.*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a02_it_enus/output/html/course_transcript.html#top)

Identity Generation in XML

Learning Objective

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

* *use Identity Generation in XML using Java Hibernate*

**1. Generating Ids using XML**

Every Hibernate class must have an Id, and that Id normally is a primary key and thus must be unique. So, if I map, as we can see in this example, an Id, all by itself, I can run it, and the first time I run through my code, it's going to run just fine. It's going to go through, and it's going to insert to this database, and you can see, I've added in with a zero Id. Now the problem is, if I run this a second time, the code runs the exact same thing, it creates an object that still has a zero Id, and guess what, I could not enter that in the database, because it's a duplicate key, that one was already out there. Now I can update my database, and so I'm going to go to my second example here. And in my second example, I'm not using a generator, but I'm using a different table. You can see I'm using the Generic\_auto table, which has an auto incrementer on my Id. That's a feature in your database that every time I insert something to a table, the primary key will automatically pick the next item in line. So your Id will be 1, 2, 3, 4, 5, and so on. So in this case, if I run this code, it runs just fine and happy. And it will insert the value, but as far as Hibernate knows, the Id of this guy is zero. Again I can run this a second time, it'll execute, it will actually do successfully, but as far as it knows, the Id is zero. When I look in the database...   
*Hibernate is open various files are open in the document view.  
  
The NoGenerator.hbm.xml file is selected and the document displays the code:  
  
<?xml version="1.0" encoding="utf-8"?>  
<!DOCTYPE hibernate-mapping PUBLIC   
"-//Hibernate/Hibernate Mapping DTD//EN"  
"http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">   
  
<hibernate-mapping package="hibernate.domain.identifier.generator">  
   <class name="NoGenerator" table="Generic">  
<id name="id"/>  
<property name="text"/>  
   </class>  
</hibernate-mapping>  
  
The presenter switches to the IdentityMain.java tab. The document contains the code:  
  
public class IdentityMain  
{  
   private SessionFactory factory;  
  
   public static void main(String[] args)  
   {  
       new IdentityMain().run();  
   }  
     
   private void run()  
   {  
       try  
       {  
           factory = ConfigHelper.getSessionFactory();  
           Session session = factory.getCurrentSession();  
           session.beginTransaction();  
  
// Generated IDs in XML  
          NoGenerator id = new NoGenerator();  
//            NoGeneratorAuto id = new NoGeneratorAuto();  
//            Identity id = new Identity();  
//            TableStrategy id = new TableStrategy();  
//            CustomIDGenerator id = new CustomIDGenerator();  
  
           id.setText("I am text");  
  
He clicks Run and the results are displayed in the Console: Created id is 0.  
  
The presenter opens the Command Prompt to display the results table. It has a single entry with an id of zero.  
  
In Hibernate, the presenter clicks Run again. The results in the Console show that there is a duplicate entry of zero for the primary key.  
  
The presenter edits a section of the code in the IdentityMain.java document so that the code is:  
  
// Generated IDs in XML  
//            NoGenerator id = new NoGenerator();  
           NoGeneratorAuto id = new NoGeneratorAuto();  
//            Identity id = new Identity();  
//            TableStrategy id = new TableStrategy();  
//            CustomIDGenerator id = new CustomIDGenerator();  
  
He selects the NoGeneratorAuto.hbm.xml file to view the code:  
  
<?xml version="1.0" encoding="utf-8"?>  
<!DOCTYPE hibernate-mapping PUBLIC   
"-//Hibernate/Hibernate Mapping DTD//EN"  
"http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">   
  
<hibernate-mapping package="hibernate.domain.identifier.generator">  
   <class name="NoGeneratorAuto" table="Generic\_auto">  
<id name="id"/>  
<property name="text"/>  
   </class>  
</hibernate-mapping>  
  
The presenter points out that in this example he is using the "Generic\_auto" table, which adds an auto incrementer on the Id.  
  
He switches back to the IdentifyMain.java file and clicks Run. The results are displayed in the Console, which shows that the created Id is zero. He clicks Run again and the results in the Console show that the created id is still zero.*   
  
...you can see in the second table now, the first table is only id 0, the second table is 1, 2 and if I ran it, it got to be, 3, 4, 5 and so on. Now I can correct that next problem by going to the identity generator. So we can see inside of here I have my id XML mapping, with a submapping called generator, and the class has several different options. Identity means, just what we're talking about, use the Id that gets created in the database table. So after you do an insert, do a query and see which Id just got inserted inside of there. So if I go to my third example of objects inside of here, my identity approach, and I run this guy, now, it runs the exact same table, but it returns to me Id of number three. If I run it again, it will return to me an Id of number four. I look at my database table again, and you can see, it's inserted into that table 3 and 4. But now since I've mapped it intelligently, my Java code is aware of the Id that got mapped inside of there.   
*The presenter opens the Command Prompt and a new table has been generated. The table has two rows. The id for the first row is 1, the id for the second row is 2.  
  
He switches back to Hibernate and clicks the Identity.hmb.xml file. The document displays the code:  
  
<?xml version="1.0" encoding="utf-8"?>  
<!DOCTYPE hibernate-mapping PUBLIC   
"-//Hibernate/Hibernate Mapping DTD//EN"  
"http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">   
  
<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>  
  
He highlights the xml sub mapped identity generator property in the line of code:  
  
<generator class="identity"/>  
  
The presenter switches back to the IdentiyMain.java file and edits this section of the code.   
  
// Generated IDs in XML  
//            NoGenerator id = new NoGenerator();  
           NoGeneratorAuto id = new NoGeneratorAuto();  
//            Identity id = new Identity();  
//            TableStrategy id = new TableStrategy();  
//            CustomIDGenerator id = new CustomIDGenerator();  
  
The new code is:  
  
// Generated IDs in XML  
//            NoGenerator id = new NoGenerator();  
//            NoGeneratorAuto id = new NoGeneratorAuto();  
           Identity id = new Identity();  
//            TableStrategy id = new TableStrategy();  
//            CustomIDGenerator id = new CustomIDGenerator();  
  
He clicks the Run button and the results are displayed in the Console. It shows that the created id for the example is 3. The presenter clicks the Run button a second time and the Console view shows that the created id is now 4.  
  
He switches to the Command Prompt. A new table has been generated. The table has four rows. Their respective ids are 1, 2 , 3, and 4.*   
  
Now there's another strategy called sequence, which is mostly specific to Oracle, and that strategy requires me to pre-query for that Id, and then Hibernate will get the query, the Id beforehand, insert it in the object, then insert the object. We're using MySQL for the demo, so we'll just talk about that one, but that strategy will just basically replace the word identity here with the word sequence. The next strategy we can talk about is the TableStrategy, and in this TableStrategy, it's using the high-low table generator. The high-low table looks like this, where it needs a value of what is the name of the table, and we'll look at the table I just created, and then what's the next high value inside of there. The high-low basically has a bank of Ids it grabs, and it will insert until it uses up the bank and then it's going to go get the next bank. And this'll help manage that, that table helps to manage the Ids. It makes it a little bit faster, so the database doesn't have to be queried at every single insert. And so within the TableStrategy, if I go and insert the number there, and run this guy, it inserts an Id of 32768.   
*The presenter switches back to Hibernate and selects the Identiy.hmb.xml file, which contains the code:  
  
<?xml version="1.0" encoding="utf-8"?>  
<!DOCTYPE hibernate-mapping PUBLIC   
"-//Hibernate/Hibernate Mapping DTD//EN"  
"http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">   
  
<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>  
  
He highlights the xml sub mapped identity generator property in the line of code:  
  
<generator class="identity"/>  
  
The presenter selects the IdentityMain.java file and edits the following code:  
  
// Generated IDs in XML  
//            NoGenerator id = new NoGenerator();  
//            NoGeneratorAuto id = new NoGeneratorAuto();  
           Identity id = new Identity();  
//            TableStrategy id = new TableStrategy();  
//            CustomIDGenerator id = new CustomIDGenerator();  
  
The new code is:  
  
// Generated IDs in XML  
//            NoGenerator id = new NoGenerator();  
//            NoGeneratorAuto id = new NoGeneratorAuto();  
//            Identity id = new Identity();  
           TableStrategy id = new TableStrategy();  
//           CustomIDGenerator id = new CustomIDGenerator();  
  
He switches to the Command Prompt and calls up the HiLo table. The table has two columns. The values for row 1 are:  
  
tableName: Generic.  
nextHi: 1.  
  
The presenter switches back to Hibernate and clicks the Run button. The results are recorded in the Console view. The result shows that the created id is 32768.*   
  
If I go insert another one, it, because it's running from scratch, it's going to put it at an equally huge number of 65000 outside of there. If I look at those in the database, that's because of the logic that's going on inside of there, it's using the Generic table and it's inserting those Ids, so it's not auto incrementing, it's being managed outside of that. But the cool thing is, if I had a web system, or an active system, that inserted a thousand objects, they would be in sequential order, and they would very, very quickly and efficiently grab those things. The last strategy we'll talk about here is a custom generator, where you can go through and assign your own class, to be able to generate the solution out there. So I can go to that class and I can show you, I just need to implement the generator, the IdentifierGenerator interface. And inside of here I'm just randomly selecting a number between 1 and a 100, not the smartest generator ever, but it just shows you something you could do. So if you have an algorithm or an approach that you prefer, you can create your own custom solution around that. And so when I run this solution...   
*The presenter clicks the Run button a second time to run the Table strategy code in the IdentiyMain.java document. The result in the Console view shows that the created id is 65536.  
  
He switches to the Command Prompt to view the database table. The first table has been updated and the generated Ids have been added. There is also the Generic table with the tableName and nextHi columns. The tableName value for the current example is Generic and the nextHi value is 3.  
  
The presenter switches back to Hibernate and selects the CustomIDGenerator.hbm.xml file. The document displays the code:  
  
<?xml version="1.0" encoding="utf-8"?>  
<!DOCTYPE hibernate-mapping PUBLIC   
"-//Hibernate/Hibernate Mapping DTD//EN"  
"http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">   
  
<hibernate-mapping package="hibernate.domain.identifier.generator">  
   <class name="CustomIDGenerator" table="Generic">  
<id name="id">  
<generator class="hibernate.domain.identifier.generator.CustomGenerator">  
</generator>  
</id>  
<property name="text"/>  
   </class>  
</hibernate-mapping>  
  
He points out that he assigned his own class, which is hibernate.domain.idenitifier.generator.CustomGenerator.   
  
The presenter moves to the Package Explorer pane. The open directory is XMLMappingIntro and the src folder is expanded. He clicks the hibernate.domain.identifier.generator folder and selects CustomGernerator.java.  
  
The document opens in the document view and displays the code:  
  
package hibernate.domain.identifier.generator;  
  
import java.io.Serializable;  
  
import javax.persistence.Basic;  
  
public class CustomGenerator implements IdentifierGenerator  
{  
   private Random r = new Random();  
  
   @Override  
   private Serializable generate(SessionImplementor si, Object o) throws HibernateException  
  
   {  
       long id = r.nextInt(100);  
       return id;  
   }  
  
}  
  
He points out that the Custom Generator class is implementing the Identifier Generator and it is set to randomly generate a number between 1 and 100.  
  
The presenter switches back to the IdentityMain.java document and edits the following code:   
  
// Generated IDs in XML  
//            NoGenerator id = new NoGenerator();  
//            NoGeneratorAuto id = new NoGeneratorAuto();  
//            Identity id = new Identity();  
           TableStrategy id = new TableStrategy();  
//           CustomIDGenerator id = new CustomIDGenerator();  
  
The new code is:  
  
// Generated IDs in XML  
//            NoGenerator id = new NoGenerator();  
//            NoGeneratorAuto id = new NoGeneratorAuto();  
//            Identity id = new Identity();  
//            TableStrategy id = new TableStrategy();  
           CustomIDGenerator id = new CustomIDGenerator();  
  
He clicks the Run button.*   
  
...again, it's going to pick a random number, and so this time it created the Id of 34. I can run it again, and it's going to create an id of whatever random number it happens to pick, 85. No guarantee it works, eventually I will run out of numbers and get a duplicate key, but it's not a smart strategy, it's just showing you the approach towards that, but again, I'm inserting those numbers into my first table inside of there. So regardless of which strategy you pick, you probably should choose one of them, and there's a lot of great options for being to able manage your Ids outside of your solution, your code, but let it auto happen inside of Hibernate in the database.   
*The output is displayed in the Console. It shows that the created id is 34. He clicks the Run button again and the created id is 85.  
  
The presenter switches to the Command Prompt and the new created ids have been inserted into the database table.*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a02_it_enus/output/html/course_transcript.html#top)

Custom SQL in Annotations

Learning Objective

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

* *use custom SQL in annotations in Java Hibernate*

**1. Customizing SQL annotations**

Generally Hibernate has two fantastic features that enables for you. The first of which is it does all of the mapping of result sets back into Java objects for you. The second is it creates for you all your basic insert, update, delete statements, your CRUD stuff. Create, read, update, and delete is what CRUD stands for. Now sometimes though, you don't need that, because either you have a need to customize that in some special way, or you have a different approach towards doing those things. Perhaps you're calling stored procedures, or you need to otherwise do some custom behavior inside of there. Never fear, I am not limited by that in Hibernate. In Hibernate, I can go through and replace any of those solutions if I need to. And so here I have a Story class, which we've mapped before as a regular class, but I'm now going to replace the insert, the update, and the delete. There's also a query option, but that's not really a replacement, so we're not going to discuss that here, but just so you know it's available. The insert option then, allows us to go through and replace anything that Hibernate would normally generate with our own custom SQL. This is not HQL, this is SQL, just to make sure we're really aware of that. We're doing our own custom to the target platform, SQL language that we're going to do inside of there. And so we can use ANSI SQL, or it can be specific to whatever database you happen to be targeted to.   
*Hibernate is open and the documents tab contains two files: StoryCustomStatements.java and CustomSQL.java.  
  
The StoryCustomStatements.java document is currently open and displays the code:  
  
package hibernate.domain;  
  
import java.util.Date;  
  
@Entity  
@Table(name="Story")  
@SQLInsert(sql="insert into story (creationDate, storyDate, lastModifiedDate, title) values" + "(?, ?, ?, ?)")  
@SQLUpdate(sql="call specialStoryUpdateAnnotations(?, ?, ?, ?, ?)")  
@SQLDelete(sql="delete from story where id =?")  
public class StoryCustomStatements  
{  
   @Id  
   @GeneratedValue   
   private Long   id;  
  
   @Basic  
   private String title;  
  
The presenter points out the @SQL annotations and descriptions. He notes that with the description for the @SQLUpdate annotation he is calling a stored procedure.*   
  
In this case, the SQL statement's really not all that different than what we would be doing in a normal SQL statement that would be generated by Hibernate, but just to show you that we can replace that. The second one is the update, it's basically replacing the update, but the format is the same. I have a SQL statement I'm going to replace it with. In this case, I'm doing something special, I'm calling a stored procedure, and the stored procedure is going to do some sort of special update with whatever rules it has going on there. Now the one thing that's really tricky about this is I have to provide all the parameters to this. And which is why stored procedures become very, very useful. For instance, if there's certain fields I might not want to update, I still have to have that passed into my SQL. The last item here is delete, and in delete, this is just a basic general SQL statement, but it shows you how that could be done. Again it could be a stored procedure, it could be other custom SQL. The only thing that is passed into the SQL for delete though, is the id. Let's see the stuff in action. So if I come over to my sample and I go, I'm going to go ahead and run that as I walk through it. To do an execute and use these requires no special code. To save an object, I call save, persist, or save and update, any one that I normally would call. In order to update an object, I can call update, or save, or update, whatever I would normally do inside of here. So in this case, I'm calling an update, and then to delete, I just call delete on the object.   
*The presenter points out the @SQL annotations and the descriptions in the code:  
  
@SQLInsert(sql="insert into story (creationDate, storyDate, lastModifiedDate, title) values" + "(?, ?, ?, ?)")  
@SQLUpdate(sql="call specialStoryUpdateAnnotations(?, ?, ?, ?, ?)")  
@SQLDelete(sql="delete from story where id =?")  
public class StoryCustomStatements  
  
He notes that with the description for the @SQLUpdate annotation he is calling a stored procedure.  
  
He selects the CustomSQL.java file and the document opens displays the code:  
  
package hibernate;  
  
import hibernate.domain.StoryCustomStatements;  
import hibernate.util.ConfigHelper;  
  
import java.util.Date;  
  
import org.hibernate.Session;  
import org.hibernate.SessionFactory;  
  
public class CustomSQL  
{  
   private SessionFactory factory;  
  
   public static void main(String[] args)  
   {  
       new CustomSQL().run();  
   }  
  
   private void run()  
   {  
       try  
       {  
           factory = ConfigHelper.getSessionFactory();  
           Session s = factory.getCurrentSession();   
           s.beginTransaction();    
  
           StoryCustomStatements story = new StoryCustomStatements();  
           story.setTitle("Insert using custom statements");  
           story.setStoryDate(new Date());  
             
           Long id = (Long) s.save(story);  
  
           s.getTransaction().commit();  
           s = factory.openSession();  
           s.beginTransaction();  
             
           story = (StoryCustomStatements) s.get(StoryCustomStatements.class, id);  
           System.err.println(story.getTitle() + " " + story.getLastModifiedDate());  
           s.evict(story);  
  
           StoryCustomStatements updatedStory = new StoryCustomStatements();  
           updatedStory.setId(new Long(1));  
           updatedStory.setTitle("Update the story now");  
           updatedStory.setStoryDate(story.getStoryDate());  
             
           s.update(updatedStory);  
  
           s.getTransaction().commit();  
           s = factory.openSession();  
           s.beginTransaction();  
  
           story = (StoryCustomStatements) s.get(StoryCustomStatements.class, id);  
           System.err.println(story.getTitle() + " " + story.getLastModifiedDate());  
  
           s.delete(story);  
             
           s.getTransaction().commit();  
       } catch (Throwable t)  
       {  
           t.printStackTrace();  
           factory.getCurrentSession().getTransaction().rollback();  
       } finally  
       {  
           // To kill the run else Hibernate leaves it active.  
           factory.close();  
           System.exit(0);  
       }  
   }  
}  
  
The presenter clicks the Run button and the following results are displayed in the Console:  
  
Hibernate: select storycusto0\_.id as id1\_3\_0, storycusto0\_.creationDate as creation2\_3\_0\_. storycusto0\_.lastModified...  
  
Insert  custom statements 2014-02-24 08:38:00.0  
  
Hibernate: Call specialStoryUpdateAnnotations(?, ?, ?, ?)  
  
  
Hibernate: select storycusto0\_.id as id1\_3\_0, storycusto0\_.creationDate as creation2\_3\_0\_. storycusto0\_.lastModified...  
  
Update using custom statements 2014-02-24 08:37:59  
  
Hibernate: delete from story where id = ?.  
  
In the document code, he highlights the update command:  
  
           s.update(updatedStory);  
  
He also points out the delete on the object in this line of code:  
  
           s.delete(story);*   
  
There is no special Java calls just because I'm replacing the SQL, that all happens behind the scenes. But down here in the stack trace, you can see I'm actually doing the call to a stored procedure, for instance, when I do the update, when I'm calling the update inside of here. And so it's doing all the work it would normally do, only replacing the actual database calls, instead of using the Hibernate generated one, it's using the ones that I created. And it's going to be able to give me that much more flexibility and power and control for really understanding those queries. Now where would I use this, last real quick point here. Mostly you don't need to use it, but where you really want to use it is when you have that special need of a database to tweak performance, or a database administrator that really wants you to use a query in a specific format. This allows you to insert that right here, and to be able to customize that and still take advantage of all the rest of the great places Hibernate abstracts out the SQL from your Java code.   
*The presenter focuses on the stack trace open in the Console view. He highlights the call to a story procedure displayed in the stack trace as, call specialStoryUpdateAnnotations(?, ?, ?, ?, ?).*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a02_it_enus/output/html/course_transcript.html#top)

Create an Index in Hibernate

Learning Objective

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

* *create a database index in Java Hibernate*

**1. Creating indexes**

If you're choosing to have Hibernate be responsible for creating your tables, creating the DDLs, and setting up your tables in the database, you don't want to have to sacrifice the ability to use indexes as part of that solution as well. Indexes are great for searching, they're great for performance enhancements, all the great things that you can find out about on indices are still good as part of Hibernate. So inside of Hibernate, as part of the table definition, we can actually create indices that can be created by Hibernate. And so you can see in this example from the annotations, I can create an index that links to the person Social Security number. So if I want to search on somebody's Social Security number I can have an index to help me do that. And the index allows me to name the index and then put the list of columns; there could be more than one, in which the index is responsible for. So just to see this in action, I have a database with no tables inside of it. I'm going to execute my application which will then go through and create those tables. And it also inserted some data along the way, so if I look at the show tables again, now I have my table. My table was created here...   
*Hibernate is open there are three documents available in the document view: IndexMain.java, ComplexPerson.java, and ComplexPerson.hmb.xml.  
  
The ComplexPerson.java file is selected and the code is displayed:  
  
package hibernate.domain.index;  
  
import javax.persistence.Basic;  
  
@Entity  
@Table(name="ComplexPerson",  
      indexes = {@Index(name="person\_ssn\_index", columnList="ssn")})  
public class ComplexPerson  
{  
   @Id  
   @Column(name="remoteId")  
   private long id;  
     
   @Basic  
   private String name;  
     
  
   @Basic  
   @Column(name="ssn")  
   private String ssn;  
     
   @Basic  
   private String zip;  
     
     
   public long getId()  
   {  
       return id;  
   }  
  
   public void setId(long id)  
   {  
       this.id = id;  
   }  
  
   public String getName()  
   {  
       return name;  
   }  
  
   public void setName(String name)  
   {  
       this.name = name;  
   }  
  
   public String getSsn()  
   {  
       return ssn;  
   }  
  
   public void setSsn(String ssn)  
   {  
       this.ssn = ssn;  
   }  
  
   public String getZip()  
   {  
       return zip;  
   }  
  
   public void setZip(String zip)  
   {  
       this.zip = zip;  
   }  
}  
  
The presenter points out the @Table annotation and definition and the indexes that have been added below it. There is an @index annotation and it's definition as displayed in the following section of code:  
  
@Table(name="ComplexPerson",  
      indexes = {@Index(name="person\_ssn\_index", columnList="ssn")})  
  
The presenter switches to the Command Prompt to view the result of the show tables command. At the moment there is a database without any tables.  
  
He switches back to Hibernate and selects the IndexMain.java file. IT contains the code:  
  
public class IndexMain  
{  
   private SessionFactory factory;  
  
   public static void main(String[] args)  
   {  
       new IndexMain().run();  
   }  
     
   private void run()  
   {  
       try  
       {  
           factory = ConfigHelper.getSessionFactory();  
           Session session = factory.getCurrentSession();  
           session.beginTransaction();  
  
           for (int i = 0; i < 50; i++)  
           {  
               session.persist(createComplexPerson());  
           }  
             
           session.getTransaction().commit();  
       } catch (Throwable t)  
       {  
           t.printStackTrace();  
       } finally  
       {  
           // To kill the run else Hibernate leaves it active.  
           factory.close();  
           System.exit(0);  
       }  
   }  
     
   private int idIndex = 1;  
   private ComplexPerson createComplexPerson()  
   {  
       ComplexPerson c= new ComplexPerson();  
       int index = idIndex++;   
       c.setId(index);  
       c.setName("CP " + index);  
       c.setSsn("1234567"+ index);  
       c.setZip("12345");  
         
       return c;  
   }  
}  
  
He clicks Run. The results are displayed in the Console. He switches back to the Command Prompt and runs the show tables command. The output is a single table with one column: Tables\_in\_bignews. The value is complexperson.  
  
He runs the describe complexperson command. The output is a table with five columns: Field, Type, Null, Key, Default, and Extra. There are four rows and the Field column values for the rows are remoteId, name,ssn, and zip.*   
  
...and it has all of the attributes that I want out there. And I can actually see by doing a show index – this is a MySQL command here – on that table, the index that was created. So the name you can see, matches what I had inside of my annotation. It creates that key name for the index, it puts it on the column for that, and then all the data that was created is in there, in that table, and is indexed appropriately for the Social Security number. Now the Social Security numbers are made up here obviously, I'm just doing them sequentially, but as I do my search I'd have all the great performance tuning that comes from that. And I'm not limited to do that through annotations. I can do the same thing in XML mapping as well. And so in the XML mapping version of that, I simply am attaching the index to the property definition itself. I don't do that at the top, at the class definition, I do that down on the property definition itself. The rationale behind that's different design; it's just a different way of doing things and we can see that.   
*The presenter runs the command: show index from complexperson.  
  
The output is a table with columns that include: Key\_name, Seq\_in\_index, and Collation. He points out in the Key\_name column, that the primary key has been created with key name of person\_ssn\_index value.  
  
He runs the command:  
  
select \* from complexperson  
  
A table is created with the data requested. The table has four columns: remoteId, name, ssn, and zip.  
  
The presenter switches back to Hibernate and clicks the ComplexPerson.hbm.xml file to open the document. It contains the code:  
  
<?xml version="1.0" encoding="utf-8"?>  
<!DOCTYPE hibernate-mapping PUBLIC   
"-//Hibernate/Hibernate Mapping DTD//EN"  
"http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">   
  
<hibernate-mapping package="hibernate.domain.index" >  
   <class name="ComplexPerson" table="complexPerson">  
       <id name="id" column="remoteId"/>  
       <property name="name"/>  
       <property name="ssn" index="xml\_person\_ssn\_index"/>  
       <property name="zip"/>  
   </class>  
</hibernate-mapping>  
  
He points out that in the xml mapping version, he is attaching the index to the property definition as displayed in the code:  
  
       <property name="ssn" index="xml\_person\_ssn\_index"/>*   
  
And so I can show you this real quick in the XML mode as well by just swapping out the way I've configured my mapping. So right now you can see in my mapping I've linked to a lot of classes, but the one we care about is the ComplexPerson inside of here. So instead of using the ComplexPerson annotation's mapping, I'm going to swap that out with my XML mapping. I can go through now and then I can drop my database. I can create it again, and just to validate no tables are out there. Run it one more time, and we can see I have my table created again, and I have my index created again, only this time you can see it's created from XML. And so the XML annotation's version can also be used to create…not annotations...the XML mapping version can also be used to create the setup for an index. So either way you want to set it up, either through annotations or through XML. We can set up indices inside of Hibernate as we create our tables and have a fully complete, complex, robust database design.   
*The presenter switches to the Package Explorer pane. The General folder contains the src folder. He opens the hibernate.cfg.xml file in document view.  
  
He scrolls through the xml mapping code and highlights the part of the code that maps the complexperson index:  
  
<mapping class ="hibernate.domain.index.ComplexPerson:/>  
  
He swaps the complex person mapping annotation out and replaces in the xml with the line of code:  
  
<mapping resource="hibernate/domain/index/ComplexPerson.hbm.xml"/>  
  
The presenter switches to the Command Prompt and runs the command:  
  
drop database bignews  
  
He runs the show tables command to verify that there are no tables in the database.  
  
He switches back to Hibernate and opens the IndexMain.java file and runs the code.  
  
He returns to the Command Prompt and runs the command:  
  
describe complexperson  
  
The output is a table with six columns: Field, Types, Null, Key, Default, and Extra. There are four rows and their values in the Field column are remoteId, name, ssn, and zip.  
  
The presenter runs the command:  
  
show index from complexperson  
  
The index table is generated. There are a number of columns including Key\_name, Seq\_in\_index, and Column\_name. He points out the primary key index has been created but that the index key name value is now xml\_person\_ssn\_index.*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a02_it_enus/output/html/course_transcript.html#top)

Detailed Property Mapping with Annotations

Learning Objective

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

* *define property mapping using annotations in Java Hibernate*

**1. Defining property mapping**

To get accurate mapping of objects, we need to be able to extend our attributes, much more detailed in how they map to the columns in databases. And so past just the @Basic here, which is basically used to map similar data types between Java and the database with exact name matches, the name of the attribute has to match the name of the column exactly, I can use additional features. Now the first one we'll talk about here actually doesn't map to the database at all, it's @Transient. And this is a way of telling the Hibernate, "hey don't worry about this attribute, ignore this field, it doesn't go to the database at all." It's something I need on the Java side, but it doesn't need to be persisted. In this case it's like a count of something, or something like that, that only has to do with runtime execution. Now beyond that I can get into more detailed types, and if I we look at our Story class, you can see we have a TemporalType. So the first date we have in here is the storyDate, which all we really care about is year, month, day. We can use @Basic annotation on that, that works fine. Or I can choose as this *creationDate* goes, again it's just year, month, day, but I can use the @Temporal option to map to a date, and this will just store year, month, day inside of the database. If I want the time out there, I can go to the full *TIMESTAMP* type, which this is year, month, date, hour, minute, second as well that's loaded. And there's a couple of different types I can choose from there, but that gives me the next level of mapping.   
*Hibernate is open and there are three documents open in the document view: Generic.java, Story.java, and GenericMain.java.  
  
Generic.java is selects and its code is displayed:  
  
   @Id  
   private long id;  
     
   @Basic  
   private String text;  
     
   @Basic  
   @Column()  
   private Date aDate;  
     
   @Basic  
   private double sampleMoney;  
     
   @Basic  
   private boolean aBoolean;  
  
   @Transient  
   private int countNotPersisted;  
  
The presenter points out the @Basic annotation as well as the @Transient annotation. He selects the Story.java file. The document contains the code:  
  
   @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)*   
  
You can actually use the @Type annotation as well to map to any class you want to, and there's some custom type stuff inside of there that we're not going to discuss here. But we're not limited to that. There's actually an additional annotation inside of here, which I can get more specific even on these @Basic fields, it's the @Column annotation. And so we can see from our javax.persistence, this is the JPA definition, that we can add in the column, and there's a lot of different features I can add in from this. And so I can define the column exactly how I want it to be defined, particularly if I'm doing my own database creation. Or I can put the type outside of there, or I can put what table it comes from, or I can put the name of the column. So if the name does not match, and in this case it does in my database, I can say this is the name of the column outside of there. If it's something different than it, then I can make it something different, if it's the same, I can omit this. But another interesting feature is I can tell whether this column is updatable or not.   
*The presenter switches back to the Generic.java file. The current code is:  
  
   @Basic  
   private String text;  
     
   @Basic  
   @Column()  
   private Date aDate;  
     
   @Basic  
   private double sampleMoney;  
     
   @Basic  
   private boolean aBoolean;  
  
   @Transient  
   private int countNotPersisted;  
  
He edits the code by adding an @Column annotation below the first @Basic annotation. As he types @Column, a pane of options opens, he selects the Column - javax.persistent option. Next he types () top open the definition options.  
  
The definition options pane opens and there are a number of options, including type, name, columnDefinition. He chooses the name:String-Column option and types name="text", updatable=true).  
  
The edited code is now:  
  
   @Basic  
   @Column(name="text", updatable=true)  
   private String text;*  
  
There's a separate feature for insertable, and this tells the annotation, tells Hibernate, whether or not as it's generating queries, I should allow this column to be updated. So as it stands, I can make updatable=true, and this is the default. This is not doing anything, this is how it would. And so if I go through and run my test case here, my test case will create an object, it will save the object, and then it's going to update the text of the object and then update that in the database. So my two objects that I'm creating, a simple object and I'm creating a full object with everything filled inside of there. And if I look at my database, you can see I have a simple object created, where the date is null, the money is zero, the Boolean is empty, or I have a full object where everything's been done. And I don't just have the basic text inside of here; I have the updated text, based off of not the text I initially put inside of there, full object, but the updated full object text. So let's go through and update our database, and we're going to go back and change this, where our column is no longer updatable. So we're going to say updatable=false.   
*The presenter selects the GenericMain.ava file, which contains the code:  
  
           {  
               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 = (Generic) o;  
               System.err.println(g.getId() + " " + g.getText());  
           }  
  
He clicks Run and the results are displayed in the Console view:  
  
0 Simple Object  
1 Update Full Object  
  
He points out in the code in the java document that the object is created, then with s.save(g); it is saved.  
  
The text is updated and the database is updated by the line of code:  
  
               g.setText("Update Full Object");  
               s.update(g);  
  
He switches to the Command Prompt to show that a table has been created in the database.   
  
The table has five columns: id, text, aDate, sampleMoney, and aBoolean. There are two rows.  
  
The first row is:  
  
id: 1  
text value: simple object.  
aDate: null.  
sampleMoney: zero.  
aBoolean: blank.  
  
The second row is:  
  
id: 1  
text value: update full object.  
aDate: 2014-02-24.  
sampleMoney: 199.99.  
aBoolean: a boolean value.  
  
The presenter points out that the text value is not just the basic text but the Update Full Object text. He switches back to Hibernate and highlights the code in the GenericMain.java document.  
  
He highlights the initial text description, which is setText("Update Full Object");. He then highlights what is reflected in the table which is the setText("Update Full Object");.  
  
The presenter switches back to the Command Prompt and types source setup.txt. He then switches back to Hibernate and selects the Generic.java file. He edits the code by changing the updatable value from true to false:  
  
   @Basic  
   @Column(name="text", updatable=true)  
   private String text;*   
  
So now when I run this code, I will insert this just fine, but it will forbid me from updating it here, and let's see the interesting results that comes along. So when I run my code, it inserts the object and it actually prints it out as if it was updated. So in the Java side of things, it doesn't prevent me from calling a setText, but if I go and look at the database side of things, it still says full object, where if you remember from the previous example, it said updated full object. And that's because when I did the update inside of there to this and hit save, notice there's no call to do an update in the database, because Java and Hibernate have figured out, oh yeah I called this method, but I'm not allowed to change it, so there's no reason to call update on that. And so I have not actually updated that table at all with that object. So a great little feature to be able to control columns that might be able to be created, but might be read-only after that. I can also do things where I can't insert and such like that. The @Column gives me a lot of control, so research that a little bit more to find out exactly how to map and control your annotations and your columns and your mappings inside of Hibernate.   
*The presenter selects the GenericMain.java file. He clicks the Run button and the results are displayed in the Console view. The results are the same as before, a Simple Object and an Update Full Object  have been called.  
  
He switches to the Command Prompt to view the results in the database. The table is generated with the same five columns as before and two rows, but this time the text says Full Object rather than Updated Full Object.*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a02_it_enus/output/html/course_transcript.html#top)

Lazy Loading

Learning Objective

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

* *configure a fetching strategy by defining lazy and eager loading using either annotations or XML mapping in Java Hibernation*

**1. Using fetching strategies**

If all the data we have is in one table, it's pretty cheap to load it all; it's all the same columns, it's a single query. But when there's joins that are involved, it adds a certain amount of layers of overhead to that query. Hibernate helps us manage that by allowing those joins, those related objects, to be loaded under a couple different strategies. We can see here inside of the one-to-many relationship we have here with the other object, there's a feature to set the fetching strategy. Your fetching strategy can be set to either nothing, which is the default, or EAGER or LAZY. And either one of these will work functionally the same way. As far as the Java is concerned, it's going to load these pages and they're going to be there, and I'm going to use them. There's a difference behind the scenes though on how we do this work. So if I leave it as default, that's the same as leaving it as LAZY. Hibernate's default is what's called lazy loading. So if I explicitly set it to LAZY now and I run it a sample, and this sample does nothing but load a bunch of stories from the database, and then shows all the stories with all the pages inside of there. And so it does a bunch of loading of stories, and then here you can see it does a query to get the short descriptions of all these pages.   
*Hibernate is open and three document tabs are open in document view: Story.java, LazyLoadingMain.java, and Story.hbm.xml.  
  
Story.java is selected and the document contains the code:  
  
   @Temporal(TemporalType.DATE)  
   private Date   creationDate;  
     
   @Temporal(TemporalType.TIMESTAMP)  
   private Date   lastModifiedDate;  
  
   @OneToMany(cascade=CascadeType.ALL), fetch=FetchType.EAGER)  
   @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 points out the fetching strategy is the @OneToMany annotation. The current strategy is set to EAGER. The presenter opens the description options and there are two options: Eager and Lazy.  
  
He selects the LazyLoadingMain.java file and the document displays the code:  
  
   private void run()  
   {  
       try  
       {  
           Session s = factory.getCurrentSession();  
           s.beginTransaction();  
             
           new BulkInsert(s).createABunchForSearching();  
             
           System.err.println("!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!");  
           System.err.println("See all the short descriptions");  
           Criteria c = s.createCriteria(Story.class);  
             
           List<Story> all = (List<Story>) c.list();  
           StoryToString.printAllStories(all);  
  
He clicks the Run button to load each story from the database. The Console displays the following code for each story being loaded:  
  
Hibernate: /\* insert hibernate.domain.Story \*/ insert into Story (creationDate, lastModifiedDate, storyDate, title) values (?, ?, ?, ?)  
  
Hibernate: /\* insert hibernate.domain.Page \*/ insert into Page (creationDate, htmlIncluded,  lastModifiedDate, pageNumber...  
  
Once all of the stories have been loaded, the details for each story are displayed in the Console as follows:  
  
Page 2  
  
Man fist bumps Doughnut – Page 2  
  
creation: 2014-02-16 modified: 2014-02-24 11:3:58.0  
  
Hibernate: delete from Page  
Hibernate: delete from Story  
Hibernate: ALTER TABLE Story AUTO\_INCREMENT = 1  
Hibernate: ALTER TABLE Page AUTO\_INCREMENT = 1.  
  
The presenter scrolls through the code and highlights the view all of the short descriptions query. There are a hundred separate queries, one for each story.   
  
Next, the data for each story is loaded, page by page, and includes the page number, title, and creation date of the story been loaded from the database.*   
  
So after it loads the pages, it does a whole bunch, you can see it keeps going for a very long time. There's actually a hundred of them here, because I know I created 100 pages in my database, 100 queries to go load the page data, and then we get around to loading the information. So in this case it's a very inefficient use of resources, as I've made 101 queries to the database to be able to load 100 stories and their pages inside of here. So it's undesirable, this use of the lazy loading is very undesirable. So let me change that to EAGER instead. So if I change the FetchType to EAGER, and then I go and run my solution again, it's going to go load those stories and it's going to go through and do the query. So it's inserting all these stories in my test cases first, and then it's going to go load all the stories as a single statement. And so I have one big statement that does some left outer joins with the page table, and loads all the pages up at once. And then it reconciles those into my individual object, and you can see one query gives me all 100 objects with all their pages filled in.   
*The presenter points out that in the trace stack in the Console there is a list of a hundred queries being generated, one for each page in the database.  
  
He switches back to the Story.java file and changes the FetchType description from Lazy to Eager. He then selects the LazyLoadingMain.java file and clicks the Run button.  
  
The results are displayed in the Console view. The Trace Stack shows the stories being loaded. It then puts out a query to see all the short descriptions. This time instead of generating a hundred separate queries, one for each page, it now loads all the queries as a single statement.  
  
Then all of the information of the pages are loaded into the trace stack.*   
  
So the nice part about this second query is I'm not hitting the database over and over and over again, but I am returning back a heck of a lot of data. So which one's the right one, and that's a trick answer, a trick question. The answer is simply, it depends what you're doing. If I may or may not use the second-level details of that object, I probably want to lazy load. If I know I'm going to use the second-level details of that object, I probably want to eagerly fetch those. And I need to pick which one. Well I don't want to pick which one each and every time because it's not always used the same, so I can manage the fetch strategy in a number of different ways. So we saw the annotation's strategy for this, we saw how I can change that within the annotations. I can also do it inside of my XML mapping, because there's an attribute on the set, the list, the map, those different mappings for lazy, I can make that "true" or "false". But well beyond that, there's also the @FetchProfile option.   
*The presenter selects the Story.hbm.xml file and scrolls to the section of code:  
  
</id>  
       <property name="title"/>  
       <property name="storyDate"/>  
       <property name="creationDate"/>  
       <property name="lastModifiedDate"/>  
  
       <set name="pages" table="page" cascade="all"  
               inverse="true" lazy="true" fetch="select">  
           <key>  
               <column name="storyid" not-null="true" />  
           </key>  
           <one-to-many class="hibernate.domain.Page" />  
       </set>  
   </class>  
</hibernate-mapping>  
  
He highlights the code where the loading type is mapped in the XML:  
  
<set name="pages" table="page" cascade="all"  
               inverse="true" lazy="true" fetch="select">*  
  
And so the @FetchProfile is an annotation that you can put on the class as a whole. And I'm not going to build the whole thing, but I want to let you see the annotation to start with. The @FetchProfile allows you to set up a series of overrides, and those overrides can be chosen in your code based off of the work you have going on. So if I know I'm going to do a query that's going to need all that second-level data, I go grab the @FetchOverride that allows me to make it EAGER loading. And so I can keep it LAZY by default and then make it EAGER, or vice versa. If I know I only want a quick look at the data, then I can do that over the top. So between @FetchProfiles and the multiple mappings, we can certainly manage how we're going to load data from the database, whether it's EAGER or LAZY. And so you need to make those choices, you need to make those designs, but Hibernate gives you the tools in which you can switch back and forth very easily.   
*The presenter switches back to the Story.java file and examines a section of the code:  
  
   @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}  
                             )  
})  
public class Story   
{  
   @Id  
   @GeneratedValue  
   private long   id;  
  
He edits the last part of the code by inserting a @FetchProfile annotation so that it becomes:  
  
})  
@FetchProfile()  
public class Story   
{  
   @Id  
   @GeneratedValue  
   private long   id;  
  
He then brings up the options for the description pointing out that you can set it to lazy or eager.*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a02_it_enus/output/html/course_transcript.html#top)

Defining with the Entity Annotation

Learning Objective

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

* *refine basic mapping in JPA and Java Hibernate*

**1. Mapping with JPA**

Annotations are driven by the JPA definition, but Hibernate has extensions to allow you to further refine the mapping of an entity. So let's see how Hibernate customizes the basic mapping provided by JPA. The core of mapping using annotations is the @Entity tag, and as you can see, the @Entity tag actually comes from JPA, it doesn't come from the Hibernate framework at all. Nothing wrong with that, Hibernate has just chosen to subvert to that tag as the core of the foundation. Now it doesn't mean we can't use the @Entity tag, we just need to add it, second it to the @Entity tag from JPA. So the base tag from JPA has nothing inside of it except for a name, and that name is just basically a marker inside of there, it doesn't really allow us to customize it before, much must past that. But the @Entity tag from Hibernate, even though it's deprecated, still gives us options to be able to customize our mapping further. So for instance, as an example we can talk about the dynamicInsert and set that to be true, and dynamicUpdate and set that to be true.   
*Hibernate is open and there are two tabs in the documents view: GenericMain.java and Generic.java.  
  
The Generi.java tab is open and the document contains the code:  
  
package hibernate.domain;  
  
import java.util.Date;  
  
import javax.persistence.Basic;  
import javax.persistence.Column;  
import javax.persistence.Entity;  
import javax.persistence.Id;  
import javax.persistence.Transient;  
  
@Entity  
public class Generic  
{  
   @Id  
   private long id;  
     
   @Basic  
   private String text;  
  
The @Entity annotation as well as the import: java.persistence.Entity; tags are highlighted. The presenter inserts a new line into the code below the @Entity annotation and types @Entity.  
  
He opens the template proposals. There are three options: Entity - javax.persistence, Entity - org.hibernate.annotations, and EntityListeners - javax.persistence. The first and the last options are disabled.  
  
He selects Entity - org.hibernate.annotations and the code is changed from @Entity to @org.hibernate.annotations.Entity. The presenter switches back to the @Entity annotation and opens the template proposals box. The only enabled option is name : String - Entity.  
  
He switches back to the JPA Entity tag and reopens the template proposals box. There are seven options: dynamicInsert : boolean - Entity, dynamicUpdate : boolean - Entity, mutable : boolean - Entity, optimisticLock : OptimpisticLockType - Entity, persister : String - Entity, polymorphism : PolymorphismType - Entity, and selectBeforeUpdate : boolean - Entity.  
  
The presenter selects dynamicInsert and dynamicUpdate and sets both values to true. The final line of code is:  
  
@org.hibernate.annotations.Entity(dynamicInsert="true", dynamicUpdate="true")  
  
The code is depreciated.*   
  
Again there's a lot of other options outside of here, whether the class is mutable or not, whether I want to put a different persister out there, the locking strategies, polymorphism, and inheritance strategies. All those things are discussed in different places, but it's all these features I can put on my class. Now these two though, basically allow me, when I execute my queries, when I execute my inserts and updates, I can go through and trim out and truncate any fields to my queries that aren't needed for my solutions. And in my database here, I didn't reset it, so let me do that real quick. And I'm going to execute that one more time, so I don't end up on a primary key clash inside of there, and now I've inserted my values. So we can see, I've got the values inside of here, they've been inserted of it before, but when I do the insert query inside of here, I don't have to provide every single one of the values. Since the first query inside of here only provided the basic ones...it did not provide a date, notice, the date does not show up in the second insert. When I do the update, it's only doing an update on a text value. So if I look at my example inside of here, I'm setting the text, I'm not setting anything else, so I'm only doing the update based off the text value that I sent inside.   
*The presenter selects the GenericMain tab. The document contains the code:  
  
package hibernate;  
  
import java.util.Date;  
  
import hibernate.domain.Generic;  
import hibernate.util.ConfigHelper;  
  
import org.hibernate.Session;  
import org.hibernate.SessionFactory;  
  
public class GenericMain  
{  
   private SessionFactory factory;  
  
   public static void main(String[] args)  
   {  
       new GenericMain().run();  
   }  
  
   private void run()  
   {  
       try  
       {  
  
He clicks the Run button. The Console view displays the trace stack and an exception is thrown. The presenter switches to the Command Prompt and resets the database.  
  
He switches back to Hibernate and clicks Run again. This time the trace stack shows that two objects were created: zero simple object and 1 update full object.  
  
The presenter switches to the Command Prompt and the table has five columns and two rows.  
The first row is:  
  
id: zero.  
text: Simple Object.  
aDate: null.  
sampleMoney: zero.  
aBoolean: blank.  
  
The second row is:  
  
id: 1.  
text: Update Full Object.  
aDate: 2014 - 02 -25.  
sampleMoney: 199.99.  
aBoolean: a Boolean value.  
  
The presenter switches back to Hibernate and scrolls through the trace stack to highlight the inserts. The log displays the feedback:  
  
insert into Generic (aBoolean, sampleMoney, text, id) values (?, ?, ?, ?)  
insert into Generic (aBoolean, sampleMoney, text, id) values (?, ?, ?, ?)  
  
He points out the update in the trace stack, which displays as:  
  
update Generic set text=? where id=?  
  
He scrolls to the code in the GenericMain.java document, which relates to the update in the trace stack:  
  
               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);  
  
He highlights the save and update command, which he has set on the text value.*   
  
So past that, I can further customize my solution, I don't have to just use this deprecated tag, if I don't need those things, I don't need it at all. I can use other JPA tags, such as the @BatchSize. And so the @BatchSize is telling it, for instance, a size of 10, when you're doing your work, when you have an operation that lines up more than 10, or 12, or 15, or a 100, or whatever batch size is, of these entities, of this class, Generic, go ahead and flush it at that point, go ahead and save those to the database, don't wait till there's a thousand, or 10 000, or a million entities to be saved before you flush, always go ahead and flush on that batch size. And then the other one that's fun inside of here is the @Where clause. And so the @Where clause, from the Hibernate annotations in this case, allows me to do something like adding a clause Where, oops, lower case W, I don't need to where at all, I apologize, sampleMoney > 0.   
*The presenter selects the Generic.java file. He edits the code at the point:  
  
package hibernate.domain;  
  
import java.util.Date;  
  
import javax.persistence.Basic;  
import javax.persistence.Column;  
import javax.persistence.Entity;  
import javax.persistence.Id;  
import javax.persistence.Transient;  
  
@Entity  
@org.hibernate.annotations.Entity(dynamicInsert=true, dynamicUpdate=true)  
public class Generic  
{  
  
He inserts a new line below the JPA Entity tab and types @BatchSize(size=10). He then inserts another new line and types @Where and opens the template proposals pane. The options are Where - org.hibernate.annotations and WhereJoinTable - org.hibernate.annotations.  
  
He selects the Where options and inserts the definition:  
  
(clause="sampleMoney > 0")*   
  
So I already have the @Where inside of there, but we are saying, I only care about instances of this class when my money is there. When there's no money, I don't really care about instances of this class. And so the net effect of this is it will still insert, it will still update, it will still do all this work, but when it returns values from that class, up again, I need to clear my database. When it returns values from that class, it's going to do so in such a way that it returns back only values that have a value greater than zero. So you can see it only returned back one of the two objects that I inserted, because when I look at the two objects, it has that object then reflected to the zero object that's not returned back. So all these extra options using the alternative form of the Hibernate entity annotation or the extra annotations that can go on the entity, allow me to customize my entity and really get into specific behaviors and specific mappings that I want from my entity.   
*The presenter adds these annotations to the Generic.java document:  
  
@Entity  
@org.hibernate.annotations.Entity(dynamicInsert=true, dynamicUpdate=true)  
@BatchSize(size=10)  
@Where(clause="sampleMoney > 10")  
  
He switches to the GenericMain.java document and clicks the Run button. The trace stack throws an exception, he switches to the Command Prompt and resets the database. He clicks the Run button in Hibernate again and the trace stack shows 1 Update Full Object.  
  
The presenter switches to the database to show that in the table the update full object is the only object in the database that meets the criteria of the clause.*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a02_it_enus/output/html/course_transcript.html#top)

The ID and Advanced ID Options

Learning Objective

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

* *use advanced ID options in Java Hibernate*

**1. Mapping advanced Ids**

Every Hibernate class requires an Id, and in annotations, it just simply is a matter of putting an @Id tag inside of there, and if you so choose, you can put some sort of generator strategy around that. That's a whole different discussion and really only solves the problem of when your Id is a single column inside of your class. What we need to be able to do, past this basic Id, is allow you to have other options for managing your tables. And some tables that single column, that single primary key just isn't good enough. I need to have, for instance, an external multi-columned Id associated with that. So in this example, we have a, just a generic class that uses some externalized Id that has more than one attribute, more than one column associated with that. If I look at this Id, you can see I have an Id that's based off of, say, the Social Security number's last four digits of a person, and their zip code. So if they're lost on a website, they're trying to log in, they type in their last four digits, they type in their zip code, that's the Id by which I find them. Not the best Id, but it still sometimes is needed in my business description.   
*Hibernate is open and there are five tabs in the document window: IdentityMain.java, identity.java, EmbededID.java, MyID.java, and RemoteID.java.  
  
Identity.java is selected and the code is displayed:  
  
package hibernate.domain.identifier.generator;  
  
import javax.persistence.Basic;  
  
@Entity  
@Table(name="Generic\_Auto")  
public class Identity  
{  
   @Id  
   @GeneratedValue(strategy=GenerationType.IDENTITY)  
   private long id;  
  
The presenter highlights the @Id annotation and also highlights the generation strategy. He selects the EmbeddedID.java file and it contains the code:  
  
package hibernate.domain.identifier;  
  
import javax.persistence.AttributeOverride;  
  
@Entity  
@Table(name="ComplexPerson")  
public class EmbeddedID  
{  
   @EmbeddedId  
   @AttributeOverride(name="lastFour", column=@Column(name="ssn"))  
   private MyID id;  
  
The @EmmbeddedId annotation is highlighted. The presenter also highlights the EmbeddedID class and the MyID attribute  
  
He selects the MyID.java file and examines the code that describes the MyID attribute:  
  
@Embeddable  
public class MyID implements Serializable  
{  
   private String lastFour;  
   private String zip;  
  
   public String getLastFour()  
   {  
       return lastFour;  
   }  
   public void setLastFour(String lastFour)  
   {  
       this.lastFour = lastFour;  
   }  
   public String getZip()  
  
The getLastFour command is highlighted. The presenter highlights the lastFour string description.*   
  
Well I want to represent that ID as a class on its own, and yet I want to be able to use that as a primary key on one or maybe more classes. So the key to doing that is to use the @Embeddable annotation. This is saying this class can be embedded inside of another class as part of an Hibernate mapping. Now notice there's no description of what table it maps to, because it can be used across multiple classes. The description of what columns and rows, and all that stuff that match up, is based off of this @AttributeOverride annotation, that's used inside of this class to say hey, those attributes in the class associated with Id, this is how they are going to be mapped. So for the last four, instead of being lastfour in the database, it's ssn. So I'm going to change the column mapping inside of there. Notice I don't have to provide any override for the second one inside of here, which is the zip, because zip matches exactly in the database. When I look in the database structure, we're going to see zip is what's the column name and it's what's the attribute name. I only have to override when it's different.   
*The presenter examines a section of code in the MyID.java document:  
  
@Embeddable  
public class MyID implements Serializable  
{  
   private String lastFour;  
   private String zip;  
  
   public String getLastFour()  
   {  
       return lastFour;  
   }  
   public void setLastFour(String lastFour)  
   {  
       this.lastFour = lastFour;  
   }  
   public String getZip()  
  
He points out the @Embeddable annotation. He switches to the EmbeddedID.java file and examines a section of code:  
  
@Entity  
@Table  
  
e(name="ComplexPerson")  
public class EmbeddedID  
{  
   @EmbeddedId  
   @AttributeOverride(name="lastFour", column=@Column(name="ssn"))  
   private MyID id;  
     
   @Basic  
   private String name;  
  
   public MyID getId()  
   {  
       return id;  
   }  
  
The presenter points out the @AttributeOverride annotation and the description, which is (name="lastFour", column=@Column(name-:ssn")).  
  
He selects the IdentityMain.java file, which shows the code:  
  
//            Identity id = new Identity();  
//            TableStrategy id = new TableStrategy();  
//            CustomIDGenerator id = new CustomIDGenerator();  
//            id.setText("I am text");  
             
//            long assigned = (Long) session.save(id);  
  
//            System.err.println("!!!!!!!!!!!!!!!!!!!!!!!!!!!!!");  
//            System.err.println("Created id is " + assigned);  
             
           showEmbedded(session);  
             
//            showRemote(session);  
             
           session.getTransaction().commit();  
       } catch (Throwable t)  
       {  
           t.printStackTrace();  
       } finally  
       {*   
  
So let me run a little sample inside of here, and we'll run a little solution, where in this solution, I'm embedding an object, I'm just creating an object with the last four social security number and a zip code, and as you can see it's inserting it into the table right here. And so I can actually go and look at that table, and there you can see I have a social security number, zip code, into that, even though the structure of my source code was that I'm putting the two items into the Id field, and I am setting that whole object into my person. Again, just a little bit of depth and mapping, but not having to create an additional class, an additional map class, which allows that embedded Id to be used across multiple classes. I can share that, I can find different ways to be able to reuse that class in that mapping. So embedded multi-columned keys is through the @Embeddable tag, the other option for being able to set an ID sometimes is I have to have a foreign key associated with that. I have some sort of remote Id that's generated outside of my object. And so if I have a class, a table that's subordinate to another class, within the @Id, I can add additional annotations to that. The first one being the @MapsID.   
*The presenter clicks the Run button and the trace log displays the results.  
  
Hibernate: insert into ComplexPerson (name, ssn, zip) values (?, ?, ?).  
  
He scrolls through the code in the document view to highlight the section:  
  
   private void showEmbedded(Session session)  
   {  
       EmbeddedID person = new EmbeddedID();  
       MyID id = new MyID();  
       id.setLastFour("1234");  
       id.setZip("90210");  
       person.setId(id);  
       person.setName("My Person");  
         
       session.persist(person);  
   }  
  
He points out the Ids that have been referenced and then he points out the insert in the stack trace. It reads:  
  
Hibernate : insert into ComplexPerson (name, ssn, zip) values (?, ?, ?).  
  
He switches to the Command Prompt and a table has been created in the database. It has four columns and one row. The row reads:  
  
remoteID: Null.  
ssn: 1234.  
zip: 90210.  
name: MyPerson.  
  
He switches back to the IdentityMain.java file in Hibernate and highlights the section of code:  
  
       id.setLastFour("1234");  
       id.setZip("90210");  
       person.setId(id);  
  
The presenter switches to the remoteID.java file. It contains the code:  
  
@Entity  
@Table(name="ComplexPerson")  
public class RemoteID implements Serializable  
{  
   @Id  
   @MapsId("remoteId")  
   @JoinColumn(name="remoteId", referencedColumnName="id")  
   @OneToOne  
   private NoGenerator partner;  
  
He highlights the @Id annotation and then the @MapsId annotation.*   
  
And so what we're saying is, I am not the primary keeper of the Id, somebody else is that. The name of that mapped Id is remoteId, because if we looked at that database table from before, we're going to use the same table for this example, but we're going to use the remoteId column, which is going to map off to that other table. And to do that, I need to join with some object that is my parent object, if you will, and in this case I am calling the partner object. And so the other object, the details are unimportant, but we know it has an Id inside of there. So the reference column that I am looking for is the id object from this other table, that I am looking up. And so my remoteId joins with that Id that's out there, in order to be able to have those things aligned in this one-to-one relationship. So what we're saying is, I am kind of a composite object as part of this other guy, even though I'm located in a separate table. The best way to see this is to look at the database table, and the best way to do that is let's run the test and see the net result. So let me uncomment the remote test, and let me go through and run this.   
*In the RemoteID java file, presenter examines the code:  
  
@Entity  
@Table(name="ComplexPerson")  
public class RemoteID implements Serializable  
{  
   @Id  
   @MapsId("remoteId")  
   @JoinColumn(name="remoteId", referencedColumnName="id")  
   @OneToOne  
   private NoGenerator partner;  
  
He highlights the @MapsId annotation and points out the name of the mapped Id in the description, which is remoteId. He switches to the table in the database and points out the remoteId column.  
  
The presenter then highlights the @JoinColumn annotation and links it in relation to the NoGenerator partner object. He highlights the referencedColumnName = "id" attribute in the @JoinTable annotation description, as well as the name = "remoteId".  
  
The presenter selects the IdentifyMain.java file and scrolls to the section of code:  
  
//           System.err.println("!!!!!!!!!!!!!!!!!!!!!!!!!!!!!");  
//            System.err.println("Created id is " + assigned);  
             
           showEmbedded(session);  
             
//            showRemote(session);*   
  
And you can see, the first thing when I run this test, is I need to create my partner object, I need to set up an Id in my partner object, and then I need to save that partner object before I save myself. And so for myself, I'm going to set the partner as being the target, it's going to be an associated object within me, I'm going to set any data I need to in my own object, and then I'm going to save myself second. And so I save into the generic, then I save into the complex person tables over here, which is just my sample table names. And we can see here, now I have the first one saved is my partner – my partner object – that's from the generic table. And then the second one inside of here, you notice, I've done my local data here, but the Id matches the id of that one up above. And that's the key person there, it allows me to reference the other class, through that, relationship, and map this foreign key inside of the database, inside of my class instead, as just a simple association between two classes. So that's three strategies for being able to manage the different types of keys that my business and my domain model might require me to keep those relationships between objects.   
*The presenter examines the code in the IdentityMain.java file:  
  
   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 points out the partner object and the Id . He then points out persist(partner) and persist(sample).  
  
The presenter goes to the information in the trace stack and highlights the inserts. The trace stack shows the information:  
  
Hibernate : insert into Generic (text, id) values (?, ?)  
Hibernate : insert into ComplexPerson (name, ssn, zip, remoteId) values (?, ?, ?, ?)  
  
He switches to the Command Prompt to view the database tables. The first table displays the id, text, date, sample money, and boolean values for the partner object, and the remoteId table now has a second row which shows the data for the My  Sample object. The id column value for the partner object is 1 and it matches the remoteId of the My Sample object.*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a02_it_enus/output/html/course_transcript.html#top)

The Table and SecondaryTable Annotations

Learning Objective

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

* *use @Table and @SecondaryTable in Java Hibernate*

**1. Using @SecondaryTable**

An entity maps to one table by default, and if that entity doesn't match the table name that goes out there, in this case, I have a class called DoubleTableObject, a sample we'll talk about in a second, and a table, Generic, it's going to map to. Then I use the @Table tag to make that alignment. I can change from the default, the assumption that the class name aligns with the table to the specific table that actually aligns to using the @Table tag. And the @Table tag gives me a bunch of other features; I can set up indices, and constraints, and schemas. But I only really care about those things if Hibernate is generating the tables for me, they don't really mean anything if the tables are preset. So in our example here, we have a worse situation than that, we're not even mapping to a different name table, we actually need to map to two different tables. Our schema in the database doesn't align with how we want to represent the objects in our object model. And not necessary a bad thing, it might just be a different use case, it might just be a different need from our Java class, but we definitely want to be able to pull from two different tables. And Hibernate allows us to do that seamlessly within mapping, without having to do extra work inside of Java, inside of our solution, we don't have to load from two tables and merge the objects or anything like that. So the way I can do that is with the @SecondaryTable tag.   
*Hibernate is open and there are two tabs in the document view: doubleTableObject.java and SecondaryMain.java.  
  
The DoubleTableObect.java file is selected and the document displays the code:  
  
package hibernate.domain.table;  
  
import java.util.Date;  
  
@Entity  
@Table(name="Generic")  
@SecondaryTable(name="ComplexPerson",   
               pkJoinColumns={@PrimaryKeyJoinColumn(name="remoteId", referencedColumnName="id")})  
public class DoubleTableObject  
{  
   @Id  
   private long id;  
     
   @Basic  
   private String text;  
     
   @Basic  
   private Date aDate;  
     
   @Basic  
   private double sampleMoney;  
     
   @Basic  
   private boolean aBoolean;  
  
  
The presenter points out the DoubleTableObject class and the table that he is mapping to, which is Generic. He does this with the @Table annotation.  
  
He edits the description of the @Table tag by inserting a comma and then opening the template properties pane. There are four options: catalog, indexes, schema, and uniqueConstraints[]. He closes the pane without selecting an option and deletes the comma from the code.*   
  
And so the @SecondaryTable tag allows me to go to a second table out there and join, based off of some primary key, into the data I have in my first table. So now, if I look at my database, I can say, I have a table out there, Generic, and that forms the first five columns I want inside of my...the five attributes I want inside of my Java object. But I can also look at the complexperson table, and the complexperson table adds in three additional attributes that I want as part of my full object. Again I'm not really making something that makes a lot of sense business wise, I'm just showing you the feature with my DoubleTableObject here. But the net result is I have id, and the four attributes from the Generic table, and then I have three additional attributes that are coming out of the ComplexPerson table. So up top, I've said I need this @SecondaryTable ComplexPerson. And I need the primary key JoinColumns to point from remoteId to the id of the Generic class, again, from our database. The remoteId is a foreign key relationship with the id of the Generic class.   
*The presenter points out the @SecondaryTable annotation in the code:  
  
@Entity  
@Table(name="Generic")  
@SecondaryTable(name="ComplexPerson",   
               pkJoinColumns={@PrimaryKeyJoinColumn(name="remoteId", referencedColumnName="id")})  
public class DoubleTableObject  
{  
   @Id  
   private long id;  
     
   @Basic  
   private String text;  
     
   @Basic  
   private Date aDate;  
     
   @Basic  
   private double sampleMoney;  
     
   @Basic  
   private boolean aBoolean;  
  
He switches to the Command Prompt to view the database. He runs the command:  
  
describe generic  
  
The generic table is created. It has five columns: Field, Type, Null, Key, Default, and Extra. There are five rows and the respective values in the Field column are: id, text, aDate, sampleMoney, and a Boolean.  
  
The presenter runs the command:  
  
describe complexperson  
  
The complex person table is returned. It has the same columns as the generic table and it has four rows: remoteId, ssn, zip, and name.  
  
He switches back to the DoubleTableObject.java file in Hibernate. He points out the five attributes in the generic table. He then scrolls through the code to reveal the attributes that are coming from the complex person table. The code is displayed:  
  
@Basic  
   @Column(table="ComplexPerson")  
   private String name;  
     
   @Basic  
   @Column(table="ComplexPerson")  
   private String ssn;  
     
   @Basic  
   @Column(table="ComplexPerson")  
   private String zip;  
  
He scrolls back to the first part of the code:  
  
@Entity  
@Table(name="Generic")  
@SecondaryTable(name="ComplexPerson",   
               pkJoinColumns={@PrimaryKeyJoinColumn(name="remoteId", referencedColumnName="id")})  
public class DoubleTableObject  
{  
  
He points out the complex person table, which is the secondary table, and he highlights the @PrimaryKeyJoinColumn annotation, which is pointing the remoteId column in the secondary table to the Id column in the Generic table.  
  
He switches back to the tables in the database to point out the relationship between the remoteId key column in the complex person table and the Id column in the Generic table.*  
  
So then I need to let each column know...it's a @Basic mapping...but each column needs to be told which table it's coming out off. So the default is you don't need to put this, it just assumes it's part of the table that's part of the entity mapping, whatever the @Table object is. But when I add in the SecondaryTable, I then have to tell certain columns, and in this case, name, Social Security number, zip, that you come from this other table. When you're doing the query mapping, in the back end in Hibernate, all generated by the Hibernate, it's going to be able to map things appropriately. So now I have an object, a DoubleTableObject, that I'm setting in the attributes, as if it was any other normal Java object. I'm not aware of the database structure, I don't know about the mapping at this point. I'm going to go off and persist that object, and as I go off and do that, let me save that real quick, I run my code, and it's just going to seamlessly drop this in the database.   
*The presenter switches to Hibernate to highlight the basic mapping of the columns to tables in the code:  
  
@Basic  
   @Column(table="ComplexPerson")  
   private String name;  
     
   @Basic  
   @Column(table="ComplexPerson")  
   private String ssn;  
     
   @Basic  
   @Column(table="ComplexPerson")  
   private String zip;  
  
He selects the SecondaryMain.java file and the document contains the code:  
  
public class SecondaryMain  
{  
   private SessionFactory factory;  
  
   public static void main(String[] args)  
   {  
       new SecondaryMain().run();  
   }  
     
   private void run()  
   {  
       try  
       {  
           factory = ConfigHelper.getSessionFactory();  
           Session session = factory.getCurrentSession();  
           session.beginTransaction();  
  
           DoubleTableObject o = new DoubleTableObject();  
           o.setId(1);  
           o.setaBoolean(true);  
           o.setaDate(new Date());  
           o.setName("Double me");  
           o.setSampleMoney(22.22);  
           o.setSsn("4321");  
           o.setText("Test Me double");  
           o.setZip("44444");  
             
           session.persist(o);  
  
He points out the DoubleTableObject and the attributes that he is setting into the table and the persist.  
  
He saves the DoubleTableObject.java file, switches back to the SecondaryMain.jave file and clicks Run.*   
  
It inserts into the Generic table, the five values that goes there, it inserts into the ComplexPerson, the five values...the four values that go there, with the id being one of those. And as I come back to my database here, and I look at the solution, I've inserted the Generic table, I've inserted the ComplexPerson table, and the remoteId aligns with the id of the Generic table, they both are set up. So the mapping simply goes from something very simple, started of at @Entity, adding in the @Table that that points to, adding in any @SecondaryTables, you can have more than one if you so choose, and then going on and mapping the columns from there. So seamlessly, I can think of very complex mapping and make it seem fairly simple, I think, once you get it all set up and get everything aligned, as long as you have the key pieces in place.   
*The output is displayed in the Console:  
  
Hibernate: insert into Generic (aBoolean, aDate, sampleMoney, text, id) values (?, ?, ?, ?, ?)  
Hibernate: insert into ComplexPerson (name, ssn, zip, remoteId) values (?, ?, ?, ?)  
  
Hibernate performs two inserts, one for the values into the columns in the Generic table and one for the values into the columns in the ComplexPerson table.  
  
He switches to the database tables in the Command Prompt and the two tables have been generated. The id value in the Generic table matches the remoteId value in the complex person table.*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a02_it_enus/output/html/course_transcript.html#top)

The class tag Options

Learning Objective

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

* *use class tag options in Java Hibernate*

**1. Exploring class tag options**

The class tag gives us a ton of options for really exerting a lot of control and precise behavior of how we want to map and have actions executed across our different classes and our different tables. And so if we look inside of here, there's so many different options we can choose from. And we won't go through all of them, but there's a few interesting ones I want to pull out, such as dynamic-insert and dynamic-update. This is a cool feature that allows your SQL, your generated SQL statements to your database to be customized based off of what data is actually there in your object. So I'm not just overwriting things randomly, or setting things that don't need to set. So really quick, let me show you what this looks like without that set, where...I don't need to save that...where I'm going to, well, I guess I do need to save that, where I need to go through and run this with extra features. So notice here, I have two insert statements that are filling every single one of these fields, they're passing them along, a slight inefficiency inside of there. I'm going to go look at my database. I have passed over NULL values, here the date specifically has NULL value that's been added. So I can choose to go to my mapping of this, and say dynamic-insert = "true" and so it's going to pick that insert statement, it's going to customize that insert statement based off of what I'm actually sending over. So now when I go and execute this code again...   
*Hibernate is open and there are three files in the document view: GenericMain.java, Generic.java, and Generic.hbm.xml.  
  
The Generic.hbm.xml file is open and contains the code:  
  
<hibernate-mapping package="hibernate.domain">  
   <class name="Generic" table="generic\_auto">  
       <id name="id">  
           <generator class="native"/>  
       </id>  
  
       <property name="text"/>  
       <property name="aDate"/>  
       <property name="sampleMoney"/>  
       <property name="aBoolean"/>  
   </class>  
</hibernate-mapping>  
  
The presenter points out the class tag. He opens the template options window and there are a number of options. He points out: dynamic-insert and dynamic-update.  
  
He closes the window and switches to the GenericMain.java file. The document contains the code:  
  
public class GenericMainXML  
{  
   private SessionFactory factory;  
  
   public static void main(String[] args)  
   {  
       new GenericMainXML().run();  
   }  
  
   private void run()  
   {  
       try  
       {  
           factory = ConfigHelper.getSessionFactory();  
           Session s = factory.getCurrentSession();   
           s.beginTransaction();  
  
He clicks the Run button and the output is displayed in the Console.  
  
Hibernate: insert into generic\_auto (text, aDate, sampleMoney, aBoolean) values (?, ?, ?, ?)  
  
Hibernate: insert into generic\_auto (text, aDate, sampleMoney, aBoolean) values (?, ?, ?, ?)  
  
Hibernate: select this\_.id as id1\_6\_0, this\_.text as text2\_6\_0, this\_.aDate3\_6\_0\_, this\_.sampleMoney as sampleMo4\_6\_0\_, this\_....  
  
1 Simple Object  
2 Simple Object.  
  
He switches to the Command Prompt to view the table in the database. The first object has a Null value for the date column, and a zero amount for the sampleMoney column and no Boolean value.  
  
He switches back to Hibernate and opens the Generic.hbm.xml file and edits the class tag by adding a dynamic-insert="true" description after the name and table description.  
  
He switches back to the GenericMain.java file and clicks Run.*   
  
...when I look at the SQL statements, notice the first SQL statement, the one without a date, only passes over the three parameters that actually have a date. It doesn't change the net result over here on the database side, it still inserts the same thing, and this time with additional Ids, because we're automatically updating those, but it doesn't have to pass those fields apart. So if I have an object with a bunch of data, like a big blob or a clob or something like that, otherwise I wouldn't want to send across, I don't have to send that across. I can customize that. Same thing as we said, goes true for dynamic-update. And then there's other control options such as, if we come down here, a little bit further, select-before-update. And this basically ensures that I'm going to load up the object before I do an update, and really get the latest state; basically do a check on which things are really needed to be updated. So I don't...I might not have to do an update at all, my update might be an expensive gesture, it might be something that is just cheaper to go ahead and do that select, see if anything actually changed before overriding things, and locking tables, and committing transactions, and things like that. Now another choice inside of here you might see is polymorphism, and the polymorphism choice here gives us a bunch of different answers, implicit and explicit, in how it's going to join between tables.   
*The output is displayed in the Console:  
  
Hibernate: insert into generic\_auto (text, sampleMoney, aBoolean) values (?, ?, ?)  
  
Hibernate: insert into generic\_auto (text, aDate, sampleMoney, aBoolean) values (?, ?, ?, ?)  
  
Hibernate: select this\_.id as id1\_6\_0, this\_.text as text2\_6\_0, this\_.aDate3\_6\_0\_, this\_.sampleMoney as sampleMo4\_6\_0\_, this\_....  
  
1 Simple Object  
2 Simple Object  
3 Simple Object  
4 Simple Object.  
  
He switches to the database on the Command Prompt. A new table has been added, with four rows. The last two have an id of 3 and 4 respectively. They are duplications of the first two objects that were inserted.  
  
The presenter switches back to Hibernate and opens the Generic.hbm.xml file. He clicks in the class tag and the template proposals window opens. He scrolls through the options to select-before-update.  
  
He then scrolls to find the polymorphism option and selects it to add into the class tag. By default polymorphism is set to implicit. He opens the xml template proposals and there are two options: implicit and explicit.*   
  
We're not going to talk about polymorphism here entirely, but just to know, polymorphism almost always means a join between tables to load the objects. And so this will allow me to implicitly load, or explicitly load, to say whether that join happens automatically or whether it should be delayed or done later on. And so it's another great control on the fine-tuning of the performance and the desired behavior of the class. So this class actually has no polymorphism, I'm going to take this back out. There's so many different features here, such as, if you wanted to, you can change the persister model altogether. The persister is what Hibernate uses to save the class. You can change entirely how this class is saved by plugging in your own if you need special behavior, or you can do it with locking strategies here. You can see there's optimistic locking strategies, and some version there, and that's how it's going to lock the database to manage rows. One really cool little thing I wanted to show you real quick live is the where clause. So let's say, you have a class that has different rules associated with it. Like maybe, instead of actually deleting rows, you have a flag that marks them as deleted, or I only want to load rows that the user can see, and there's some database permissions comments inside of there. Well the where clause allows me to attach a query, so in our case, we're going to do something simple, we're going to say sampleMoney > 0.   
*The presenter deletes the polymorphism attribute from the table and open the template proposals window again. He points out other options such as persister and optimistic-locking. He selects the where clause and inserts it into the class tag.  
  
He adds a value of sampleMoney > zero to the clause.*   
  
I only care about, in this case, objects where this value is greater than zero, and so the net result of this, when I go off and save this, is when it loads up the objects, it only cares about objects that meet this criteria. If it doesn't meet this criteria, it's as if it did not exist in the database. So before, you can see our test here inserts a couple of objects, and then loads them all back through a query to load everything from the class. So now when I run the query, with the where clause inside of there, I inserted a couple of new objects, but I'm only getting objects 2, 4, and 6 back from my database. Now what happened on the database, nothing happened. I still have six objects out there, but three of them have zero dollar values. And so my greedy little algorithm here, my greedy little mapping, only cares about objects that have money. So again, a nice little feature for your class. So it's definitely worth digging into this class tag inside of your XML mapping and understanding all the great little features I can do to customize the exact behavior I want out of my Hibernate mapping.   
*The presenter switches back to the GenericMain.java file and examines the code:  
  
           {  
               Generic g = new Generic();  
               g.setId(1);  
               g.setText("Simple Object");  
               s.save(g);  
           }  
             
           {  
               Generic g = new Generic();  
               g.setId(2);  
               g.setText("Full Object");  
               g.setaBoolean(true);  
               g.setaDate(new Date());  
               g.setSampleMoney(199.99);  
               g.setCountNotPersisted(5);  
               s.save(g);  
           }  
  
           for (Object o : s.createCriteria(Generic.class).list())  
           {  
               Generic g = (Generic) o;  
               System.err.println(g.getId() + " " + g.getText());  
           }  
             
           s.getTransaction().commit();  
       } catch (Throwable t)  
       {  
           t.printStackTrace();  
           factory.getCurrentSession().getTransaction().rollback();  
       } finally  
       {  
  
He clicks the Run button and the output shows that three new objects have been created with the Ids of 2, 4, and 6.  
  
He switches to the table in the Command Prompt and there are now six rows in the table but only the objects with the id of 2, 4, and 6 have a sampleMoney value greater than zero.*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a02_it_enus/output/html/course_transcript.html#top)

Advanced XML Table Mapping

Learning Objective

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

* *use advanced XML table mapping in Java Hibernate*

**1. Utilizing advanced XML mapping**

On most of our mappings, in Hibernate, go from one table to one class. Occasionally our class model and our design and our database don't really align the way we want them to. And so there's times where we really would like to take the contents of a couple different tables, and have them be represented in a single class. So here we have an object that represents just a DoubleTable, nothing specific business wise, but it has a whole bunch of attributes that really makes sense for them to be together, as far as our domain model object goes. And so in XML mappings, we just create an object that represents all those attributes. And we have getters and setters associated with it, an empty constructor, all the requirements we'd normally have to be a required Hibernate object. When we get to our mapping then, we have our regular old class mappings, and we have to pick one table that's going to be the main table inside of there, it doesn't necessarily matter which table that is, but ideally it's the one that represents the primary amount of data, and obviously the primary key that's going to be part of that table, whichever ones drives the value of primary key. And then it's going to be up to how your database is designed. And so as we go through and do our mapping, we start with, you know, doing the id, and then the properties that are associated with that table.   
*Hibernate is open and three tabs are displayed in the document view: SecondaryMain.java, DoubleTableObject.java, and DoubleTableObject.hbm.xml.  
  
The presenter selects the DoubleTableObject.java tab to open the document. It contains the code:  
  
public class DoubleTableObject  
{  
   private long id;  
   private String text;  
   private Date aDate;  
   private double sampleMoney;  
   private boolean aBoolean;  
   private String name;  
   private String ssn;  
   private String zip;  
  
   public long getId()  
   {  
       return id;  
   }  
  
   public void setId(long id)  
   {  
       this.id = id;  
  
He points out the class name and the attributes that are added to the table. The presenter selects the DoubleTableObject.hbm.xml file. It contains the code:  
  
<hibernate-mapping package="hibernate.domain.table" >  
   <class name="DoubleTableObject" table="generic">  
       <id name="id"/>  
       <property name="text"/>  
       <property name="aDate"/>  
       <property name="sampleMoney"/>  
       <property name="aBoolean"/>  
         
       <join table="ComplexPerson">  
        <key column="remoteId"/>  
       <property name="name"/>  
       <property name="ssn"/>  
       <property name="zip"/>  
       </join>  
         
   </class>  
</hibernate-mapping>  
  
He points out the class tag that is being used for mapping. He then points out the Id and the properties that are associated with the generic table.*   
  
But we don't have to be limited to that, we can join into a separate table. So if we come, really quickly while we're here, and we can look at our database design, there is the primary table we looked at. You can see we have the Id, the text, the date, the money, and the Boolean, that all represent back to those primary values from the generic table. But we have a separate table out there, called ComplexPerson, and that table has a few additional values. The remoteId being the foreign key that ties these two together. So that, being the design, we can throw them all into a single class using the join tag inside of here. So within our XML mapping, we use join to point to a separate table, and the join then also needs a key column. And the key column represents that foreign key element out there that ties these two tables together. And then we just map the other properties as you see here, the name, the Social Security number, and zip, from this user, and they all end up in the single object. So notice, there's no sub object, there's no secondary class that has to be done there, it will all end up in one class when we're all said and done.   
*The presenter points out the join table tag in the code:  
  
<hibernate-mapping package="hibernate.domain.table" >  
   <class name="DoubleTableObject" table="generic">  
       <id name="id"/>  
       <property name="text"/>  
       <property name="aDate"/>  
       <property name="sampleMoney"/>  
       <property name="aBoolean"/>  
         
       <join table="ComplexPerson">  
        <key column="remoteId"/>  
       <property name="name"/>  
       <property name="ssn"/>  
       <property name="zip"/>  
       </join>  
         
   </class>  
</hibernate-mapping>  
  
He switches to the database view in the Command Prompt window and runs the command:  
  
describe generic  
  
The generic table is generated and populated with the associated attributes reflected in the XML.  
  
He runs the command:  
  
describe complexperson  
  
The complex person table is generated. It contains the attributes assigned to it in the XML. He points out that the remoteID value links in the complexperson table links to the id value in the generic table.*   
  
And so we have a little test here to show that, where we create this DoubleTableObject. We only have to set the Id one time, but then we set all of the properties straight up, as if they were just a single object, save that object, and commit that transaction. So we run our test here, and it's going to go off, and it's going to do two inserts here. So it starts with an insert into generic, as our primary table that has the keys set. And then it will go through and insert into our second class, our ComplexPerson, in there. And so, we can take a peek at that, and you can see now, our generic class has its data set, and then our remote class has its data set. And so this join mapping gives us a lot more flexibility in being able to tie our classes together, being able to tie our tables together, mapped in such a way that both models are happy. I'm not limited by my database design when I am creating my Java object model, nor do I have to compromise my Java object model towards a few specific things in the database.   
*The presenter selects the SecondarytMain.java tab and the document displays the code:  
  
public class SecondaryMain  
{  
   private SessionFactory factory;  
  
   public static void main(String[] args)  
   {  
       new SecondaryMain().run();  
   }  
     
   private void run()  
   {  
       try  
       {  
           factory = ConfigHelper.getSessionFactory();  
           Session session = factory.getCurrentSession();  
           session.beginTransaction();  
  
           DoubleTableObject o = new DoubleTableObject();  
           o.setId(1);  
           o.setaBoolean(true);  
           o.setaDate(new Date());  
           o.setName("Double me");  
           o.setSampleMoney(22.22);  
           o.setSsn("4321");  
           o.setText("Test Me double");  
           o.setZip("44444");  
             
           session.persist(o);  
             
           session.getTransaction().commit();  
       } catch (Throwable t)  
  
The presenter points out that the DoubleTableObject is set up with all of the properties in that single object. He points out the persist for the save, and clicks the Run button.  
  
The code runs and the following results are recorded in the stack trace in the Console view:  
  
Hibernate: insert into generic (text, aDate, sampleMoney, aBoolean, id) values (?, ?, ?, ?, ?)  
  
Hibernate: insert into ComplexPerson (name, ssn, zip, remoteId) values (?, ?, ?, ?).  
  
The presenter highlights that Hibernate perform two inserts, one set of values for the generic table and another set of value for the complex person table.  
  
He switches to the Command Prompt. Two separate tables have been created but the id value in the generic table matches the remoteId value of the complex person table.*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a02_it_enus/output/html/course_transcript.html#top)

Custom SQL in XML Mappings

Learning Objective

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

* *use custom SQL in XML mappings in Java Hibernate*

**1. Customizing SQL in XML mapping**

Hibernate has two highly desirable features. The first of which pretty much is universally accepted as being useful in mapping result sets back to Java objects. Everybody loves that, nobody wants to have to do that by hand. The other side is Hibernate creates for you, your insert, update, delete, and some basic queries. And not everybody loves that, some people like to control that on their own. So within your XML mapping inside of Hibernate, you can actually update the mapping to allow you to override and use your own version of those basic queries. So we have a basic class here, we're using custom statements for this story. So it's mapped just like a story would normally be, and since it's XML annotations, we don't have anything inside of our Java class, it's just a plain old Java class with its attributes. If we go to the Hibernate mapping then, we have just a normal mapping. We have a class, some properties, and all that sort of stuff. But we can add in these tags, the sql-insert, sql-update, and sql-delete here, tags that will allow us to replace what Hibernate would normally generate and execute, with something that we're going to put in of our own native SQL. Now it might be ANSI SQL, that will work on any database. It might even choose to be customized to point to Oracle, or DB2, or whatever database you're working with. And so inside of here, this insert statement, you notice I'm doing some fancy stuff.   
*Hibernate is open and the document view displays three open tabs: CustomSQL.java, StoryCustomStatements.hmb.xml, and StoryCustomStatements.java.  
  
The StoryCustomStatements.java tab is selected and the document displays the code:  
  
package hibernate.domain;  
  
import java.util.Date;  
  
public class StoryCustomStatements  
{  
   private Long   id;  
   private String title;  
   private Date   storyDate;  
   private Date   creationDate;  
   private Date   lastModifiedDate;  
  
   public Long getId()  
   {  
       return id;  
   }  
   public void setId(Long id)  
   {  
       this.id = id;  
   }  
   public String getTitle()  
   {  
       return title;  
   }  
   public void setTitle(String title)  
   {  
       this.title = title;  
   }  
   public Date getStoryDate()  
   {  
       return storyDate;  
   }  
   public void setStoryDate(Date storyDate)  
   {  
       this.storyDate = storyDate;  
   }  
   public Date getCreationDate()  
   {  
       return creationDate;  
   }  
   public void setCreationDate(Date creationDate)  
   {  
       this.creationDate = creationDate;  
   }  
   public Date getLastModifiedDate()  
   {  
       return lastModifiedDate;  
   }  
   public void setLastModifiedDate(Date lastModifiedDate)  
   {  
       this.lastModifiedDate = lastModifiedDate;  
   }  
}  
  
The presenter selects the StoryCustomStatements.hbm.xml tab.*   
  
So for instance, when I add a story to the database, perhaps I'm choosing that I don't want to take the timestamp of the Java system, I want to take the timestamp from the database system. And so there's a little bit of some goofy SQL inside of here to work around what's going on, because the Hibernate system requires that every mapped field gets sent over as part of the insert. And so what I'm doing inside of here is I'm taking the map value they're going to give me, I'm going to multiply it by zero, to make it zero, and then add in the current timestamp from the database, which essentially is the current timestamp from the database. This trick works really well in the XML mapping to be able to get that call to the database timestamp. It's a little bit ugly, but it works. On the update side, I'm not messing with that, I'm just going straight stored procedure. I've taken the five parameters that Hibernate, again, requires that get sent out. The five parameters are the ones we've mapped in the titles, it's based on your object and the five parameters we've mapped in this file. And that's going to allow me to create a stored procedure to do the same thing. And then delete just takes one procedure, which is the object, one attribute, excuse me, one parameter, which is the objects id that's getting passed over.   
*The StoryCustomStatement.hbm.xml document contains the code:  
  
<?xml version="1.0" encoding="utf-8"?>  
<!DOCTYPE hibernate-mapping PUBLIC   
"-//Hibernate/Hibernate Mapping DTD//EN"  
"http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">   
  
<hibernate-mapping package="hibernate.domain">  
   <class name="StoryCustomStatements" table="story">  
       <id name="id" column="id" type="long">  
           <generator class="native"/>  
       </id>  
       <property name="title"/>  
       <property name="storyDate"/>  
       <property name="creationDate"/>  
       <property name="lastModifiedDate"/>  
         
       <sql-insert>  
        insert into story (title, storyDate, creationDate, lastModifiedDate) values  
                         (?, ?, ? \* 0 + CURRENT\_TIMESTAMP, ? \* 0 + CURRENT\_TIMESTAMP)  
       </sql-insert>  
       <sql-update>  
        call specialStoryUpdate(?, ?, ?, ?, ?)  
       </sql-update>  
       <sql-delete>delete from story where id = ?</sql-delete>  
   </class>  
</hibernate-mapping>  
  
He points out the sql-insert, sql-update, and sql-delete tags that have been added to the xml mapping. He examines the sql-insert statement and the code:  
  
       <sql-insert>  
        insert into story (title, storyDate, creationDate, lastModifiedDate) values  
                         (?, ?, ? \* 0 + CURRENT\_TIMESTAMP, ? \* 0 + CURRENT\_TIMESTAMP)  
       </sql-insert>*   
  
So I can go and execute this, and the code to execute this is no different than if you were using the original map statements. You can see here I have a save call, I have down here an update call, and then finally I have a delete call, which then inserts a story, updates a story, deletes a story, and you can see here, when it's going to do it, it's going to the call the custom insert. That's actually a method I have, you know a print line I have right here in red, but you can see the Hibernate call printing out the SQL that's being put out there. It's calling my stored procedure that I've set up in the database. Stored Procedure can do anything, it doesn't matter what it's doing inside of there. The interesting thing I can do here is I can replace the everyday generated insert, update, or delete calls from Hibernate with my own custom statements. So the net result of this is I am not limited to what Hibernate provides for insert, update, or delete. Whether it's a special business rule, or satisfying a database administrator, or whatever my need is, I can tweak and customize Hibernate to do exactly what I needed from out of my system. And thus, it's even more powerful than it is with just the basic mappings and CRUD statements.   
*The presenter switches to the CustomSQL.java tab and clicks the Run button. The document displays the code:  
  
           StoryCustomStatements story = new StoryCustomStatements();  
           story.setTitle("Insert using custom statements");  
           story.setStoryDate(new Date());  
             
           Long id = (Long) s.save(story);  
  
           s.getTransaction().commit();  
           s = factory.openSession();  
           s.beginTransaction();  
             
           story = (StoryCustomStatements) s.get(StoryCustomStatements.class, id);  
           System.err.println(story.getTitle() + " " + story.getLastModifiedDate());  
           s.evict(story);  
  
           StoryCustomStatements updatedStory = new StoryCustomStatements();  
           updatedStory.setId(new Long(1));  
           updatedStory.setTitle("Update the story now");  
           updatedStory.setStoryDate(story.getStoryDate());  
             
           s.update(updatedStory);  
  
           s.getTransaction().commit();  
           s = factory.openSession();  
           s.beginTransaction();  
  
           story = (StoryCustomStatements) s.get(StoryCustomStatements.class, id);  
           System.err.println(story.getTitle() + " " + story.getLastModifiedDate());  
  
           s.delete(story);  
             
           s.getTransaction().commit();  
  
The Console displays the output:  
  
Hibernate: insert into story (title, storyDate, creationDate, lastModifiedDate) values (?, ?, ? \* 0 + CURRENT\_TIMESTAMP, ?...  
Hibernate: select storycusto0\_.id as id1\_8\_0\_, storycusto0\_.title as title2\_8\_0\_, storycusto0\_.storyDate as storyDat3\_8\_0\_,...  
  
Insert using custom statements 2014-02-24 09:53:43.0  
  
Hibernate: call specialStoryUpdate(?, ?, ?, ?, ?)  
Hibernate: select storycusto0\_.id as id1\_8\_0\_, storycusto0\_.title as title2\_8\_0\_, storycusto0\_.storyDate as storyDat3\_8\_0\_,...  
  
Update the story now 2014-02-24 09:53:43.0  
Hibernate: delete from story where id = ?.*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a02_it_enus/output/html/course_transcript.html#top)

Detailed Property Mapping in XML

Learning Objective

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

* *perform detailed property mapping in Java Hibernate by specifying controls in the property and column tags within XML*

**1. Using XML property mapping**

Out-of-the-box Hibernate is made to let you use a lot of conventions to easily create your classes and your mappings. So in our basic mapping of a table just made up here, the <property> tag automatically assumes the column names exactly match whatever the property names are, and vice versa. If I'm creating from scratch, this works great. If I'm in an existing database it might not work great. I don't necessarily want to have to use those cryptic database names in my Java; they seldom align well. So there's lots of options I can throw into these properties to go inside of there. So there's a different, basically control of each property, but it doesn't really get to the same level of granularity as if I go ahead and open this property up and add in a <column> tag. So you can see the <column> tag gets to the database, and the specific items that I want to go with in the database. Now for this column, it actually aligns to the same…excuse me, the same attribute aligns to the same name, it doesn't have to.   
*Hibernate is open and there are four tabs in the document view: Generic.java, Story.hbm.xml, Generic.hbm.xml, and GenericMainXML.java.  
  
The Generic.hbm.xml tab is selected and the document contains the code:  
  
<hibernate-mapping package="hibernate.domain">  
   <class name="Generic" table="generic\_auto">  
       <id name="id">  
           <generator class="native"/>  
       </id>  
  
       <property name="text"/>  
       <property name="aDate"/>  
       <property name="sampleMoney"/>  
       <property name="aBoolean"/>  
   </class>  
</hibernate-mapping>  
  
The presenter clicks in the sampleMoney property tab and opens the XML template proposals window.  
  
There are a number of options, these include: formula, index, and access. He closes the options without selecting any and inserts a new line break below the sampleMoney tag to insert a column tag.  
  
He adds the line of code:  
  
                 <column name="sampleMoney">  
                 </column>*   
  
In another class we have over here, we actually need to be able to define the column, because we need to know specifically what's going on. There is no storyid defined in any of these objects, "storyid" is a column that's in this case is a foreign key in the database, but I can specify it separately. But in our case, I can specify this whatever I want to. If this was named something different, it could be pointed at something differently. And so this doesn't affect it, one way or another, whether I have this mapping there or not, I go through and I run it and it shows up with inserting my objects in the database. The objects in the database, they all show up exactly as they would, otherwise the names all align as you can see here to our base objects. So we don't necessarily need this every single time, but when they don't align, it works well. But it does give us a lot of options even beyond this. So for the column, I can specify all sorts of information about how this column should be mapped. I can tell whether it's nullable or not, I can tell the SQL type that I'm expecting it to be. So for instance in this case, if I'm expecting it to be a double or a float, I can put that out there. This type that has to do with how Hibernate maps those things, it's not the actual database type, it's how Hibernate maps to those database types, so that's pretty easy. I can go through and pretty much put any precision values inside of there I want to do, or I can even specify the whole item. This becomes very useful if I'm generating the database from Hibernate and I want to have very precise controls over how that data's represented.   
*The presenter selects the Story.hbm.xml tab and examines the code:  
  
<hibernate-mapping package="hibernate.domain">  
   <class name="Story" table="story">  
       <id name="id" column="id" type="long">  
           <generator class="native"/>  
       </id>  
       <property name="title"/>  
       <property name="storyDate"/>  
       <property name="creationDate"/>  
       <property name="lastModifiedDate"/>  
  
       <set name="pages" table="page" cascade="all"  
               inverse="true" lazy="true" fetch="select">  
           <key>  
               <column name="storyid" not-null="true" />  
           </key>  
           <one-to-many class="hibernate.domain.Page" />  
       </set>  
   </class>  
</hibernate-mapping>  
  
He switches back to the Generic.hbm.xml code. He selects the GenericMainXML.java tab and clicks the Run button. The output in the Console shows that two simple objects have been created:  
  
1 Simple Object  
2 simple Object  
  
He switches to the Command Prompt to view the table in the database. The column names match the property names as set out in the Generic XML document.  
  
He switches back to the XML in Hibernate and clicks in the sampleMoney tag. The template proposal window opens. He selects sql-type and it is inserted into the tag. The presenter then deletes the sql-type.*   
  
Again if you have that much level of control, you might have an existing database, but if you so want to do that from Hibernate you can; there's a lot of really interesting options inside of there. So we're not limited to that though, inside of the <property> as we said, there's other options inside of here, such as whether or not I want to allow this table to be…this column to be inserted, or whether I want it to be updated, there's also that option down here. So for instance, let's just say for whatever reason, this money value is not something that this application should be inserting. Maybe it's a banking application, this is a client tool, and so I don't want this client to be able to add data...add information to their bank account. They want to be able to see it, but I can say insert="false" and maybe update="false". So when I change this over, and I change this mapping over, and I go to execute this and go run this at this point...   
*The presenter reopens the template options. The options include: length, node, scale, unique-key, and unique="false."  
  
He selects the insert option and adds it to the sampleMoney property tag. By default the value is set to true. He changes it to false. He also adds the update statement and sets it to false.*   
  
...we can see I am inserting the rows, I did two inserts inside of there, but it did not include as part of the insert statement the money value. And it then is updated, and pulled those rows, and when I look at those rows, now notice that the money value stays zero, even though that row four there has the exact same insert, and I should have inserted $199.99, I am not doing it because I've said, you cannot insert, you cannot update. And so I have full control, within reason, of the mapping between my data model and my database. So my object model and my database model can be aligned to the needs that they are out there. The database model is for the whole application, the object model is for this specific application, and I can treat it as such. And so investigate these features of the property…the <property> tag and the <column> tag, and you'll have a lot better understanding of the controls you can use in Hibernate.   
*The presenter switches to the GenericMainXML.java tab and clicks the Run button. The results are displayed in the Console view:  
  
Hibernate: insert into generic\_auto (text, aDate, aBoolean) values (?, ?, ?)  
Hibernate: insert into generic\_auto (text, aDate, aBoolean) values (?, ?, ?)  
  
Hibernate select this\_.id as id1\_6\_0\_, this\_.text as text2-6\_0\_, this\_.aDate as aDate3-6-0\_, this-.samplMoney as sampleMo4-6\_0-0,...  
  
1 Simple Object  
2 Simple Object  
3 Simple Object  
4 Simple Object  
  
He opens the Command Prompt to view the tables. The money value in the sampleMoney column for the new objects is now zero.*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a02_it_enus/output/html/course_transcript.html#top)

Embedded Objects with @EmbedDED or the component tag

Learning Objective

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

* *embed objects in Java Hibernate with the @Embeddable annotation or through XML with the component tag*

**1. Embedding objects**

For my sample application here, I have a database design that is not normalized in the way I prefer. I have a address that's embedded inside of another table, and that table has other fields I don't really care about, but I'd like to be able to treat that address as a Java object of its own. So you can see I can create a Java object class with a city, state, zip, and street address, and that I want to be able to use in my Java object model, as an object that can be passed around and included as many other objects. But again, the problem is it's not mapped that way in my database. So what I can do is I can make this object embeddable with the @Embeddable annotation, and we'll see how to do this in a second in XML as well. By doing this, I don't have to map anything else, this might be embedded in a number of different other classes, but it can be treated separately in Java. So then in a parent class such as RealEstate, I can use an @Embedded tag to point to my address inside of there. Now if everything aligns and all the attributes match the column names, that's all I need to do.   
*The Command Prompt contains a RealEstate table with five columns – Field, Type, Null. Key, Default, and Extra – and six rows. The row values in the Field column are id, street, city. state, zip, and value.  
  
The presenter switches to Hibernate. There are four tabs in the document view: Address.java, EmbeddedMain.java, RealEstate.java, and RealEstate.hmb.xml.  
  
The Address.java tab is selected and displays the code:  
  
@Embeddable  
public class Address implements Serializable  
{  
   private String street;  
   private String city;  
   private String state;  
   private String zip;  
  
The presenter points out the@Embeddable tag. He switches to the RealEstate.java tab and the following code is displayed:  
  
@Entity  
public class RealEstate  
{  
   @Id  
   private long id;  
   @Basic  
   private double value;  
  
   @Embedded  
   @AttributeOverrides({  
       @AttributeOverride(name="street", column=@Column(name="street")),  
       @AttributeOverride(name="city",   column=@Column(name="city")),  
       @AttributeOverride(name="state",  column=@Column(name="state")),  
       @AttributeOverride(name="zip",    column=@Column(name="zip")),  
   })  
   private Address address;  
  
He points out the @Embedded tag.*   
  
Here's an example though of if it doesn't match the column names the way I want it to be, if the attribute names and column names don't align, I can override that using this @AttributeOverrides tag. It uses a series of @AttributeOverrides to point to city, state, zip, and street as all the different items that I can override inside of there. The name here equates to the name inside of the Address class. The column here equates to the column name inside of our database, as we're seeing over here. So this override here allows me to treat this both as an object in the Java side and as a series of columns that was part of the entity RealEstate in the database side. So I can go through and run this as a sample, where I'm going to build a basic RealEstate object. I'm going to create an Address object on its own and set that inside of there and save this, and when I go through and run this, you'll see it inserts it into the database seamlessly, treating it as if it was just a single table. And so I've inserted the data directly into our table inside of here. Now as I said, this can also be done in XML mappings fairly seamlessly. In XML, I don't have the notion of an embeddable object, but instead I have a <component>.   
*The presenter switches back to the Address.java document and examines the code:  
  
@Embeddable  
public class Address implements Serializable  
{  
   private String street;  
   private String city;  
   private String state;  
   private String zip;  
  
He then points out that the column name in the code displayed relates to the values in the Field column in the database table.  
  
The presenter selects the EmbeddedMain.java document. It contains the code:  
  
public class EmbeddableMain  
{  
   private SessionFactory factory;  
  
   public static void main(String[] args)  
   {  
       new EmbeddableMain().run();  
   }  
     
   private void run()  
   {  
       try  
       {  
           factory = ConfigHelper.getSessionFactory();  
           Session session = factory.getCurrentSession();  
           session.beginTransaction();  
  
           RealEstate r = new RealEstate();  
           r.setId(1);  
           r.setValue(100000);  
           Address a = new Address();  
           a.setStreet("123 Main");  
           a.setCity("Chicago");  
           a.setState("IL");  
           a.setZip("12345");  
           r.setAddress(a);  
             
           session.save(r);  
  
He clicks the Run button and switches to the database in the Command Prompt window to view the table. The Address table has been created as a separate table containing the inserted data.  
  
He switches back to Hibernate and selects the RealEstate.hbm.xml tab.*   
  
And a <component> here defines some…and this maps to the exact same table by the way...the exact same class as well, and so some attributes inside of the class with getters and setters – we can ignore the annotations for the purposes of the XML, they wouldn't exist if we're doing it in XML – but I point to that address with a get address, set address of class="Address". In this case I can cheat and use the Address without the package, because the package is described up here. And then I map the four properties, a street, city, state, and zip. Again I do not have to describe the columns here, because in this case the database table aligns perfectly to the mapping of the attributes as the naming goes. So like we had in RealEstate here, I could've left out the attribute mapping in the XML mapping. I indeed did leave this out and I did not define the columns. So embeddable objects really give you a lot of flexibility towards being able to have your object model represented in a perfect, pristine, exactly as you want it form, even if the database can't be represented as such. And so use that to really glue together these two pieces without having to sacrifice on either side.   
*The RealEstate.hbm.xml contains the code:  
  
<hibernate-mapping package="hibernate.embed">  
   <class name="RealEstate" table="RealEstate">  
       <id name="id" column="id" type="long">  
           <generator class="native"/>  
       </id>  
       <property name="value"/>  
  
<component name="address" class="Address">  
       <property name="street"/>  
       <property name="city"/>  
       <property name="state"/>  
       <property name="zip"/>  
</component>  
   </class>  
</hibernate-mapping>*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a02_it_enus/output/html/course_transcript.html#top)

Basic Querying – HQL

Learning Objective

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

* *perform basic querying using HQL in Java Hibernate*

**1. Implementing queries in HQL**

Hibernate introduces the Hibernate Querying Language, HQL, which is very similar to SQL, but set in a Java context, and the objects that you are defining. Introducing the basics of how to create, execute, and work with these results is key before diving deeper into the details of HQL. Moving from managing database by hand into Hibernate, means that I'd like to be able to move away from the Structured Query Language, SQL, as the way I am primarily communicating with the database. One way I can do that is with criteria objects, but we're here to talk about HQL, the Hibernate Query Language, which allows us to move away from database-specific or even ANSI standard SQL into a Hibernate generated language that looks kind of like SQL, but allows us to talk in terms of objects, talk in terms of concepts in Hibernate, and then abstracts out from us the need to understand the underlying database to some degree. So a very basic HQL query could look as simple as this. I could go through for my session; I create a query using a string "from Story". Then notice I don't have to say SELECT, I don't have to say SELECT\*, I don't have to all that details out there. I just have a class out there called Story, capital S. I don't know what the database name is, it could be Story, capital S, it could be story, lower case S, it could be anything, but that allows me to query on that. So if I go and run my test real quick, you can see this will pull back all of the stories out the database. I have a little test script that puts 100 stories in the database, and there is all 100 of them coming back from the database.   
*Hibernate is open and there are two documents open in the document view: QureryingMain.java and StoryQuery.java.  
  
StoryQuery.java is selected and displays the code:  
  
   // Used in Basic Querying  
   public List<Story> loadAllQuery()  
   {  
       Query stories = session.createQuery("from Story");  
       List<Story> all = stories.list();  
       return all;  
   }  
     
   // Used in Basic Querying  
   public List<Story> queryForAllOnADay(Date day)  
   {  
       Query q = session.createQuery("from Story where storyDate = :date");  
       q.setReadOnly(true);  
       q.setDate("date", day);  
       return q.list();  
   }  
     
   // Used in Basic Querying  
   public List<Story> queryForAllOnATopic(String topic)  
   {  
       Query q = session.createQuery("from Story where title like :topic");  
       q.setReadOnly(true);  
       q.setString("topic", "%" + topic + "%");  
       return q.list();  
   }  
  
The presenter selects the QueryingMain tab which displays the code:  
  
// Basic Querying  
//            driver.run(TestDriver.QUERY\_ALL);  
//            driver.run(TestDriver.QUERY\_DATE);  
//            driver.run(TestDriver.QUERY\_TOPIC);  
  
He uncomments the first example and clicks Run. The following are some of the results that are displayed in the Console view:  
  
97 - 2014-02-15 - President fist bumps Eggplant  
98 - 2014-02-24 - Woman hugs Eggplant  
99 - 2014-02-23 - Man congratulates Doughnut  
100 - 2014-02-15 - Doughnut high fives Eggplant  
  
Hibernate: delete from Page  
Hibernate: delete from Story  
Hibernate: ALTER TABLE story AUTO\_INCREMENT =1  
Hibernate: ALTER TABLE Page AUTO\_INCREMENT =1.*   
  
Now obviously that's not enough. I don't query for everything in the database very often, so I need to add some criteria on there, I need to be more specific. So my next test here, you can see I can select everything from the Story class, where the date, the storyDate is something. And here's where HQL comes in as a huge advantage. In JDBC, dealing with SQL, I have to deal with parameters, and parameter orders, and question marks, and inserting those, and prepared statements, and all that fuss. Well in HQL, I created my query, my HQL query and I can actually name the parameters here. I use the colon and then whatever name I want to put after that, as long as it adheres to some basic rules of being one word, and you know, using basic text and things like that. And so beyond that, I can go through and I can set that date to be any Java date in there that is passed in for the user. And as a side effect I can actually, you know this is a choice, I can set that query to be read-only, which means I can be a little bit more efficient, not having to maintain that query or open to be able to update the database. So I can make this query, plug this in, and then just simply list the results that are coming back for me. And so if I go and run that second test here, let me set this up to go to the second test...   
*The presenter points out the next HQL querying example, where he uses the where clause to search by a specific date in the story database. He uses this HQL code:  
  
      Query q = session.createQuery("from Story where storyDate = :date");  
  
He also highlights that he can set the date to be any date passed in by the user using, as well as setting the query to be read only using the code:  
  
       q.setReadOnly(true);  
       q.setDate("date", day);  
  
He use this code to list the results:  
  
       return q.list();  
  
The presenter switches back to the QueryingMain.java document, comments the previous test and uncomments the current test.*   
  
...and run it, I can show you just the stories that occur on the date that was passed in. And so it's a significantly shorter list of stories. So the where clause pretty much operates the same way it does in SQL. Now we're not limited to just where clauses and equals, we can also use things like like, no pun intended. So the like clause in there allows us to query on a topic here. We're searching on the story title in this case, it's a text field, and I want to search for a word or phrase inside that text field. And so I can have somebody pass in some word or phrase to me, and then as an HQL query I have to put the parentheses...not the parentheses...the percentage signs in appropriately. So before and/or after, depending on how I want to search. So here I want to search anywhere in the title for the topic, and so that's what I'm setting up here. I have to do this by hand in HQL queries to be able to add that inside of there if I want to truly do wildcard searches. But again through and I can list the results of the query and it's going to return back to me another set of stories. So I can go through and run that test.   
*He clicks Run. The results of the queries are shown in the Console. This time only the queries that match the clause have been returned.  
  
The presenter focuses on the next querying example, the topic query, in the StoryQuery.java document. The code for doing a wild card topic query is:  
  
   // Used in Basic Querying  
   public List<Story> queryForAllOnATopic(String topic)  
   {  
       Query q = session.createQuery("from Story where title like :topic");  
       q.setReadOnly(true);  
       q.setString("topic", "%" + topic + "%");  
       return q.list();  
   }  
  
The presenter switches back to the QueryingMain.java document and clicks Run. The results are displayed in the Console window.*   
  
And so this test comes back, and it returns all the queries having to do with dogs. So whether the dog is doing something, or whether something's being done to the dog inside of here, it is a same query with a wildcard happening inside of there, returning back just those stories. So these are some basic HQL queries. Obviously SQL is much more involved than this and so is HQL, but really the fundamentals is covered right here. I list the name of the class that I'm searching on, I use attributes that point to class attributes. Again I don't know what the database columns are called, and then I can pass in Java variables using setString, setDate, setBoolean, whatever Java types are involved inside of there. So I'm insulated from the database. The basics of HQL is there to allow me to stay in a Java world, but still have a bridge that gives me a SQL-like structure in which to do querying.

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a02_it_enus/output/html/course_transcript.html#top)

Basic Querying – Criteria

Learning Objective

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

* *perform basic querying with simple Java method calls using the Criteria API in Hibernate*

**1. Querying using Criteria**

Hibernate provides two options for you to do querying; there's HQL, the Hibernate Query Language, and then there's Criteria objects. And so we want to dive into Criteria objects as they give you a very programmatic way within Java to build up a query using just simple Java method calls and adding on, and then from there generating your query into your SQL base, letting Hibernate do all the work, all the magic behind the scenes. So the most basic query you can do is getting everything out of a class, and in Criteria objects, that's really simple. I just simply go to my session and I say create a criteria for whichever class you wish to query into, and that's going to, by default, load everything out of that class. So we'll see how we can add on more to this here in a second, but we just simply can start with that criteria for all the stories and list it. And the list() method is basically saying, "hey go execute that and return back to me, not only a result set, but a list of mapped objects you get back from that." So if I run this query, very easy. Sorry, I've got to write it from my main application. I run this query quick and easy, it's going to go through, it's going to load the mapping, and then it's going to load all those objects. You can see it's getting all 100 objects out of my database here. Now as it does that query you can see it's selecting each of the items from my database.   
*Hibernate is open and there are two tabs in the document view: QueryingMain.java and StoryCriteria.java.  
  
The StoryCriteria tab is selected and the code is displayed:  
  
   // Used in Basic Querying  
   public List<Story> loadAllQuery()  
   {  
       Query stories = session.createQuery("from Story");  
       List<Story> all = stories.list();  
       return all;  
   }  
     
   // Used in Basic Querying  
   public List<Story> queryForAllOnADay(Date day)  
   {  
       Query q = session.createQuery("from Story where storyDate = :date");  
       q.setReadOnly(true);  
       q.setDate("date", day);  
       return q.list();  
   }  
  
The presenter discusses the first example and points out the line of code that creates a session for the class that is being queried:  
  
       Query stories = session.createQuery("from Story");  
  
The following is the code to return the list of results and mapped objects:  
  
       List<Story> all = stories.list();  
  
The presenter switches to the QueryingMain.java tab which displays the code:  
             
//            driver.run(TestDriver.CRITERIA\_ALL);  
//            driver.run(TestDriver.CRITERIA\_DATE);  
//            driver.run(TestDriver.CRITERIA\_TOPIC);  
  
He uncomments the first test and clicks the Run button. A list of all the objects in the class is returned and displayed in the Console.*   
  
So I don't have to be limited to just all the objects. I can get more specific objects inside of there as I choose. I can go through and in a second query, show you how I can query for all the stories on a particular date. So I have a method inside of here I've created, say hey, search for me all the stories on a date, you pass me in the date. And so in my Criteria object, I start with the same basic criteria, give me all the stuff on stories. Now I've chosen to make it read-only, that's an option. I just wanted to show you as an option inside of Criteria objects, if I'm not going to modify the results of this guys, I can make it read-only, which means I can basically not have to keep open a connection out to the database. And so for the Criteria objects, when I'm searching on dates, it's going to do a very exact search on hour, minutes, seconds as well, so I have to go through and clear out those to be zero, zero, zero, which is matching what's in my database for the story. And so past that, once I've got my date cleaned up, I'm going to add to that Criteria that I created on Story, a restriction. And this class here, Restrictions has all sorts of restrictions I can put inside of there. You can see there's equals, there are all equals, there's likes, there's empty's and not empty's, and greater than, and less than, but we just care about the equals, so when the date that got past in is equal to the "storyDate".   
*The presenter switches to the StoryCriteria tab and focuses on the next example. The code for querying by date is:  
  
   // Used in Basic Querying  
   public List<Story> allOnADay(Date day)  
   {  
       Criteria c = session.createCriteria(Story.class);  
       c.setReadOnly(true);  
  
       // Using the Criteria, we have to 'clean' the time or it will not match  
       // the database column without time  
       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.eq("storyDate", fromDate.getTime()));  
       return c.list();  
   }  
  
The last line of code is adding a restriction to the criteria. The presenter clicks on the code and opens the Template Proposals window to display the restriction options. He selects the equals restriction.*   
  
So I can go through and I can run this sample, and it's going to return back to me fewer stories. It's not going to return all the stories; it's going to return all the stories that happened on the 17th in this case. As you see, a much smaller set of stories that are being returned. Now the third test I'm doing inside of here is I can go through and search on just the topic. And so a third example here is instead of searching on date, I'm going to search on a Restriction where the "title" is like, where I have a MatchMode of ANYWHERE, a topic. So I don't have to worry about messing with parentheses, or I don't have to worry with percentage signs. I'm just going to say, the title should include this text, and the MatchMode could become before or after or…before or after, I could have it anywhere inside the text of whatever I pass in. So I if run this test then, I'm going to search all stories that are…not searching a date, but involve anything to do with a dog. So you can see, the dog is either before or after in this case, and it returned all the stories where a dog is involved. And so this is some basic querying, there's so much more we can do with Criteria objects, but this is a foundation for being able to do searching entirely in Java. I don't have to know SQL, I don't have to even know HQL, I just need to know how to assemble objects in Java.   
*The presenter switches to the QueryingMain.java tab and clicks Run for the topic criteria. The results that match the criteria and the equals restriction are returned and displayed in the Console view.  
  
He switches back to the StoryCriteria.java document and discusses the block of code:  
  
   // Used in Basic Querying  
   public List<Story> allOnATopic(String topic)  
   {  
       Criteria c = session.createCriteria(Story.class);  
       c.setReadOnly(true);  
       c.add(Restrictions.like("title", topic, MatchMode.ANYWHERE));  
       // In criteria, we don't have to worry about adding the % by hand, we  
       // can specify that in code unlike the HQL approach  
       return c.list();  
   }  
  
He points out that he has added a restriction to the search. He opens the template options for the MatchMode. They include END, EXACT, and START.  
  
He switches to the QueryingMain.java document and clicks Run\. The results that match the restriction and match mode are returned as a list in the Console view.*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a02_it_enus/output/html/course_transcript.html#top)

Advanced Query Management

Learning Objective

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

* *order and paginate result data using advanced query management options in HQL and Criteria objects within Java Hibernate*

**1. Ordering queries**

As I'm searching into the database doing some real deep searching, I normally am going to get to a point where the results I'm getting back are going to be well excessive for my ability to consume them in a single look. And so both from a human standpoint and from a database standpoint, I want to order and paginate those results in a way where I can consume them a little bit at a time. It doesn't do me any good to get a million rows back, I can only look at maybe 10 or 15 or 100 on a page at a time. So within our HQL, we can first of all add in the notion of order by, and so order by is the same way it works in SQL, where I can pick some field that I'm going to sort the result set by, and in this case, I'm sorting by the storyDate descending. I can do ascending or descending inside of here. And so I don't want to just look at any random story, I want to put them in an order basically order them by a date. Now past that, I'm going to set a match maxFetch size here. So I don't want to return back a million rows, I want to return back at most some size, and in this case we're going to use 10 as our example.   
*Hibernate is open and there are three tabs in document view: QueryingMain.java, StoryQuery.java, StoryCriteria.java.  
  
The presenter selects the StoryQuery tab and the code is displayed:  
  
   // Used in Advanced Query Management  
   public List<Story> PagedQuery(int startPage, int maxFetch)  
   {  
       Query c = session.createCriteria(Story.class);  
       q.addOrder(Order.desc("storyDate"));  
       q.setReadOnly(true);  
       q.setMaxResults(maxFetch);  
       q.setFirstResult(startPage);  
       return q.list();  
   }  
  
He points out the orderby criteria: StoryDate descending. He uses the this code to set the fetch size:  
  
       q.setMaxResults(maxFetch);*   
  
And when I'm searching through there then, I need to know which set of pages am I looking at. The first 10, the second 10, the third 10, the fourth 10, because I don't want the user to have to look at the 10 over and over again. If I get the first 10, it doesn't do me any good, unless I set the fetch to, well, get the second 10 if they hit the next button, and so on and so forth. You've seen that behavior I'm sure before. So let's go ahead and look at this in action. So as I go through, my test is actually going to go through and pull back the first three pages. My test creates 100 stories, we're ordering them by date, and then we can go pull back the first three pages. And so here you can see the first page pulls back 10 stories, page one inside of here. And as it's loaded it back, it did a query for the stories, and then because it's lazy loading involved it loaded the pages for each of those 10 stories. So here's the first 10 stories that I got. And then I went back and I went to page two, and so this method was called again and instead of passing in a one this time, it passed in a two. And again the maxFetch size was 10, so for page two, it went and queried, it loaded the pages as needed, and there is my second 10 that's coming in. Again you notice the date here, up here was 24, 23, now we're down to 23, 22, we're paging through the database, and then the third one down here 22, 21.   
*The presenter selects the QueryMain.java tab and clicks Run. The results of the query are displayed in the Console view.  
  
The results of the query are displayed in batches of ten and are organized according to page numbers.*   
  
So in HQL, I can use the order by to basically get the results back in a specific order that I know, I can predict, that I can come back to over and over again as I page it. And then in the query object that I've created from my HQL, I'm setting my max results and my fetch location basically to allow me to paginate my results. Now the vast majority of this is the same when I move into Criteria. So let me switch my test case over here. And I look at my Criteria solution here; I'm again going to do a search on the Story. I'm going to search all of stories here, and I add the order simply by adding in…calling a call it to addOrder. And so there's an Order helper outside of there, and I can…this order can be either ascending or descending, either way we want to, on a property name. And the property name here is going to be the "storyDate" – the same thing we were doing before. It's the exact same query, just built through HQL. Again we made it read-only, maxFetch size, match result size. The only difference here is I'm working on a Criteria object, the call are named the exact same thing.   
*The presenter selects the QueryingMain.java tab and uncomments the next test. He switches to the StoryCriteria.jave tab and the following code is displayed:  
  
   // Used in Advanced Query Management  
   public List<Story> allPaged(int startPage, int maxFetch)  
   {  
       Criteria c = session.createCriteria(Story.class);  
       c.addOrder(Order.desc("storyDate"));  
       c.setReadOnly(true);  
       c.setMaxResults(maxFetch);  
       c.setFirstResult(startPage);  
       return c.list();  
   }  
  
This is the code for the add order call with an order of descending, and on a property name of storyDate  
  
       c.addOrder(Order.desc("storyDate"));*   
  
I can still in Criteria set a maxFetch size and the starting page. So I can go through and run that sample and really the results are going be essentially the same. So again, I get page one out here. In this case the Criteria object is not loading the pages; it's basically not needing the pages, so it's a little bit simpler that way. It's not anything against HQL or Criteria, it's just the choice that's made and the way this is mapped. But I'm coming in and I'm getting the first 10 pages on the 24th, and in this case my random generator made a whole bunch of pages on the 24th. And then there's another 10 pages on the 24th, and then finally when we get into the third page, you can see we moved from the 24th into the 23rd. So we are ordering our pages, we're returning them back. Our paging structure can be done whether in HQL or in Criteria and our order structure can be done in either one of those as well. It's just the choice of which one we prefer to work with, but the capabilities are the same for this advanced feature of managing queries through pagination.   
*The presenter selects the QueryingMain.java tab and clicks Run. The results of the criteria query are displayed in the Console. They have been ordered into groups of ten results.*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a02_it_enus/output/html/course_transcript.html#top)

Query by Example

Learning Objective

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

* *structure a database query using a Query by Example strategy in a Criteria object within Hibernate*

**1. Using by Example**

Structuring a query to a database is not always an easy task, it's often very nontrivial. One branch of thought of making querying easier is the notion of Query by Example. And in the Query by Example, what I do is I create an example like you can see here, a Story() object, and fill that object and say, hey give me any results that match, sort of kind of like the Story() object. In Hibernate I can do that through criteria searches. And so I'm going to through and create a story and say, give me anything that matches like a cat. We'll ignore the stuff...comment out for a second. And so I'm going to create a criteria into the Story() class, like I normally would with Criteria objects, but now I'm going to create a Query by Example by using the Criteria object Example.create for the criteria that I just set up. And for the example I need to set the criteria by which I'm going to search. So I'm going to say search anywhere, you can match anywhere. And so anything in the title that has a cat, whether it comes early or after, it doesn't matter where in the text shows up. And furthermore, ignore the case. Notice I did lowercase cat where in all the examples we're going to see it's upper case Cat. So I can set that there and then I can add in to the criteria, the example outside of there. So again, I'm creating an example on the Story() object I created up above, setting the match criteria, and then to the actual Hibernate Criteria object I'm going to go do…add that Create by Example limitation to that search and then run the result. So let me just go ahead and run this, and you can see I get back everything that's a cat.   
*Hibernate is open and there are two documents in the document view: QueryingMain.java and StoryCriteria.java.  
  
The StoryCriteria.java tab is selected and contains the code:  
  
   // Used in Query By Example  
   public List<Story> byExample()  
   {  
       Story story = new Story();  
       story.setTitle("cat");  
         
//        Calendar cal = Calendar.getInstance();  
//        cal.set(Calendar.DATE, 22);  
//        story.setCreationDate(cal.getTime());  
  
       Criteria c = session.createCriteria(Story.class);  
        
       Example ex = Example.create(story);  
       ex.enableLike(MatchMode.ANYWHERE);  
       ex.ignoreCase();  
       c.add(ex);  
//        c.setResultTransformer(Criteria.DISTINCT\_ROOT\_ENTITY);     
       return c.list();  
   }  
  
He edits the code so that it becomes:  
  
   // Used in Query By Example  
   public List<Story> byExample()  
   {  
       Story story = new Story();  
       story.setTitle("cat");  
         
//        Calendar cal = Calendar.getInstance();  
//        cal.set(Calendar.DATE, 22);  
//        story.setCreationDate(cal.getTime());  
  
       Criteria c = session.createCriteria(Story.class);  
        
       Example ex = Example.create(story);  
       ex.enableLike(MatchMode.ANYWHERE);  
       ex.ignoreCase();  
       c.add(ex);       c.setResultTransformer(Criteria.DISTINCT\_ROOT\_ENTITY);     
       return c.list();  
   }  
  
He switches to the QueryingMain.java document. The following code is displayed:  
  
// Query By Example  
//            driver.run(TestDriver.BY\_EXAMPLE);  
  
He enables the test and clicks the Run button.*   
  
Now when I do this search it's returning back multiples because the story is tied to a page and the page returns one hit for each story that comes out there, and that's why I have three of them out there. Nothing wrong with that, that's just standard database work. And so I want to go through and add here a DISTINCT call, and I do that in Criteria objects by setting the results transformer and adding the DISTINCT call. And so I'm going to do that, and now let me run the exact query one more time, and now you'll see we get one unique Story() object for each one that's out there. I don't get multiples for the pages; I get a much more condensed set of stories that have to do with cats. Now I don't just have to search on a title, on clear text, I could also search on let's say a date. Let's say I want everything that happened the 22nd, I want all the stories that happened on the 22nd. So I've uncommented that, I'll go back and look at that here, while my test runs, but I'm creating a calendar, I'm creating a date, the 22nd, and then I'm setting the creation date of the story to anything that was created on the 22nd. So now you can see when I run that criteria by example, all the stories that were listed on the 22nd are the ones that are coming back. Now Hibernate's smarter than this, our sample is smarter than this. I can say, give me all the cat stories that happened on the 22nd, I can put both of them out there. That's the whole idea of Criteria by Example; you have this set of columns, if I give you examples for each columns, give me anything that matches my example. So I go back here, let's run this one more time...   
*The presenter runs the query by example test and the results are displayed in the Console. The query results include everything in the database that includes the word cat. There are three instances of the query results returned.  
  
The presenter switches back to the StoryCriteria.java tab and edits the section of the code:  
  
       Example ex = Example.create(story);  
       ex.enableLike(MatchMode.ANYWHERE);  
       ex.ignoreCase();  
       c.add(ex);  
//        c.setResultTransformer(Criteria.DISTINCT\_ROOT\_ENTITY);     
       return c.list();  
   }  
  
He uncomments the setResultTransformer function to add a distinct call to the query.  
  
He switches back to the QueryingMain.java document and clicks the Run button. This time no multiple objects of the query are returned in the Console view.  
  
The presenter edits the code:  
  
   public List<Story> byExample()  
   {  
       Story story = new Story();  
       story.setTitle("cat");  
         
//        Calendar cal = Calendar.getInstance();  
//        cal.set(Calendar.DATE, 22);  
//        story.setCreationDate(cal.getTime());  
  
       Criteria c = session.createCriteria(Story.class);  
        
       Example ex = Example.create(story);  
       ex.enableLike(MatchMode.ANYWHERE);  
       ex.ignoreCase();  
       c.add(ex);  
//        c.setResultTransformer(Criteria.DISTINCT\_ROOT\_ENTITY);     
       return c.list();  
   }  
  
He comments the title query and uncomments the date query code. The code is:  
  
       Calendar cal = Calendar.getInstance();  
       cal.set(Calendar.DATE, 22);  
       story.setCreationDate(cal.getTime());  
  
He creates a calendar, a date, and creation date. He can now run a query for all entries created on a specified date.  
  
He runs the query and the results are displayed in the Console. He then uncomments the title word query and runs the query for both criteria in the example.  
  
The Console shows that no results are returned from the database that match the query.*   
  
...and we can see, oh I got nothing back. So either there's a mistake, or more likely in my sample of 100 there was no cat stories on that date, yeah there you go. Now on the 22nd we have some cat stories. Now shocking to find out a day in the news without a cat story, but hey, it is the modern Internet out there. Alright so the criteria here allows you to Query by Examples which is a really simple and powerful way to be able to do searches. It's not without risks, I have to be aware of text matching that's going on there, and I'm really not paying much attention to performance here. If I'm going to put the criterias like this, it's not something I want to put out for millions of users. It could really cripple my database depending on how the search is done, particularly if I'm searching across joined tables or multiple tables. So I want to do this very judiciously, but for coming up with quick and easy searches, particularly for trusted power users, Criteria by Example is a great, great way of being able to conduct a search.

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a02_it_enus/output/html/course_transcript.html#top)

© 2016 Skillsoft Ireland Limited