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

**Course Transcript**

Getting Started with Java Hibernate

**Getting Started**

| [1. Course Introduction](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a01_it_enus/output/html/course_transcript.html#t2) |

| [2. Bootstrapping Hibernate – Command Line](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a01_it_enus/output/html/course_transcript.html#t6) |

| [3. Hibernate Architecture](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a01_it_enus/output/html/course_transcript.html#t10) |

| [4. Basic Tests – Insert](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a01_it_enus/output/html/course_transcript.html#t14) |

| [5. Basic Tests – Update Transient Data](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a01_it_enus/output/html/course_transcript.html#t18) |

| [6. Basic Tests – Update Persistent Data](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a01_it_enus/output/html/course_transcript.html#t22) |

| [7. Basic Tests – Delete](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a01_it_enus/output/html/course_transcript.html#t26) |

| [8. Basic Tests – Reading](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a01_it_enus/output/html/course_transcript.html#t30) |

| [9. Interceptors for Hibernate Actions](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a01_it_enus/output/html/course_transcript.html#t34) |

**Transactions, Batches, and Logging**

| [1. Running Hibernate under JPA](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a01_it_enus/output/html/course_transcript.html#t39) |

| [2. Managing Sessions and Transactions](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a01_it_enus/output/html/course_transcript.html#t43) |

| [3. Batch Processing](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a01_it_enus/output/html/course_transcript.html#t47) |

| [4. Hibernate and Logging](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a01_it_enus/output/html/course_transcript.html#t51) |

**Caching and Custom Types**

| [1. Caching Data](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a01_it_enus/output/html/course_transcript.html#t56) |

| [2. Unique Item Query](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a01_it_enus/output/html/course_transcript.html#t60) |

| [3. Defining Custom Types](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a01_it_enus/output/html/course_transcript.html#t64) |

| [4. Filtering Data Results](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a01_it_enus/output/html/course_transcript.html#t68) |

**Mapping Data**

| [1. Allowing Hibernate to Create Tables](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a01_it_enus/output/html/course_transcript.html#t73) |

| [2. Multiple Mappings of Tables](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a01_it_enus/output/html/course_transcript.html#t77) |

| [3. Basic Mapping – Annotations](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a01_it_enus/output/html/course_transcript.html#t81) |

| [4. Basic Mapping – XML](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a01_it_enus/output/html/course_transcript.html#t85) |

| [5. Associations and Foreign Keys – Annotations](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a01_it_enus/output/html/course_transcript.html#t89) |

| [6. Associations and Foreign Keys – XML](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a01_it_enus/output/html/course_transcript.html#t93) |

Course Introduction

Learning Objective

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

* *start the course*

**1. Introduction to the course**

Java Hibernate allows you to create, save, and update data objects in a supported database. You can update or reback these objects into a Java program. Tony Lowe is an experienced software engineer and instructor and in this course he will demonstrate the basics of reading and writing objects to your RDBMS. He will also explore transactions and batches, caching, and basic data mapping. Let's get started.

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a01_it_enus/output/html/course_transcript.html#top)

Bootstrapping Hibernate – Command Line

Learning Objective

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

* *set up JAR files, configure a SessionFactory connection to a database, and use a ConfigHelper class to load Hibernate into a command-line application*

**1. Bootstrapping within Hibernate**

Hibernate can be run in any Java program via the command line or as a Desktop app; and directly configured and loaded from your basic Java solution. The steps are simple overall, but understanding the conventions is key to a quick-and-easy start into Hibernate. Hibernate is very flexible in the type of applications it can be used in. And so let's look at Hibernate just starting up inside of a basic command-line application. There're a few key things that you need to do to get started. The first thing that you need to do is get your hibernate.jar files set up. I've created a library, because there's a lot of jar files that's needed for base Hibernate…version 4.3 actually has a folder in it that says required jar files, and this is all of the ones that are included. There's optional ones, but these are the core ones you need. So once you've got that inside your system, you need a hibernate.cfg file. By default, it shows up in the base directory of your class path. You can name it whatever you want to; you can place it wherever you want to but, by convention, it's named hibernate.cfg.xml – and this is a sample of what that file looks like.   
*Java Eclipse is open and the hibernate.cfg.xml tab is displayed in the editor. The code that displays is:  
  
<?xml version='1.0' encoding='utf-8'?>  
<!DOCTYPE hibernate-configuration PUBLIC  
"-//Hibernate/Hibernate Configuration DTD 3.0//EN"  
"http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">  
  
<hibernate-configuration>  
  
   <session-factory>  
<!-- Database connection settings -->  
<property name="connection.driver\_class">com.mysql.jdbc.Driver</property>  
<property name="connection.url">jdbc:mysql://localhost:3306/BigNews</property>  
<property name="connection.username">root</property>  
<property name="connection.password">test</property>  
  
<!-- JDBC connection pool (use the built-in) -->  
<property name="connection.pool\_size">1</property>  
  
<!-- SQL dialect -->  
<property name="dialect">org.hibernate.dialect.MySQL5InnoDBDialect</property>  
  
<!-- Enable Hibernate's automatic session context management -->  
<property name="current\_session\_context\_class">thread</property>  
  
<!-- Disable the second-level cache  -->  
<property name="cache.provider\_class">org.hibernate.cache.in  
ternal.NoCacheProvider</property>  
  
  
The presenter then navigates over to the packages explorer where the AnnotationsIntro file is expanded revealing the src folder which contains the files, hibernate, hibernate.cfg.xml, and log4j.properties. The libraries JRE System Library, Sl4J, and Hibernate are also available. The presenter expands the Hibernate library, and the following files are displayed: antlr-2.7.7.jar, dom4j-1.6.1.jar, hibernate-common, hibernate-core-4.3, hibernate-jpa-2.1-a, jandax-1.1.0.Final, javassist-3.18.1-GA, jboss-logging-3.1.3, jboss logging-anno, jboss-transaction, and mysql-connector.*   
  
That file starts off with defining your session-factory, and the key part of the session-factory is your connection to the database. This can be done directly in the session-factory. It can actually be done as an external to that, using JNDI or an extra object like that but, in our case, I'm pointing to the…in this case, mysql.jdbc.Driver, a localhost provider, and then username and password for my database. I can tweak that if I want to – based off the JDBC stuff such as the connection.pool\_size, but I then have configured my connection to the database and Hibernate will manage that connection for me. As it manages sessions, it'll create, and dispose of, and refresh, and all those things you need to do with the database sessions. Past that, Hibernate wants to know what database you're talking to. You can use just some generic dialect inside of there, but you're losing the advantages Hibernate can do to optimize, tune, and otherwise customize the generated SQL to use your database best. It actually becomes very limiting if you don't use the proper dialect out there, and you can have some mysterious errors that'll pop up from time to time.   
*The presenter focuses on the line of code:  
  
<property name="dialect">org.hibernate.dialect.MySQL5InnoDBDialect</pro*   
  
So in my case, I'm using this MySQL5InnoDBDialect, and this has been customized towards the version of mySQL I'm using to really, really be able to optimize the queries out there. If you're using a different database, that's fine; Oracle, or DB2, or Sybase or, you know, all the different vendors out there most likely have their own dialect to be able to customize to that. So outside of that...there's a few other debug options outside of there. You can talk about caching if you want to; that's a separate topic altogether – not required, or simply do you want to have a debug option such as showing the generated SQL. This will spit out into your log every bit of SQL that Hibernate's producing; it lets you go off and debug those sorts of things. If you don't want that in your log – either for security reasons or just simply to clean it up – you set that to be false, and go from there. So there's lots of different options you can set inside of here – we're by no means covering them all but, other things, such as: does Hibernate creates your tables and manage your tables for you or not, or the batch\_size is going to go out there.   
*The presenter then scrolls down to reveal the additional code:   
  
<!-- Echo all executed SQL to stdout -->  
       <property name="show\_sql">true</property>  
  
       <!-- Enables adding of comments to SQL when generated -->  
       <property name="hibernate.use\_sql\_comments">true</property>  
  
       <!-- Required to sync the Hibernate schema to your database, else it will break! -->  
<!--         <property name="hbm2ddl.auto">create</property> -->  
  
       <!-- Used to save occasionally when doing large inserts of data -->  
       <property name="hibernate.jdbc.batch\_size">10</property>  
  
<!-- Used to save occasionally when doing large inserts of data -->  
       <property name="hibernate.jdbc.batch\_size">10</property>  
  
       <mapping class="hibernate.domain.Story"/>  
       <mapping class="hibernate.domain.Page"/>*  
  
And then, finally, all the different mappings. You can map to classes, you can map to files – all of those need to be included as part of your bootstrap. So anything that's an annotated file needs to be known about here; anything that's a mapping file in XML needs to be known about here – and there's some examples of each of these in our sample file out there. So this config file again just sets up Hibernate; it doesn't actually load it – you need to consciously load Hibernate. The way I'm doing that is I'm choosing to create my own ConfigHelper class. Of course, I can put these lines of code anywhere, but it's nice to externalize that to a class and kind of encapsulate those details. So inside of my ConfigHelper class, what I want to do is…as an application, I really only want to deal with Hibernate's SessionFactory. This is what I go through to get sessions, and sessions is what I go through to get my queries, and save objects, and update objects, and things like that. So I need to start with the Configuration; I need to load in the configuration file we just looked at. Again if I use the convention, I have hibernate.cfg.xml and that's in the base of my class path – all I need to do is say new Configuration(). If I don't put it there, then there's different options by which I can adjust my Configuration to point to where that file's going to be located. And that configuration that I ask it to configure(); it loads itself up, it goes and checks things out, and goes from there.   
*The presenter then scrolls down further to reveal the code:   
  
   <mapping class="hibernate.domain.StoryCustomStatements"/>  
       <mapping class="hibernate.domain.StoryArchive"/>  
       <mapping class="hibernate.domain.StoryBrief"/>  
       <mapping class="hibernate.domain.StoryDescription"/>  
       <mapping class="hibernate.domain.PublicStory"/>  
       <mapping class="hibernate.domain.Newspaper"/>  
         
       <mapping class="hibernate.domain.Generic"/>  
       <mapping class="hibernate.domain.State"/>  
  
       <mapping resource="hibernate/domain/proc/State.hbm.xml"/>  
       <mapping resource="hibernate/domain/proc/PublicStory.hbm.xml"/>  
       <mapping resource="hibernate/domain/proc/named.hbm.xml"/>  
   </session-factory>  
</hibernate-configuration>  
  
The presenter clicks the CommandLineMain tab in the editor, and the following code displays in the editor:   
  
   private Read           read;  
   private Update         update;  
   private Delete         delete;  
  
   public static void main(String[] args)  
   {  
       new CommandLineMain().run();  
   }  
  
   public CommandLineMain()  
   {  
       factory = ConfigHelper.getSessionFactory();  
       read   = new Read(factory);  
       create = new Create(factory, read);  
       update = new Update(factory, create);  
       delete = new Delete(factory, create);  
   }  
  
   private void run()  
   {  
       try  
       {  
           factory.getCurrentSession().beginTransaction();  
  
//  Create Demo  
//            create.run(Create.PERSIST);  
//            create.run(Create.SAVE);  
//            create.run(Create.SAVE\_OR\_UPATE);  
//            create.run(Create.SAVE\_OR\_UPATE\_EXISTING);  
  
The presenter selects the ConfigHelper.java tab in the editor and the following code is displayed:   
  
package hibernate.util;  
  
import org.hibernate.SessionFactory;  
import org.hibernate.boot.registry.StandardServiceRegistryBuilder;  
import org.hibernate.cfg.Configuration;  
import org.hibernate.service.ServiceRegistry;  
  
public class ConfigHelper  
{  
   private static final SessionFactory sessionFactory = buildSessionFactory();  
  
   private static SessionFactory buildSessionFactory()  
   {  
try  
{  
   // Create the SessionFactory from hibernate.cfg.xml  
   Configuration configuration = new Configuration();  
   configuration.configure();  
   ServiceRegistry serviceRegistry = new StandardServiceRegistryBuilder().applySettings(configuration.getProperties()).build();   
   return configuration.buildSessionFactory(serviceRegistry);  
} catch (Throwable ex)  
{  
   // Make sure you log the exception, as it might be swallowed  
   System.err.println("Initial SessionFactory creation failed." + ex);  
   throw new ExceptionInInitializerError(ex);  
}  
   }  
  
   public static SessionFactory getSessionFactory()  
   {  
return sessionFactory;  
   }  
}*   
  
Once I have the Configuration() configured, I can go and getProperties() from that and ask that to build the serviceRegistry. This serviceRegistry is there to allow me to plug Hibernate into all sorts of different solutions. I can plug it into web servers, I can plug it into Spring, I can plug it into...whatever the target location I have going on. For us – we're a command-line application, so what I want to do from that serviceRegistry and that configuration is, I want to get a SessionFactory. So to build the SessionFactory, I need a serviceRegistry; I need to load that from the configuration, and return that to my application. And so my command-line application then builds that factory and from that factory, I can go through and get sessions and whatnot. So I'm going to show this in action; I'm going to run this in Debug mode, just so you can get a feel for this. It's going to run a little bit slow in Debug mode; it's not this slow in real time. But before I go through and configure anything, I have a breakpoint. And if I step into this code right here, it...you can see it's loading up some stuff, it's finding...actually, I stepped over the code, I didn't mean to say step into...it's finding up a bunch of files here – I'll actually come over here where I can see the log in a little-bit bigger form – it's finding the Hib cfg files, it's reading the mappings for all of the subclasses and all of the XML files I've put out there. It's connecting a connection pool to my database – you can see it's connected to that using the username and password; I see the password here is masked. And then it's set up the dialect; all the things we talked about – it's getting itself ready to go.   
*The presenter navigates back to the CommandLineMain tab. He clicks Debug and the Debug window is displayed. The presenter clicks the CommandLineMain <init> () line: 32 fle in the Debug window, and the Console displays a series of files. The presenter closes the debug window, and then expands the Console window and focuses on the output:  
  
11:09:25,030  INFO Configuration:2092 - HHH000040: Configuration resource: /hibernate.cfg.xml  
11:09:28,143  INFO Configuration:757  - HHH000221: Reading Mappings from resource: hibernate/domain/proc/State.hbm.xml  
11:09:29,603  INFO Configuration:757  - HHH000221: Reading Mappings from resource: hibernate/domain/proc/PublicStory.hbm.xml  
11:09:32,780  WARN DriverManagerConnectionProviderImpl:93 - HHH000402: Using Hibernate built-in connection pool (not for production*   
  
From there it's going to return the SessionFactory and run the application...oops, I hit the wrong thing here; let's go back to my Debug mode, and we'll just let that run from there – and it goes through and runs the application, and it's done. So we'll go back to this log. In this case, it went through, it inserted, it queried, it did all sorts of interesting things. The core here though, that I want you to take away from this, is there's a series of steps to get Hibernate up and running the first time. Most everything comes out of this hibernate.cfg file, but you want to isolate your application from this – you want to basically get to the point of just dealing with the SessionFactory. And from there, everything's pretty much downhill. Isolate this out; whatever application you have – actually, if you use Spring or use the server solution, you can even skip a lot of these steps. But it will certainly help you to keep your application simple by working through the hibernate.cfg file, and making your choices and your configurations there.   
*The presenter clicks Run and the Run As dialog box opens. It contains the options: Java Applet and Java Application. The presenter closes the window and clicks Debug. The debug window is displayed once again. The presenter clicks Run, closes Debug mode, and expands the Console. The presenter then clicks the hibernate.cfg.xml tab, and then the ConfigHelper.java tab.*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a01_it_enus/output/html/course_transcript.html#top)

Hibernate Architecture

Learning Objective

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

* *configure Java Hibernate to work with applications*

**1. Configuring Hibernate architecture**

To successfully work in Hibernate, we want to have a good understanding of how it fits into our application, and the choices for implementing Hibernate, and also the customizations we can use within Hibernate through configurations. And so Hibernate is designed to be highly flexible. It doesn't require any specific server solution or any other backup problem. It's pure Java, it's self-contained, and has a lot of different plug-in options to let you customize the implementation using the frameworks and other components you really need and it's not limited to specifically any other type of application. You can build the landline apps, web apps, services, best jobs, desktop, you name it, if you can call Java you can call Hibernate. I'm not sure mobile entirely in that sense, but pretty much anywhere else. Now Hibernate is based off of this JPA solution, actually Hibernate came first and JPA is based off of Hibernate, but these days JPA is an open standard that provides the core of the annotations that Hibernate will use. JPA is not an entire solution, it needs tools like Hibernate in order to be able to do all of the work, but Hibernate realizes JPA as the front-end and then has a lot of specifics that you can realize and go beyond with the basic JPA standard includes.   
*Heading: Hibernate vs JPA.  
  
JPA provides most of the interfaces and annotation definitions although it is not an implementation of an entire solution. It can however be used as a point of replacement if any single implementation is not working.   
  
Hibernate is a JPA realization. It uses JPA for a lot of the front end, but it extends it as well.*   
  
So at the high level, Hibernate fits between your app in the database. Hibernate is going to manage the database, it's going to do all the interactions with the database and our job is going to be to configure Hibernate to do that work. We do that through a Hibernate config file that talks about the database connections and how are we going to plug-in the rest of our applications and setup like that, and then we customize our application using either XML mappings or annotated classes. And Hibernate loads those and figures out how to map our data to the database and back appropriately. And so we're going to use those APIs from Hibernate in order to make our database actions happen. So the detailed architecture within Hibernate gives us a bunch of configuration options. We can choose different JDBC providers to different databases. If we want to do transactions differently we can use JTA transactions, or Hibernate transactions, or other providers. There's many caching solutions inside of there and Hibernate even allows us to plug-in to JMX, to do monitoring of the performance of Hibernate. And this is an even an exhaustive list of, there are so many options we can use to customize the behavior and performance characteristics of Hibernate.   
*Heading: High Level Architecture.  
  
Hibernate fits between your app and database. It is configured and customized within your code and files. Hibernate provides the bridge to the database. Your app will focus on the APIs based on your mappings and objects.   
  
A diagram illustrates how Hibernate interacts with an app's database. At the bottom is the Database above which is Hibernate. Above Hibernate is the Hibernate.cfg.xk, XML Mappings, and Annotated Classes, and then finally your app.  
  
Heading: Detailed Architecture.  
  
A diagram displays Hibernate as containing JDBC, JTA, CACHE, and JMX.   
  
Hibernate has numerous configurable parts that can be used in whole or in part, including, Connections, Transactions, Caching, Querying, and JMX - Messaging.*  
  
So as we go and build a Hibernate application, Hibernate is a core framework that we are going to realize. We're going to load it up, we're going to start it up. But Hibernate from there will load our configuration, it's going to get our mappings, our caching, our security, all that sort of stuff, our config is based off of, as we can see here, our XML mappings and our domain classes. And so some of the domain classes have annotations, some of them might not depending on how we wish to map that. But then the other key thing is Hibernate plugs itself in the logging, and so we can get full details about the execution and path of Hibernate from its plug-in to and in this case, simple log4j is its choice, but we can configure that tool to use whatever tool we want, log4j or other solutions, and configure the logging to be as detailed or formatted however we want to through traditional logging configurations.   
*Heading: Building with Hibernate.  
  
A process diagram shows how you can build using Hibernate. Hibernate leads into Database connections, mappings, caching and security. This in turn leads into Java Domain Classes with annotations, and XML Mappings. XML Mappings then leads into Java Domain Classes. The Java Controller leads into Hibernate, Java Domain Classes with annotations, and Java Domain Classes. Hibernate also leads into Logging and the Logging Config.*   
  
So as we said, the mapping options, there's two of them out there. We can do XML files, which externalize the mappings of Hibernate from our Java model, and we tie our classes to our table. This is the way it was originally built before annotations existed within the Java solution. So it does keep everything separate from Java, a separate file and allows a certain flexibility in how we're going to model this and tie these things together. Now annotations is great because it ties it straight into the model. When I have a very straightforward application annotations keep me from having to build a secondary sort of files and allows me to see right there within my Java solution how it's mapped to the database. And so it's all in one spot and it's really the preferred method for most development outside of the special need to do very customized or overlaying mappings. So when we get to querying, Hibernate provides several solutions there as well. Hibernate gives you the Create, Read, Update, Delete, for reading all or reading one of those out of the box. We don't have to do anything special for that, there's methods to allow us to insert an object, update an object, delete an object, get it by its primary key.   
*Heading: Mapping Options.  
  
The two mapping options available are, XML, and Annotations.   
  
With XML, mapping is completed in a Hibernate defined XML format. Classes and attributes can be tied to tables and columns; this was the original method of mapping. XML mapping is separate from Java, and must maintain a separate file.   
  
Annotations uses mapping within the Java object and define the tie to the tables. Each attribute is mapped to a column in line. Annotations can be easier to manage as it is all in one location.  
  
Heading: Querying Options.  
  
Hibernate provides basic options when it comes to querying including, Create, Update, Delete, Read One, and Read All. You are able to construct queries in Java by building objects. Hibernate Query language lets you construct queries in a combination of SQL and Java data items.   
  
SQL provides a place that you are able to fall back on. SQL enables you to access your stored procedures.*   
  
On top of that though I can do whatever querying I want to, however interesting it might be, using either criteria objects which is an entirely Java based solution for doing querying where I can basically decorate a object tree and come up with interesting sequel queries at the end or what's called HQL, the Hibernate Query Language which is a combination of SQL syntax plus Java objects that I can use. So I can do my SQL querying basically using the syntax of my Java object. And if all that fails, I can still use SQL. Hibernate allows us to plug-in a custom SQL whether our DBAs or other database experts come up with interesting queries, or even use stored procedures or any technique that we would use normally in SQL whether it's NCSQL or customized to our database. So this is a high level view of Hibernate which hopefully you can see, gives you all the flexibility you need to create whatever type of applications you want and connect to the database however you need.

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a01_it_enus/output/html/course_transcript.html#top)

Basic Tests – Insert

Learning Objective

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

* *create, save, and update Java Hibernate data objects using the Save, Persist, and saveOrUpdate options*

**1. Configuring Java Hibernate data objects**

The Hibernate magic comes in its really great ability to do Create, Read, Update, and Delete (CRUD) operations. And so we're going to look at the Create operations here. There's a couple different ways I can create objects. The first is a call to PERSIST and so, quite simply, I create a Story object in this case – any object that's mapped into Hibernate – and I call PERSIST on that. And in this case, it's going to not only map the object to the database, but all of the cascaded objects – all of those objects associated with this object there. So if I go and **Run** my PERSIST test case inside of here, you can see I go through, I load up the mapping, I create an object, and I insert that to the Story, and then insert each of the pages associated with that Story into the database. I can load that Story back out to take a peek at it, and you can see – in the database, I have my Story and then the two pages that get associated with that. And so the PERSIST call, very simply, just goes off and saves it. Now the biggest thing about the PERSIST call, is it attempts to return as soon as possible. It goes off and it does the insert and it returns really quickly. Where, if my object has some sort of id generation that's happening on the database side, it will not stick around to check out what the ID is on a PERSIST call. It's basically saying: I don't care what the ID that gets generated is, I'm not going to use this object later anyway – throw it away, run quickly.   
*Java Eclipse is open on the Create.java tab. The code that is visible in the editor is:  
  
     
   public void callPersist(Story story)  
   {  
// Persist may or may not get the ID at this moment depending on how the Database works...  
session.persist(story);  
// Thus at this point, the persistent object may not have an ID set, which is quicker, but not if you need the ID  
   }  
  
                  
   public Long callSave(Story story)  
   {  
// Save will ensure the ID is returned when it is generated on the DB side, and return the object  
Long id = (Long) session.save(story);  
return id;  
   }  
  
         
   public void callSaveOrUpdate(Story story)  
   {  
// No return ID on this one  
session.saveOrUpdate(story);  
   }  
  
The presenter then clicks the CommandlineMain.java tab and the following code is available in the Editor:   
  
private void run()  
   {  
       try  
       {  
           factory.getCurrentSession().beginTransaction();  
  
//  Create Demo  
//            create.run(Create.PERSIST);  
//            create.run(Create.SAVE);  
//            create.run(Create.SAVE\_OR\_UPATE);  
//            create.run(Create.SAVE\_OR\_UPATE\_EXISTING);  
  
                                   
//  Read Demo  
//            read.run(Read.GET\_BY\_ID);  
//            read.run(Read.GET\_MISSING);  
           read.run(Read.LOAD\_BY\_ID);  
//            read.run(Read.LOAD\_MISSING);  
//            read.run(Read.LOAD\_TO\_EXISTING);  
//            read.run(Read.REFRESH);  
//            read.run(Read.SHOW\_LAZY\_LOADING);  
  
           
//  Update Transient Demo              
//          update.run(Update.UPDATE\_FROM\_SCRATCH);  
//          update.run(Update.UPDATE\_WITHOUT\_ALL);  
                        
//  Update Persistent Demo              
//          update.run(Update.LOAD\_AND\_UPDATE);  
//          update.run(Update.LOAD\_AND\_CALL\_UPDATE);  
//          update.run(Update.LOAD\_AND\_MERGE);  
           
  
//   Delete Demo  
//            delete.run(Delete.DELETE\_A\_PAGE);  
//            delete.run(Delete.DELETE\_PARENT\_ONLY);  
//            delete.run(Delete.DELETE\_TRANSIENT);  
//            delete.run(Delete.DELETE\_PERSIT  
  
  
The presenter then clicks Run. The output is displayed in the Console. The presenter focuses on the output:   
  
Hibernate: /\* inserthibernate.domain.Story \*/ insert ...  
Hibernate: /\* inserthibernate.domain.Page \*/ insert...   
Hibernate: /\* inserthibernate.domain.Page \*/ insert...   
  
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!  
Read and print the story from the database  
Hibernate: select story0\_.id as id1\_3\_0\_, story0\_.create...  
  
  
Welcome to Hibernate! - 2014-02-25 00:00:00.0  
  
Page 1   
  
I am the first page  
  
Page 2   
  
I am the second page  
  
The presenter selects the Create.java tab.*   
  
If I care about the ID, I need to move along. Instead of PERSIST, I need to go to the SAVE operation. So, the SAVE operation – I'm going to go ahead and **Run** that – it's just a similar, single, one-line call, but it returns back the ID as a result. So it means that I might hang up on the database for a second or two – depending on the nature of the database, depending on the nature of the ID being generated – until I get the ID back. It's just basically adding that one little extra step to my transaction, to my call, to my sequence – to ensure that the ID is available to the rest of the flow; and after I've made the call to SAVE. So you can see here in this case again, I'm inserting the Story; I'm inserting each of the pages inside of there, and then I'm reading it back out and I get the page back up – essentially no difference between PERSIST and SAVE outside of the ID being returned. The third option is to callSaveOrUpdate. SaveOrUpdate allows Hibernate to be clever and say, "Hey, if this hasn't been inserted in the database yet, insert it; if it's already in the database, just update the fields inside of there." So it's basically you saying as a developer: I don't care if I'm overwriting something or not; just make sure it's in the database. And so I can **Run** that test case – quite clearly – and in this first test case, there's nothing in the database that matches it – so let me go through and insert this into the database.   
*The presenter scrolls down to reveal additional code:   
  
       public void callSaveOrUpdate(Story story)  
   {  
       // No return ID on this one  
       session.saveOrUpdate(story);  
   }  
  
     
   public void callSaveOrUpdateExisting(Story story)  
   {  
       callSave(story);       // Save it as the ID of 1 (since we are starting from scratch)  
       session.evict(story);  // Dump the object out of Hibernate  
  
       Story second = sg.buildTransientCopy(story);  
       second.setTitle("Changed Title, Same ID");  
       session.saveOrUpdate(second);  
   }  
}  
  
He then navigates to the CommandLineMain.java tab and amends the code line:  
  
//  create.run(Create.SAVE\_OR\_UPATE);  
  
He removes the //. The code is now:  
  
create.run(Create.SAVE\_OR\_UPATE);.  
  
The presenter clicks Run, and the output is displayed in the Console.*   
  
So I'm calling saveOrUpdate, and you can see an insert's happening – and I'm going through and inserting each of those from there. All right, so that's easy-peezy, nice and simple; pull it back from the database. SAVE\_OR\_UPDATE pretty much acts like SAVE in this case. Let me just show you, real quick, what happens if that actually already exists in the database. So here's another test case; I'm going to **Run** it, and we'll go over and look at the code while it's running. Where I am calling SAVE on it, I'm evicting that from this session. The evict is an important call that says, "Hey, I don't care about it anymore; stop tracking this session, take it out of any cache that's going on there." And then I'm going to go through, build a copy of that Story, and then change the Title. So it's the same ID; just I'm changing the Title inside of there. So you can see up top, it's inserting it in the database; that's what's happening beforehand. And now I'm going to do the save/update.   
*The presenter focuses on the output:   
  
Hibernate: /\* inserthibernate.domain.Story \*/ insert ...  
Hibernate: /\* inserthibernate.domain.Page \*/ insert...   
Hibernate: /\* inserthibernate.domain.Page \*/ insert...   
  
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!  
Read and print the story from the database  
Hibernate: select story0\_.id as id1\_3\_0\_, story0\_.create...  
  
  
Welcome to Hibernate! - 2014-02-25 00:00:00.0  
  
Page 1   
  
I am the first page  
  
Page 2   
  
I am the second page  
  
The presenter then amends the codeline:  
  
// create.run(Create.SAVE\_OR\_UPATE\_EXISTING); by removing the //. The code is now:  
  
create.run(Create.SAVE\_OR\_UPATE\_EXISTING);  
  
The presenter clicks Run and the output is displayed in the Console. He focuses on the section of output:  
  
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!  
Read and print the story from the database  
  
Changed Title, Same ID - null  
  
Page 1   
  
I am the first page  
  
Page 2   
  
I am the second page  
  
creation: null modified: null  
Test Complete  
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!  
Hibernate: /\* update hibernatedomain.Story \*/ update....  
Hibernate: /\* update hibernatedomain.Page \*/ update....  
Hibernate: /\* update hibernatedomain.Page \*/ update....  
Hibernate: delete from Page  
Hibernate: delete from Story*   
  
Okay so when I do the save/update here, you can see I'm updating the Story instead. So I'm updating the Story to match those values instead. So I'm not loading it a second time afterwards; I'm just showing it the first time, but...I'm sorry...no, I am actually loading it the second time – I'm not showing it two times, I'm just showing it the second time, but I'm loading up that Story with the Title that's different. I'm changing the Title and I'm updating that in the database; and you can see the calls to update here instead, since the ID already exists. So those are kind of the three options for doing creation of stories; PERSIST, SAVE, and then saveOrUpdate – as long as it doesn't exist. And so use those as your rules and business because it makes sense, and then you can insert data all you like in a single call from a persistence object.

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a01_it_enus/output/html/course_transcript.html#top)

Basic Tests – Update Transient Data

Learning Objective

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

* *process and update transient data from outside the current flow of Hibernate CRUD operations*

**1. Updating transient data**

As you wish to update data in Hibernate – as part of the Create, Read, Update, and Delete (CRUD) functionality – there's really two ways to update the data and it's based off the state of that data. There's a transient state and a persistent state of data and, this time, we're going to talk about transient state. So what we mean by transient is saying, our object does not currently live inside of any active session. So we need to basically understand how our object is going to get loaded into the session and update it appropriately, because Hibernate works from within the session. And so the update of data in a transient state happens from an object that is basically taken from other...some third-party source. Maybe you're getting it through the Web, maybe you're getting it through the file that you're loading in – but you're building that object separately. So in the first test we're going to look at here, I'm going to be updating an object that I just built right now. So I build the object, but I'm going to tie it to an ID that exists in my database; I'm going to match it to a story I've already existed, and I'm going to update the Title of that story.   
*Java Eclipse is open on the CommandLineMain.java tab. The code that displays in the editor is:  
  
//  Create Demo  
//            create.run(Create.PERSIST);  
//            create.run(Create.SAVE);  
//            create.run(Create.SAVE\_OR\_UPATE);  
//            create.run(Create.SAVE\_OR\_UPATE\_EXISTING);  
             
             
//  Read Demo  
//            read.run(Read.GET\_BY\_ID);  
//            read.run(Read.GET\_MISSING);  
           read.run(Read.LOAD\_BY\_ID);  
//            read.run(Read.LOAD\_MISSING);  
//            read.run(Read.LOAD\_TO\_EXISTING);  
//            read.run(Read.REFRESH);  
//            read.run(Read.SHOW\_LAZY\_LOADING);  
             
  
//  Update Transient Demo              
//          update.run(Update.UPDATE\_FROM\_SCRATCH);  
//          update.run(Update.UPDATE\_WITHOUT\_ALL);  
  
                         
//  Update Persistent Demo              
//          update.run(Update.LOAD\_AND\_UPDATE);  
//          update.run(Update.LOAD\_AND\_CALL\_UPDATE);  
//          update.run(Update.LOAD\_AND\_MERGE);  
           
  
The presenter scrolls to reveal the code:  
  
   package hibernate;  
  
  
import org.hibernate.Session;  
import org.hibernate.SessionFactory;  
  
public class CommandLineMain  
{  
   private SessionFactory factory;  
     
   private Create         create;  
   private Read           read;  
   private Update         update;  
   private Delete         delete;  
  
   public static void main(String[] args)  
   {  
       new CommandLineMain().run();  
   }  
  
   public CommandLineMain()  
   {  
       factory = ConfigHelper.getSessionFactory();  
       read   = new Read(factory);  
       create = new Create(factory, read);  
       update = new Update(factory, create);  
       delete = new Delete(factory, create);  
  
He then opens the Update.java tab by clicking the update line of code. The Update.java code displays as:  
  
package hibernate.crud;  
  
import hibernate.domain.Page;  
import hibernate.domain.Story;  
import hibernate.test.StoryGenerator;  
import hibernate.test.StoryToString;  
  
import org.hibernate.Session;  
import org.hibernate.SessionFactory;  
  
public class Update  
{  
   private SessionFactory factory;  
   private Session session;  
   private Create create;  
   private StoryGenerator sg = new StoryGenerator();  
  
   public static final int UPDATE\_FROM\_SCRATCH  = 0;  
   public static final int UPDATE\_WITHOUT\_ALL   = 1;  
   public static final int LOAD\_AND\_UPDATE      = 2;  
   public static final int LOAD\_AND\_CALL\_UPDATE = 3;  
   public static final int LOAD\_AND\_MERGE       = 4;  
  
   public Update(SessionFactory factory, Create create)  
   {  
       this.factory = factory;  
       this.session = factory.getCurrentSession();  
       this.create  = create;  
   }  
  
   public void run(int testCase)  
   {  
       System.err.println("\n!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!");  
       System.err.println("The story to be updated, just added");  
       Story story = sg.buildStory();  
       create.callPersist(story);  
       StoryToString.printStory(story);  
  
       // Commit the add of the story  
       session.getTransaction().commit();  
         
       // Start a new session for the delete  
       session = factory.openSession();  
       session.getTransaction().begin();  
       session.evict(story);  
  
 System.err.println("\n!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!");  
       switch (testCase)  
       {  
           case UPDATE\_FROM\_SCRATCH:  
               System.err.println("Objects can be updated without being in session at all");  
               updateFromScratch();  
           break;  
           case UPDATE\_WITHOUT\_ALL:  
               System.err.println("You do not need a full object tree to update");  
               updateWithoutAll();  
           break;  
           case LOAD\_AND\_MERGE:  
               System.err.println("Calling  merge will update with any existing object in session");  
               loadAndMerge();  
           break;  
           case LOAD\_AND\_CALL\_UPDATE:  
               System.err.println("This will fail in a second, as it is already loaded, no need to call update");  
               loadAndCallUpdate();  
           break;  
           case LOAD\_AND\_UPDATE:  
               System.err.println("This updates automatically, as it is in session");  
               loadAndUpdate();  
           break;  
           default:  
               System.err.println("Invalid test case selected");  
           return;  
       }  
  
       // Commit the delete  
       session.getTransaction().commit();  
  
       // Open one more session to do the read  
       session = factory.openSession();  
       factory.getCurrentSession().getTransaction().begin();  
       // need to create this here to use the new Session we created above  
       Read read = new Read(factory);  
       System.err.println("\n!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!");  
       System.err.println("Show the story just updated");  
       StoryToString.printStory(read.getByID(1));  
  
  }  
     
   // Part of Basic Tests - Update Transient Data  
   public void updateFromScratch()  
   {  
       Story story = sg.buildStory();  
       story.setTitle("Updated Story Title!");  
       story.setId(1);  // We must match the ID to the story created first, this is to show it can be done with any object  
       int id = 1;  
       for (Page p : story.getPages())  
       {  
           p.setId(id++); // Matching the id to the first one created  
           p.setWords(p.getWords() + " updated");  
       }  
         
       session.update(story);*   
  
I'm also, in this case, going to create a full object tree. I'm going to update all the pages; I'm going to give each page an ID that matches the page that was created beforehand, and I'm going to update the words inside there to stay updated in each one of these pages. So let me go ahead and **Run** this test, and you can see that I can do it as long as everything exists in a transient state. So here you can see I've inserted a Story upfront: Welcome to Hibernate!, and then the first page and the second page. And then I'm updating it without having to load those into the session at all. So I'm updating each of these queries inside of there, and I'm getting a net result of the title's been updated and each of the pages have been updated inside of there as well. And they're stated in the database as being updated. It's nice, because that means I do not have to do a select every time before I do an update. I can actually, in Hibernate, enforce that and choose that to be an option but I don't have to do that from the performance standpoint; I don't want to have to make two visits to the database every single time. Now the question you might have there is: Let's say I just want to update part of it; I want to update the name of a story – do I really have to load up and configure every one of the pages? I may not even know at that moment how many pages a story has, much less what the content of them are. So I can actually update without creating the whole tree out there as well. So I can build a story and then update the Title and, in this case, I'm setting the pages to null.   
*The presenter navigates back to the CommandLineMain tab and clicks Run.  
  
The Console displays the output. The presenter focuses on the following section of output:   
  
Hibernate: /\* insert hibernate.domain.Story \*/ insert into Story  
Hibernate: /\* insert hibernate.domain.Page \*/ insert into Story  
Hibernate: /\* insert hibernate.domain.Page \*/ insert into Story  
  
Welcome to Hibernate! - Tue Feb 25 11:54:15 PST 2014  
  
Page 1   
  
I am the first page  
  
Page 2   
  
I am the second page  
  
creation Tue Feb 25 11:54:15 PST 2014 modified: Tue Feb 25 11:54:15  
  
The presenter navigate to the CommandLine.java tab and scrolls down to reveal the code:  
  
//  Create Demo  
//            create.run(Create.PERSIST);  
//            create.run(Create.SAVE);  
//            create.run(Create.SAVE\_OR\_UPATE);  
//            create.run(Create.SAVE\_OR\_UPATE\_EXISTING);  
     
             
//  Read Demo  
//            read.run(Read.GET\_BY\_ID);  
//            read.run(Read.GET\_MISSING);  
           read.run(Read.LOAD\_BY\_ID);  
//            read.run(Read.LOAD\_MISSING);  
//            read.run(Read.LOAD\_TO\_EXISTING);  
//            read.run(Read.REFRESH);  
//            read.run(Read.SHOW\_LAZY\_LOADING);  
  
             
//  Update Transient Demo              
//          update.run(Update.UPDATE\_FROM\_SCRATCH);  
//          update.run(Update.UPDATE\_WITHOUT\_ALL);  
  
             
//  Update Persistent Demo              
//          update.run(Update.LOAD\_AND\_UPDATE);  
//          update.run(Update.LOAD\_AND\_CALL\_UPDATE);  
//          update.run(Update.LOAD\_AND\_MERGE);  
           
  
The presenter selects the Update.java tab.*   
  
The concern might be: well, maybe Hibernate will delete all my pages; maybe it'll make them all go away. Well I'm going to **Run** this test case and, hopefully, show you that, fortunately, Hibernate is not that draconian – it does not go off and just update everything for me that way. So again, I create the Title here; I have the date up here, and the first page and the second page is created, and then I go through and I'm going to update at this point. So it's going to update Story, and notice as it's doing the update Story, it's not affecting the pages at all. I'm going to load that story back up and, again I see the updated Title, but the pages are all there – they're unchanged, they're unaffected. If I want to delete those pages, I can't just simply wipe them out from the object. I have to explicitly delete them. I have to go through and call a delete call to make those pages go away, and that's a topic for another time. But just know that the basic update call doesn't make either one of those happen. And the basic update call is just simply that; I can just simply do an update. I'll point out…you can do a saveOrUpdate if you wish, but if you do a saveOrUpdate you're not guaranteed to be updating something that isn't already in the database. It could save something new, and that's not necessarily desirable. So for...particularly with transient objects that you are assuming exist in the database, it's better to use update than saveOrUpdate as part of the call – unless you don't care.   
*The presenter selects the CommandLineMain.java tab, and clicks Run. The Console displays the output. The presenter focuses on the following output:  
  
The story to be updated, just added  
Hibernate: /\* insert hibernate.domain.Story \*/ insert into Story  
Hibernate: /\* insert hibernate.domain.Page \*/ insert into Story  
Hibernate: /\* insert hibernate.domain.Page \*/ insert into Story  
  
Welcome to Hibernate! - Tue Feb 25 11:54:15 PST 2014  
  
Page 1   
  
I am the first page  
  
Page 2   
  
I am the second page  
  
creation Tue Feb 25 11:54:15 PST 2014 modified: Tue Feb 25 11:54:15  
  
You do not need a full object tree to update  
Hibernate: /\* update hibernate.domain.Story \*/ update Story set create  
  
Show the story just updated  
Hibernate: select story0\_id as idl\_3\_0\_, story0\_.creationDate as  
Hibernate: select page0\_.storyId as story Id7\_3\_0\_, page0\_.id as   
  
updated Story Tile! - 2014-02-25 00:00:00.0  
  
Page 1  
  
I am the first page  
  
Page 2   
  
I am the second page   
  
The presenter navigates back to the Update.java tab.*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a01_it_enus/output/html/course_transcript.html#top)

Basic Tests – Update Persistent Data

Learning Objective

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

* *use the update and merge methods in Java Hibernate*

**1. Updating and merging persistent data**

Within our Create, Read, Update, and Delete options, Update has two choices, I can update data from a transient state, meaning it has nothing to do with the session. Or as we can talk about in this case, I can update data from a persistent state. Persistent data inside of Hibernate means it's been loaded into the session. And so in our first example here I want to show you, we can do an update without ever even actually calling update explicitly. I have a story which I'm loading based off of the ID, knowing the ID, its ID is equal to one. I'm going to take that story I'm going to update its title. Furthermore I'm going to dive into its pages, I'm going to just pick the first page inside of there and I'm going to update the text on the first page. Notice I'm not calling update, I'm not touching the session at all past loading the object. Hibernate recognizes you're making this object "dirty," that's what they refer to it as, "dirty," and when I go and commit that transaction it will save that automatically for us. So let's go ahead and **Run** that test case here. And we can see, inside of the flow, I'm adding the page, making it nice and clean and doing the insert, everything else like that. And then I'm going to go and load the page separately. So I select that page and story and the page back out. Notice it only grabs the first page in this case because that's all I'm using from the lazy loading standpoint and then I'm going to update that page.   
*Java Eclipse is open on the Update.java tab. The code that is visible in the editor is:  
  
// Part of Basic Tests - Update Persistent Data  
   public void loadAndUpdate()  
   {  
       Story story = (Story) session.load(Story.class, new Long(1));  
  
       story.setTitle("Load and update");  
         
       story.getPages().get(0).setWords("I updated the first page");  
  
       // Notice no update here!  
   }  
  
   // Part of Basic Tests - Update Persistent Data  
   public void loadAndCallUpdate()  
   {  
       Story story1 = (Story) session.load(Story.class, new Long(1));  
       story1.setTitle("Change the title of the loaded object");  
  
       Story story = sg.buildStory();  
       story.setPages(null);  
       story.setId(1);  
       story.setTitle("Load and Call update on the story");  
         
       session.update(story);  
  
       // Notice no update here!  
         
   }  
  
The presenter navigates to the CommandLineMain.java tab. The code that is visible in the editor is:  
  
//  Update Transient Demo              
//          update.run(Update.UPDATE\_FROM\_SCRATCH);  
//          update.run(Update.UPDATE\_WITHOUT\_ALL);  
           
             
//  Update Persistent Demo              
//          update.run(Update.LOAD\_AND\_UPDATE);  
//                       //update.run(Update.LOAD\_AND\_CALL\_UPDATE);  
//          update.run(Update.LOAD\_AND\_MERGE);  
           
             
//   Delete Demo  
//            delete.run(Delete.DELETE\_A\_PAGE);  
//            delete.run(Delete.DELETE\_PARENT\_ONLY);  
//            delete.run(Delete.DELETE\_TRANSIENT);  
//            delete.run(Delete.DELETE\_PERSITENT);  
             
             
           System.err.println("Test Complete");  
           System.err.println("!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!");  
           TestHelper.cleanup(factory.getCurrentSession());  
           factory.getCurrentSession().getTransaction().commit();  
       } catch (Throwable t)  
       {  
           t.printStackTrace();  
  
The presenter clicks Run. The output is displayed in the Console. The presenter focuses on the following section of output:   
  
Hibernate: select story0\_.id as id as ad1\_3\_0\_, story)\_.creationDate as  
Hibernate: select page0\_.storyId as storyId7\_3\_0\_, page0\_.id as  
Hibernate: /\* update hibernate.domain.Story \*/ update Story set  
Hibernate: /\* update hibernate.domain.Page \*/ update Page set  
Hibernate: select story0\_.id as id1\_3\_0\_, Story)\_.creationDate as   
  
Show the story just updated  
  
Page 1   
  
I am the first page  
  
Page 2   
  
I am the second page*   
  
Again I haven't told to update anywhere, but I have updated the story, I have updated the first page, and it's made those update calls behind the scenes for me when the session was committed. So if I'm going to go make the sacrifice of loading the class anyway, I don't have to go through and make any extra calls, I can just simply make the changes as it goes. Now let me show you if you accidentally. or as you're coding, you probably find this pretty quick. If I go through and I explicitly call update, so I load this story, and I change some stuff around the title, you know I change some stuff and the pages, and stuff like that. And I try and call update on that. So in this case, I have a story that's been loaded and I'm actually building a second story that I'm working on. So this is another classic area you'll run into the Hibernate. As I **Run** my test, it comes up and I get this NonUniqueObjectException. So you can see it's saying a different object with the same ID is already in this session. So if I look at my source code a little closer here, I've loaded a story inside of here.   
*The presenter removes the // from the // update.run(Update.LOAD\_AND\_CALL\_UPDATE); code line. He then clicks the Update.java class, and then clicks the CommandLineMain.java tab again. The presenter clicks Run, and the output is displayed in the Console. The presenter focuses in the output:  
  
A different object with the same identifier value was already associated with the session: [hibernate.do...*   
  
I'm actually starting to work with another story, I could be working with the same story but I'm working with another story in this case. And when I go to update the story it's saying, hey you can't do that, because I can only have one persistent object I'm working on at a time, even though the second one is transient I can't get away with that being out there and working with that object. So the question is how in the world do I know that some other process is using that in session? I mean how do I sort that out? Well you can be very assiduous and track it, and there's an evict function that you can use to get rid of objects and session. Or when it comes times to update, you can use the function called merge. So I have the exact same code up there while load something into the session and actually building a second story off of here and I'm going to go through and I'm going to merge these stories. Now notice something here. In the second example I'm changing the words on the page from the first story and at the same time I'm changing the title on the second story. And so when I go through and merge these guides it's going to be an interesting collision of what's happening inside of there. And depending on your viewpoint, Hibernate probably handles this pretty gracefully. So if we run this test, we can see I've inserted the story, and it's all set.   
*The presenter selects the CommandLineMain tab and removes the // from the update.run(Update.LOAD\_AND\_MERGE); code line. He clicks Run, and the output is displayed in the Console.*   
  
And then I'm going to update it and call merge on the existing object. So you can see the first merge updates the story name so the story is now called Merge the stories. And the second one is going to update the page name that says the first page has been updated on the merge. So the merge call is going to take all these different changes and net in a result object that's at the end. So the question is that what you want? This is kind of an interesting scenario because if I'm dealing with the web application, its unlikely I want an object to stick around that long, I don't want to keep a session open waiting for the user to come back to me later on, there could be seconds or minutes later, because that's a long time in web speak. But perhaps our batch application might open an object and work with it for a while. Often batch applications has multiple lines that affect lines in the file that would affect the same object. So I load a user, I apply this, I apply that, I make this change, I delete that. If I keep that in session, I can make all those changes progressively as I process through each step. So this is an interesting feature that mostly you want to be aware of and use appropriately because you can use this either as a persistent update, as we're saying here, or as transient and take advantage of the nature of the data and the efficiencies gained by understanding that and coding appropriately.   
*The presenter focuses on the output:   
  
Calling merge will update with any existing object in session  
Hibernate: select story0\_.id as id as ad1\_3\_0\_, story)\_.creationDate as  
Hibernate: select page0\_.storyId as storyId7\_3\_0\_, page0\_.id as  
Hibernate: /\* update hibernate.domain.Story \*/ update Story set  
Hibernate: /\* update hibernate.domain.Page \*/ update Page set  
Hibernate: select story0\_.id as id1\_3\_0\_, Story)\_.creationDate as   
  
Show the story just updated   
Hibernate: select story0\_.id as id as ad1\_3\_0\_, story)\_.creationDate as  
Hibernate: select page0\_.storyId as storyId7\_3\_0\_, page0\_.id as  
  
Merge the stories - 2014-02-25 00:00:00.0  
  
Page 1   
  
I updated the first page then will merge  
  
Page 2   
  
I am the second page*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a01_it_enus/output/html/course_transcript.html#top)

Basic Tests – Delete

Learning Objective

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

* *delete different types of data in Java Hibernate*

**1. Deleting data types in Hibernate**

The simplest of the Create, Read, and Update, and Delete functions, the CRUD functions, is probably delete, because mostly we just need to know the ID, the primary key of things to be deleted. If I know that, I can wipe it out pretty quickly, but as simple as it looks, you got to use it correctly in order for it to work right within Hibernate. There are some mapping connotations, some other stuff we're going to see here in a moment. So the first thing we're going to look at here is in our hierarchy, we have stories and pages. Pages are a dependent object on stories and so let's go through and delete a page from the stories. We've a test that can do that, we're going to run our first test here and make that happen. So in this, I'm going to create a story, this story is going to have a number of pages and I'm going to delete the first page. It's not a very safe way to delete the page, I'm just deleting page ID is equal to one, coming inside of there which could have other ramifications I mean this pages have a page numbering order associated with them. So all of a sudden page 2 and 3 are now page 1 and 2, but nobody's updated them. But that's not my responsibility here, I'm just trying to delete the page. So you can see it deletes the page just fine. When I load it back up, it does load it just fine but it starts on page 2. Not exactly perfect but hey, it works.   
*Java Eclipse is open on the Delete.java tab. The code that displays in the editor is:  
  
    public void deleteAPage(long id)  
   {  
       Page page = new Page();  
       page.setId(id);  
       session.delete(page);  
   }  
     
   public void deleteAllFromParentOnly()  
   {  
       Story story = new Story();  
       story.setId(1);  
       session.delete(story);  
   }  
     
  public void deleteAllFromTransient()  
   {  
       Story story = sg.buildStory();  
       story.setId(1);  
       int id = 1;  
       for (Page p : story.getPages())  
       {  
           p.setId(id++);  
       }  
       session.delete(story);  
   }  
  
The presenter selects the CommandLineMain.java tab and the code is displayed:  
  
package hibernate;  
  
import org.hibernate.Session;  
import org.hibernate.SessionFactory;  
  
public class CommandLineMain  
{  
   private SessionFactory factory;  
     
   private Create         create;  
   private Read           read;  
   private Update         update;  
   private Delete         delete;  
  
   public static void main(String[] args)  
   {  
       new CommandLineMain().run();  
   }  
  
   public CommandLineMain()  
   {  
       factory = ConfigHelper.getSessionFactory();  
       read   = new Read(factory);  
       create = new Create(factory, read);  
       update = new Update(factory, create);  
       delete = new Delete(factory, create);  
   }  
  
   private void run()  
   {  
       try  
       {  
           factory.getCurrentSession().beginTransaction();  
  
The presenter scrolls down to reveal additional code:   
  
/  Create Demo  
//            create.run(Create.PERSIST);  
//            create.run(Create.SAVE);  
//            create.run(Create.SAVE\_OR\_UPATE);  
//            create.run(Create.SAVE\_OR\_UPATE\_EXISTING);  
  
              
//  Read Demo  
//            read.run(Read.GET\_BY\_ID);  
//            read.run(Read.GET\_MISSING);  
           read.run(Read.LOAD\_BY\_ID);  
//            read.run(Read.LOAD\_MISSING);  
//            read.run(Read.LOAD\_TO\_EXISTING);  
//            read.run(Read.REFRESH);  
//            read.run(Read.SHOW\_LAZY\_LOADING);  
  
       
//  Update Transient Demo              
//          update.run(Update.UPDATE\_FROM\_SCRATCH);  
//          update.run(Update.UPDATE\_WITHOUT\_ALL);  
  
                         
//  Update Persistent Demo              
//          update.run(Update.LOAD\_AND\_UPDATE);  
//          update.run(Update.LOAD\_AND\_CALL\_UPDATE);  
//          update.run(Update.LOAD\_AND\_MERGE);  
  
             
//   Delete Demo  
//            delete.run(Delete.DELETE\_A\_PAGE);  
//            delete.run(Delete.DELETE\_PARENT\_ONLY);  
//            delete.run(Delete.DELETE\_TRANSIENT);  
//            delete.run(Delete.DELETE\_PERSITENT);  
             
             
           System.err.println("Test Complete");  
           System.err.println("!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!");  
           TestHelper.cleanup(factory.getCurrentSession());  
           factory.getCurrentSession().getTransaction().commit();  
       } catch (Throwable t)  
  
The presenter clicks Run and the output is displayed in the Console. The presenter focuses on the section of output:   
  
Delete one page from a story (not a safe way, but brute force)  
Hibernate: /\* delete hibernate.domain.Page \*/ delete from page where  
  
The story after delete call   
Hibernate: select story0\_.id as id1\_3\_0\_, story0\_.creationDate as  
Hibernate: select pages0\_.storyId as storyId7\_3\_0\_, pages0\_.id as id1...  
  
Welcome to Hibernate!   
  
Page 2  
  
I am the second page  
  
creation: 2014-02-25 modified: 2014-02-25 12:09:15.0   
Test Complete  
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!  
Hibernate: delete from Page  
Hibernate: delete from Story  
Hibernate: ALTER TABLE Story AUTO\_INCREMENT = 1  
Hibernate: ALTER TABLE Page AUTO\_INCREMENT = 1*   
  
Now it doesn't work the other way around, if I try and delete the story without addressing the pages and I don't have it mapped properly, I end up with a little bit of some trouble. So here's a little test that's going to try and delete the parent alone. Oh, big mistake, so again the error Cannot delete or update parent row: foreign key fails, so that the foreign key from the page to the story, that's going to deny me from deleting that story without deleting all the pages first. So by default, the delete does not fix that. If I want to fix that I have to go look at the mappings and look at cascading the delete function inside the mappings to get that to work properly. When I look at a mapping so we are going to keep going. The Delete All then if I want to do that, one of the ways I can do that is creating a transient set of objects that match exactly the network of objects that's out there from the database. So in this case I have a story I'm going to build and I have all the pages that I'm going to be having an IDs that match what's out there. I have to know the number of pages and know their IDs and everything like that. So I can go through each one of these guys, setup the hierarchy right and then delete the story because in the first case I just had a story with an ID. You know I can't just pass an ID to delete; I have to pass an object, but now it will cascade properly when I have my transient set of classes outside of there. So again I can show you this, go off and run this test.   
*The presenter selects the Delete.java tab. He navigates back to the CommandLineMain tab and removes the // from the delete.run(Delete.DELETE\_PARENT\_ONLY); code line and clicks Run. The output is displayed in the Console. The presenter focuses on the section of output:   
  
Cannot delete or update a parent row: a foreign key constraint fails ('bignews'.'pae', CONSTRAINT 'page\_ibfk\_1' FOREIGN KEY ('storyId') REFERENCES 'story'.  
  
The presenter selects the Delete.jave tab and scrolls down to reveal the code:  
    
public void deleteAllFromPersistent(long id)  
   {  
       Story story = (Story) session.load(Story.class, new Long(id));  
       session.delete(story);  
   }  
     
      
}  
  
The presenter selects the CommandLineMain.java tab. He clicks Run and the output is displayed in the Console.*   
  
It actually is cascading, it just needs to now know to cascade those associated with it. It doesn't automatically do a query to cascade where a story ID is like or anything like that, it needs to know the specific IDs and see, it deletes each page, then it deletes the story. Now even better than this, than running on a transient state, what's nicer is to go through and load that object. If I have the ID and I'm going to go to the trouble of deleting any way, that's kind of a big expensive process to begin with. I might as well go load it and then delete it. And that gives me a little bit more assurance that I'm not missing any pages; I'm not orphaning any pages, I know what that object tree looks like in its entirety. So let me go ahead and run this sample. And so here quite simply I load the page and you can see I...up here I'm inserting the page as part of my test case and then I'm loading the page, I'm giving the story and then I'm giving the pages associated with that. And then I go through exactly as we did in transient state and delete each page and then I delete the story. I look afterwards, story is gone, it's not there anymore. So this is a little bit safer way of deleting than the transient object.   
*The presenter focuses on the output:   
  
Hibernate: /\* delete hibernate.domain.Page \*/ delete from Page where  
Hibernate: /\* delete hibernate.domain.Page \*/ delete from Page where  
Hibernate: /\* delete hibernate.domain.Story \*/ delete from Story where  
  
The story after the delete call  
Hibernate: select story0\_.id as id1\_3\_0\_, story0\_.creationDate as  
  
No story loaded  
Test Complete  
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!  
Hibernate: delete from Page  
Hibernate: delete from Story  
Hibernate: ALTER TABLE Story AUTO\_INCREMENT = 1  
Hibernate: ALTER TABLE Page AUTO\_INCREMENT = 1  
  
The presenter selects the Delete.java tab. He then returns to the CommandLineMain tab and removes the // from the delete.run(Delete.DELETE\_PERSISTANT); and then clicks run. The output is displayed in the Console. The presenter selects the Delete.java tab.  
  
He focuses on the output:  
  
Delete when the object is already persistent  
  
Hibernate: select story0\_.id as id1\_3\_0\_, story0\_.creationDate as  
Hibernate: select pages0\_.storyId as storyId7\_3\_0\_, pages0\_.id as id1...  
Hibernate: /\* delete hibernate.domain.Page \*/ delete from Page where  
Hibernate: /\* delete hibernate.domain.Page \*/ delete from Page where  
Hibernate: /\* delete hibernate.domain.Story \*/ delete from Story where  
  
The story after the delete call  
Hibernate: select story0\_.id as id1\_3\_0\_, story0\_.creationDate as  
  
No story loaded  
Test Complete  
Hibernate: delete from Page  
Hibernate: delete from Story  
Hibernate: ALTER TABLE Story AUTO\_INCREMENT = 1  
Hibernate: ALTER TABLE Page AUTO\_INCREMENT = 1*   
  
So again the key behind this is for many objects, as long as they are not tied in with foreign keys, all I need is an ID. As we saw in examples up here, all I need is an object, set the ID and I can delete it. I don't care what all of the other fields of the class are. I can try that with certain objects and it will fail but otherwise I can create these networks. If I want to delete it in a smarter way you can simply, you know do queries and whatnot, you don't have to call multiple deletes for each sub object, you could create your own query but that's outside the scope of the basic delete statement we're talking about here, so we'll leave that for you to play with it later.

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a01_it_enus/output/html/course_transcript.html#top)

Basic Tests – Reading

Learning Objective

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

* *load data using Java Hibernate*

**1. Loading data in Java Hibernate**

As Hibernates designed to manage the basic Create, Read, Update and Delete, the CRUD tasks for us, we're going to look this time at the reading part. Couple of different ways to read objects in Hibernate. When we say read here, we're not looking at querying, we're looking at loading it on basically by the ID. We know their primary key, load up that object, given the primary key. And so the first method we're going to look at is called GET, so from the session I can say GET me an object of this class. So again I'm not querying deeply, I'm just saying it I know it's this class and then pass in any serialized object that is the ID and in our case we've a number, this ID, so we have to pass in capital L, Long, as an object to query on that. So if we run our little test case here for this, we're going to go through to the database and load up that object with that ID of one. You can see it loads both this title of the story and it's also going to load, I'm sorry, the title of the story and the story date, plus all the page information. So it loads the full depth of the object itself plus any associated object with it, assuming that we've to cascade on that, and it's been told to go off and load those. We'll actually see a little bit how in a minute how that loads those.   
*Java Eclipse is open and the Read.java tab selected. The code in the editor is:  
    
   // Load requires that the object is present or it will fail.  
   public Story loadByID(long id)  
   {  
       Story story = (Story) session.load(Story.class, new Long(id));  
       return story;  
   }  
  
   // An existing object can be loaded with values as an option  
   public Story loadToExisting(Story story, long id)  
   {  
       session.load(story, new Long(id));  
  
The presenter then clicks the CommandLineMain tab and the following code is displayed:   
  
//  Create Demo  
//            create.run(Create.PERSIST);  
//            create.run(Create.SAVE);  
//            create.run(Create.SAVE\_OR\_UPATE);  
//            create.run(Create.SAVE\_OR\_UPATE\_EXISTING);  
  
             
//  Read Demo  
//            read.run(Read.GET\_BY\_ID);  
//            read.run(Read.GET\_MISSING);  
//            read.run(Read.LOAD\_BY\_ID);  
//            read.run(Read.LOAD\_MISSING);  
//            read.run(Read.LOAD\_TO\_EXISTING);  
//            read.run(Read.REFRESH);  
//            read.run(Read.SHOW\_LAZY\_LOADING);  
  
  
//  Update Transient Demo              
//          update.run(Update.UPDATE\_FROM\_SCRATCH);  
//          update.run(Update.UPDATE\_WITHOUT\_ALL);  
    
             
//  Update Persistent Demo              
//          update.run(Update.LOAD\_AND\_UPDATE);  
//          update.run(Update.LOAD\_AND\_CALL\_UPDATE);  
//          update.run(Update.LOAD\_AND\_MERGE);  
  
The presenter clicks Run, and the output is displayed in the Console. The presenter focuses on the following section of the output:   
  
Show the story we just read  
  
Welcome to Hibernate! - Tue Feb 25 11:44:46 PST 2014  
  
Page 1   
  
I am the fist page  
  
Page 2   
  
I am the second page  
  
creation: Tue Feb 25 11;44:46 PST 2014 modified: Tue  
Test Complete  
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!  
Hibernate: delete from Page  
Hibernate: delete from Story  
Hibernate: ALTER TABLE Story AUTO\_INCREMENT = 1  
Hibernate: ALTER TABLE Page AUTO\_INCREMENT = 1*   
  
Now the GET call is specifically designed that if I call get and that ID is not out there...So I'm going to make the same call again but I'm going to call out an ID that doesn't exist, it's okay, I just get back a null, and then I have some logic inside of my test case to say I didn't load the story. That's okay. The GET is specifically created to be soft on you as a developer, if you load by an ID that doesn't exist. So the load call, on the other hand, is assuming that it does exist out there. So the load basically operates exactly like the GET call, you pass in the Story.class...the type of the class that you want to get...this as any class that mapped either through annotations or through XML. And I pass in the ID associated with that and then I get that object back. So the basic call to load, we can run the test for that. And we can see that we load up that object just fine. Again we got the title, this data, the story, and then the pages that's going to come along with that. The difference comes when I load a story that doesn't exist in the database, if I had given an ID that doesn't exist in the database that is a fail call and it actually gets an exception, you can see an ObjectNotFoundException, no row with the given identifier exists:. So I do not have a story number one out there.   
*The presenter selects the CommandLineMain.java tab and removes the // from the line:  
  
// read.run(Read.GET\_MISSING);.  
  
He clicks Run. The output is displayed in the Console. The presenter focuses on the section of output:  
  
No story was loaded  
Test Complete  
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!  
Hibernate: delete from Page  
Hibernate: delete from Story  
Hibernate: ALTER TABLE Story AUTO\_INCREMENT = 1  
Hibernate: ALTER TABLE Page AUTO\_INCREMENT = 1       
  
The presenter selects the Read.java tab and then selects the CommandLineMain.java tab. He removes the // from the code line:  
  
// read.run(Read.LOAD\_BY\_ID);.  
  
The he clicks Run. He focuses on the output:  
  
Welcome to Hibernate! - Tue Feb 25 11:44:46 PST 2014  
  
Page 1   
  
I am the fist page  
  
Page 2   
  
I am the second page  
  
Test Complete  
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!  
Hibernate: delete from Page  
Hibernate: delete from Story  
Hibernate: ALTER TABLE Story AUTO\_INCREMENT = 1  
Hibernate: ALTER TABLE Page AUTO\_INCREMENT = 1  
  
The presenter selects the CommandLineMain tab and removes the // from the code line:  
  
read.run(Read.LOAD\_MISSING);.  
  
He clicks Run. The presenter focuses on the error in the output:  
  
No row with the given identifier exists: (hibernate.domain.Story#1)*  
  
And so the assumption is in your coding, if you're going to be fault tolerant, if you're going to look for null you can use GET. If you absolutely are assuming it's out there in your code, you should be using a LOAD. Now there's other options for loading. The next case we can look at here is where I'm loading to an existing object. So I have an object that's out there, and maybe I want to reuse, maybe I'll have a cached object so I just don't want to have to do so much garbage collection. This is really nice if I know I'm going to do some batch processing, where I don't have to constantly be creating new objects. So I can load to an existing object inside of here where I pass in the object itself instead of the class. And as I do that I don't have to go through and create another object. So when I run this solution, it comes up and as you would guess, it loads the object just as it did every other time; title, date, pages, all that stuff gets loaded and put inside of the existing object. So again it's just really more of a convenience to not require you to build new objects every single time you make a call. The other option is you're off there working with an object. Say you have an object that you know is going to be loaded for a long time. And so thus, something could change behind the scenes...the database could be updated, there is multiple sources of data to be updated, but I still have that object loaded in my session. Well the refresh allows me to handle that situation.   
*The presenter removes the // from the code line:  
  
// read.run(Read.LOAD\_TO\_EXISTING);.  
  
He selects the Read.java tab, and then selects the CommandLineMain.java tab again. He clicks Run and the output is displayed in the Console. The presenter focuses on the output:   
  
Welcome to Hibernate! - Tue Feb 25 11:44:46 PST 2014  
  
Page 1   
  
I am the fist page  
  
Page 2   
  
I am the second page  
  
Test Complete  
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!  
Hibernate: delete from Page  
Hibernate: delete from Story  
Hibernate: ALTER TABLE Story AUTO\_INCREMENT = 1  
Hibernate: ALTER TABLE Page AUTO\_INCREMENT = 1  
  
The presenter selects the Read.java tab and then selects the CommandLineMain.java tab. He removes the // from the code line:  
  
// read.run(Read.REFRESH);.  
  
He selects the Read.java tab and clicks Run. The Run As dialog box opens. The presenter selects the CommandLineMain.java tab and clicks Run again. The output is displayed in the Console.*   
  
So I have an existing object that's loaded, I just simply call session.refresh. It's going to go look at that object, it's going to see if anything has changed, oops, sorry I need to run my test case, not that class, it's going to see if anything has changed and it's going to load it appropriately. So in our test case here, you can see it's loaded it up, I'm calling the REFRESH, it just loads it up again, and it refills that object and goes from there, it doesn't make a decision whether it needs to be refreshed; it just overwrites everything, it just gives you a new fresh look of that object. So again if some other process is out there and change the database whether it's a thread in your application or maybe it's another application altogether or a user doing it by hand, I can just ensure my application's using that long lasting object with the freshest state. The last thing I want to show you about is...we kind of talked about before...is the idea of lazy loading. In all of these samples we've been getting a page separate from the story; the pages are loaded separately. And it's important to manage the load properly. By default Hibernate mostly configures these to lazy load. And so when I do lazy loading, what basically is happening is I don't load the secondary objects, the associated objects, until I need them.   
*The presenter focuses on the output:   
  
Hibernate: /\* load hibernate.domain.Page \*/  select page  
Hibernate: /\* load hibernate.domain.Page \*/ select page   
Hibernate: /\* load hibernate.domain.Story \*/ insert into Story  
  
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!  
Show the story we just read  
  
Welcome to Hibernate! - Tue Feb 25 11:44:46 PST 2014  
  
Page 1   
  
I am the fist page  
  
Page 2   
  
I am the second page  
  
The presenter removes the // from the code line:  
  
// read.run(Read.SHOW\_LAZY\_LOADING);.  
  
The presenter selects the Read.java tab.*  
  
So in the code I'm about to run, I'm going to load that object by the ID, and I'm going to go ahead and close that and commit that transaction where I loaded that object. I'm not changing anything there, I'm just saying okay I'm done with that transaction. And then I'm going to go off and print the story. It's going to happen after it, in this case. It's going to have an interesting net affect on that because when I commit it, I'm basically closing that connection to the database and the net result of that is I get a failure in lazy loading, so I attempt to load that object again even though it's already out there. And that gives me an issue within my solution. So I need to make sure, within the session management side of it, that I am loading the object and all of its associate objects at the same time before I start messing with transaction boundaries. That might mean switching it to do eager fetching instead of lazy fetching, or whatnot. That's a topic for another time, but it's really key to understand within your loading, the context within your object's going to be used. That's the whole point of that, it's not how to manage it but just to understand the loading does have well pass the method here in a little bit of ramification inside of how it's used.   
*The presenter selects the CommandLineMain tab and clicks Run. An error message displays in the Console. The presenter focuses on the output:   
  
same identifier value was already associated with the session  : [hibernate.domain.Story#1].*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a01_it_enus/output/html/course_transcript.html#top)

Interceptors for Hibernate Actions

Learning Objective

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

* *use the Interceptor API in Java Hibernate to access, monitor, maintain, and manage key events in the persistence life cycle*

**1. Using the Interceptor API in Hibernate**

Part of the value of the Hibernate framework, is it insulates us from all the details that go into managing the interactions with the database. That's why we have it – we want to not have to worry about that in all of our code. That being said, in some cases, we do need some custom behaviors to better manage those transactions and those details. For instance, maybe we want an audit trail. Every time we do a change to the database, we want to log something appropriately or otherwise check permissions of...should the user be doing this? That might help us perform some maintenance on some of these actions; maybe we're calling extra triggers or things, based off of certain behaviors. Doesn't really matter what the motivation is; the idea is I want to see behind that curtain of what Hibernate's doing. And one of the ways to make that happen, is to use interceptors inside of Hibernate. So I'm creating my own little interceptor here that doesn't do anything interesting – except for letting us know what interceptors do – but it extends out this EmptyInterceptor interface inside of Hibernate, and that gives us access to all these different things I can do. So the first set is basically around transactions.   
*Java Eclipse is open on the MyInterceptor.java tab. The code that displays in the Editor is:  
  
package hibernate.interceptor;  
  
import java.io.Serializable;  
import java.util.Iterator;  
  
public class MyInterceptor extends EmptyInterceptor  
{  
   @Override  
   public void afterTransactionBegin(Transaction tx)  
   {  
       System.err.println("$! afterTransactionBegin");  
       super.afterTransactionBegin(tx);  
   }  
  
   @Override  
   public void afterTransactionCompletion(Transaction tx)  
   {  
       System.err.println("$! afterTransactionCompletion");  
       super.afterTransactionCompletion(tx);  
   }  
  
   @Override  
   public void beforeTransactionCompletion(Transaction tx)  
   {  
       System.err.println("$! beforeTransactionCompletion");  
       super.beforeTransactionCompletion(tx);  
  
The presenter scrolls down to reveal additional code:   
  
@Override  
   public int[] findDirty(Object entity, Serializable id, Object[] currentState, Object[] previousState, String[] propertyNames, Type[] types)  
   {  
       System.err.println("$! findDirty");  
       return super.findDirty(entity, id, currentState, previousState, propertyNames, types);  
   }  
  
   @Override  
   public Object getEntity(String entityName, Serializable id)  
   {  
       System.err.println("$! getEntity");  
       return super.getEntity(entityName, id);  
   }  
  
   @Override  
   public String getEntityName(Object object)  
   {  
       System.err.println("$! getEntityName");  
       return super.getEntityName(object);  
   }  
  
   @Override  
   public Object instantiate(String entityName, EntityMode entityMode, Serializable id)  
   {  
       System.err.println("$! instantiate");  
       return super.instantiate(entityName, entityMode, id);  
   }  
  
   @Override  
   public Boolean isTransient(Object entity)  
   {  
       System.err.println("$! isTransient");  
       return super.isTransient(entity);  
   }  
  
   @Override  
   public void onCollectionRecreate(Object collection, Serializable key) throws CallbackException  
   {  
       System.err.println("$! onCollectionRecreate");  
       super.onCollectionRecreate(collection, key);  
   }  
  
   @Override  
   public void onCollectionRemove(Object collection, Serializable key) throws CallbackException  
   {  
       System.err.println("$! onCollectionRemove");  
       super.onCollectionRemove(collection, key);  
   }  
  
   @Override  
   public void onCollectionUpdate(Object collection, Serializable key) throws CallbackException  
   {  
       System.err.println("$! onCollectionUpdate");  
       super.onCollectionUpdate(collection, key);  
   }  
  
   @Override  
   public void onDelete(Object entity, Serializable id, Object[] state, String[] propertyNames, Type[] types)  
   {  
       System.err.println("$! onDelete");  
       super.onDelete(entity, id, state, propertyNames, types);  
   }  
  
   @Override  
   public boolean onFlushDirty(Object entity, Serializable id, Object[] currentState, Object[] previousState, String[] propertyNames, Type[] types)  
   {  
       System.err.println("$! onFlushDirty");  
       return super.onFlushDirty(entity, id, currentState, previousState, propertyNames, types);  
   }  
  
   @Override  
   public boolean onLoad(Object entity, Serializable id, Object[] state, String[] propertyNames, Type[] types)  
   {  
       System.err.println("$! onLoad");  
       return super.onLoad(entity, id, state, propertyNames, types);  
   }  
  
   @Override  
   public String onPrepareStatement(String sql)  
   {  
       System.err.println("$! onPrepareStatement");  
       return super.onPrepareStatement(sql);  
   }  
  
   @Override  
   public boolean onSave(Object entity, Serializable id, Object[] state, String[] propertyNames, Type[] types)  
   {  
       System.err.println("$! onSave");  
       return super.onSave(entity, id, state, propertyNames, types);  
   }  
  
   @Override  
   public void postFlush(Iterator entities)  
   {  
       System.err.println("$! postFlush");  
       super.postFlush(entities);  
   }  
  
   @Override  
   public void preFlush(Iterator entities)  
   {  
       System.err.println("$! preFlush");  
       super.preFlush(entities);*   
  
So afterTransactionBegin, this is some work I might want to do. Perhaps I want to log what's going on in the transaction after it's started. After the transaction's completed, maybe there's some other additional work I want to do. Before the transaction's completed, I might want to go off and find "dirty" objects inside of my system or get entities. So there's all sorts of behaviors; checking if things are transient or not. All these different behaviors are things that Hibernate does regularly; they're events that happen within Hibernate. And for the few cases that I really care about the management and care of those events, then I can use this – in kind of an aspect-oriented sort of programming methodology – to keep track of what's going on inside of my system. So if I come over here to my example here and I **Run** this in everyday code, it all runs and nothing interesting happens, and it goes – except now, since I've plugged in MyInterceptor, I have all these extra little printlines that are happening. And I put inside of here printlines with this dollar sign/exclamation point, so I can see them very easily. Now how did that happen? Well MyInterceptor by itself, doesn't do anything; I have to plug that interceptor into my configuration – and let me show you where I did that. So right here, is a Helper class I have that loads my Hibernate Configuration. This is the code that loads your hibernate.cfg.xml file.   
*The presenter selects the CommandLineMain.java tab and the code is:  
  
//   update.run(Update.LOAD\_AND\_CALL\_UPDATE);  
//   update.run(Update.LOAD\_AND\_MERGE);  
//   update.run(Update.LOAD\_AND\_UPDATE);  
//   update.run(Update.UPDATE\_FROM\_SCRATCH);  
//   update.run(Update.UPDATE\_WITHOUT\_ALL);  
  
       System.err.println("Test Complete");  
           System.err.println("!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!");  
           TestHelper.cleanup(factory.getCurrentSession());  
           factory.getCurrentSession().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 Run and the output is displayed in the Console. The presenter focuses on a section of the output:   
  
$! preFlush  
$! findDirty   
$! findDirty   
$! findDirty   
$! onPrepareStatement  
  
He selects the ConfigHelper.java tab and the code is displayed:   
  
package hibernate.util;  
  
public class ConfigHelper  
{  
   private static final SessionFactory sessionFactory = buildSessionFactory();  
  
   private static SessionFactory buildSessionFactory()  
   {  
       try  
       {  
           // Create the SessionFactory from hibernate.cfg.xml  
           Configuration configuration = new Configuration();  
           configuration.configure();  
           ServiceRegistry serviceRegistry = new StandardServiceRegistryBuilder().applySettings(configuration.getProperties()).build();          
           return configuration.buildSessionFactory(serviceRegistry);  
       } catch (Throwable ex)  
       {  
           // Make sure you log the exception, as it might be swallowed  
           System.err.println("Initial SessionFactory creation failed." + ex);  
           throw new ExceptionInInitializerError(ex);  
       }  
   }  
  
   public static SessionFactory getSessionFactory()  
   {  
       return sessionFactory;  
   }  
}*   
  
After I set up this file, I'm going to tell the Configuration to use MyInterceptor. And so this is basically a little listener class that any time any of these things happen, it's going to call, and you can see inside of each of these, I've done a System.err.println and talking about what's happening. So as I executed the code then, you can see when I...up front...let's go, let's keep scrolling up, scrolling up...when I started the transaction, it lets me know – it's after the transaction began. When I've created a PrepareStatement, it lets me know that that happened. When I called the Save, it lets me know that happened, and so on and so forth. All of these different states get called into MyInterceptor. I can do something in that state or not. I can actually ignore any of these methods I want to; I don't have to implement them – I can just leave them as returning nothing, and being blank, and whatnot. And just to show you, once I've plugged in the Interceptor to the Config, it runs in every one of your operation. I have an entirely separate execution here that uses the same Helper to load the Config – it uses the same ConfigHelper to load it up inside of here. So when I **Run** this one, which is really just doing a bunch of querying, I still get tons and tons and tons of these Interceptor calls. Every time it's finding a dirty object, it calls MyInterceptor; every time it's doing a PrepareStatement, it's calling MyInterceptor.   
*The presenter selects the MyInterceptor.java tab. He then focuses on the output in the Console:   
  
$! afterTransactionBegin  
  
The story to be updated, just added  
$! getEntityName  
$! isTransient  
$! onSave  
$! onPrepareStatements  
  
The presenter selects the ConfigHelper.java tab. Then he selects the QueryMain.java tab and the code is:  
  
package hibernate;  
  
import java.util.List;  
  
import org.hibernate.Query;  
import org.hibernate.Session;  
import org.hibernate.SessionFactory;  
  
public class QueryingMain  
{  
   private Session session;  
     
   public static void main(String[] args)  
   {  
       new QueryingMain().run();  
   }  
  
   private void run()  
   {  
       SessionFactory factory = null;  
       try  
       {  
           factory = ConfigHelper.getSessionFactory();  
  
           session = factory.getCurrentSession();  
           session.getTransaction().begin();  
             
           BulkInsert adder = new BulkInsert(session);  
           adder.createABunchForSearching();  
  
           TestDriver driver = new TestDriver(session);  
  
// Basic Querying  
  
The presenter clicks Run and the output is displayed in the Console. An example of the output is:  
  
$! findDirty  
$! findDirty  
$! findDirty  
$! findDirty  
$! findDirty  
$! findDirty  
$! onPrepareStatements*   
  
And so around all of these CRUD actions, around all of these Transaction actions, around all these interesting things I don't normally see behind the scenes, I can plug in something without having to understand or modify the Hibernate code and take actions appropriately. Whether it's proactive or just simply reactive of logging and auditing, it allows me to have a better control of how Hibernate behaves and interacts with the rest of my code and solution. So the Interceptor gives us that depth of not having to recreate Hibernate, but just simply plug in to the existing life-cycle process that's happening.

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a01_it_enus/output/html/course_transcript.html#top)

Running Hibernate under JPA

Learning Objective

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

* *load an application using the JPA approach instead of Java Hibernate*

**1. Using the JPA method in Java Hibernate**

JPA is a Java standard that provides a point of replacement for your ORM framework. Loading an application using the JPA approach takes a few minor tweaks, but it's very similar to bootstrapping through Hibernate. By using JPA you give yourself the option to replace Hibernate at the cost of the many custom features Hibernate provides, but that's a trade-off to be discussed. In some cases you want to choose to execute Hibernate under JPA instead of underneath base Hibernate in itself. It allows you to remove Hibernate if you so choose. Doing that pretty significantly changes the way I startup the application, but after that everything pretty much runs the same. So the first thing we can point to is, you need the Hibernate jar files that are out there, that you would normally add your Hibernate application, this is my library that points to all the required jar files. But in addition to that, you have to include this one, maybe more in different versions of Hibernate, jar file that allows Hibernate to act as an EntityManager. And EntityManager is the key class in JPA that loads up and allows you to do persistence actions. Past that I'm no longer using the Hibernate config file in the same way. Instead I'm using a persistence.xml file. And persistence.xml is defined by JPA to live underneath the META-INF folder underneath your class path.   
*Java Eclipse is open on the EntitymanagerExample.java tab. The code that is displayed in the editor is:  
  
package hibernate.jpa;  
  
import hibernate.domain.Story;  
import hibernate.test.StoryGenerator;  
  
import java.io.IOException;  
  
import javax.persistence.EntityManager;  
import javax.persistence.EntityManagerFactory;  
import javax.persistence.Persistence;  
  
public class EntityManagerExample  
{  
   public static EntityManagerFactory getEntityManagerFactory()  
   {  
      EntityManagerFactory emf = Persistence.createEntityManagerFactory("mysql-demo");  
      return emf;  
   }  
     
   public static void main(String[] args) throws IOException  
   {  
       EntityManager em = null;  
       try  
       {  
           EntityManagerFactory f = getEntityManagerFactory();  
             
            em = f.createEntityManager();  
             
           StoryGenerator sg = new StoryGenerator();  
           Story s = sg.generate(1).get(0);   
             
           em.getTransaction().begin();  
           em.persist(s);  
           em.getTransaction().commit();  
       } catch (Throwable t)  
       {  
           t.printStackTrace();  
       } finally  
       {  
           // To kill the run else Hibernate leaves it active.  
           em.close();  
           System.exit(0);  
       }  
             
   }  
}  
  
In the Package Explorer, the EntityManager folder is expanded. It contains the src, hibernate.jpa, EntityManager, Meta- INF, and persistance.xml files. The JRE System Library, Hibernate, and Hibernate-JPA libraries are also visible. The presenter expands the Hibernate library. It contains the jar files: antlr-2.7.7.jar, dom4j-1.6.1.jar, hibernate-common, hibernate-core-4.3, hibernate-jpa-2.1-a, jandax-1.1.0.Final, javassist-3.18.1-GA, jboss-logging-3.1.3, jboss logging-anno, jboss-transaction, and mysql-connector. The presenter expands the Hibernate - JPA file to reveal the hibernate-entitymanager-4.3.1Final.jar file. He then expands the META-INF file in the Package Explorer, and the persistance.xml file is displayed.*   
  
And so the Hibernate config...that's just in your base directory of your class path. This is in the META-INF folder. It acts much the same way. Inside of here, inside of JPA I point to classes that have annotations that are part of them. I point to the database to get that set up, again I have the driver, the URL, the username, the password. I set up the dialect to point to the appropriate one. And then I can do different persistence framework. In this case hibernate customize parameters such as do I want to show the SQL or things like that. They're all different options that can show inside of there to do customize what's going on with your framework. So this is very much like a Hibernate config file, there's a few different options but it's abstracted out to not be typed to Hibernate per say. Now the one thing I wish you note here is, I'm naming my persistence to that. I can, inside of JPA, define more than one of this if I have more than one type of mapping I want to have you going on. And so when I go back to my code, I need to load up from this class static method   
Persistence.createEntityManagerFactory, which persistence you want to create, that's the name I saw before, the mysql-demo, that's the name I saw before, I need to match it up inside of here.   
*The presenter selects the persistance.xml tab and the code is visible in the editor:   
  
<?xml version="1.0" encoding="UTF-8" standalone="no"?>  
<persistence xmlns="http://java.sun.com/xml/ns/persistence"  
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="2.0"  
xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/xml/ns/persistence/persistence\_2\_0.xsd">  
<persistence-unit name="mysql-demo" transaction-type="RESOURCE\_LOCAL">  
  
<description>MySQL Persistence Unit</description>  
<provider>org.hibernate.ejb.HibernatePersistence</provider>  
  
<class>hibernate.domain.Story</class>  
<class>hibernate.domain.Page</class>  
  
<properties>  
<property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver" />  
<property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/BigNews" />  
<property name="javax.persistence.jdbc.user" value="root" />  
<property name="javax.persistence.jdbc.password" value="test" />  
<property name="hibernate.dialect" value="org.hibernate.dialect.MySQLInnoDBDialect" />  
<property name="hibernate.show\_sql" value="true" />  
<property name="hibernate.format\_sql" value="true" />  
<property name="hibernate.transaction.flush\_before\_completion"  
value="true" />  
<property name="hibernate.hbm2ddl.auto" value="update" />  
</properties>  
  
</persistence-unit>  
</persistence>*   
  
Past that I've loaded the EntityManagerFactory and I'm ready to go. The EntityManagerFactory gets me my instance and from that I can create my instance of the EntityManager, which allows me to create transactions, call saves, do queries...all this sort of stuff that I wouldn't normally do at the Hibernate SessionFactory. So let's see this guy of action, let's go ahead and run it. Now the other thing it does slightly differently is it doesn't tie itself into logging nearly as cleanly or simply as Hibernate does by default. You can see I get a warning up top, I don't have a new logging set up. Not a big deal if I don't need it. It's still going to spit out my queries, and you can see I did an inserting the story, and it inserts the page and it actually kind of spits out the queries a little cleaner than it does in Hibernate instead of being out in one line, I can make it a bit more readable if that's desirable for me. That might not be. In order to enable logging, we'd have to do the extra configuration to get log for Jsetup and be able to have that happen, where Hibernate itself uses simple log for J to make it basically get out-of-the-box basic logging even if we haven't configured anything. So not to say that running as an entity manager is a bad thing, it's just that Hibernate certainly rewards you for using its way of starting up and bootstrapping, but this is another great option if your application wants to use, or already has plug-ins to use the entity manager from a JPA provider.   
*The presenter navigates back to the EnityManagerExample.java tab. He clicks Run and the output is displayed in the Console. The presenter focuses on the output:  
  
log4j:WARN No appenders could be found  
log4j:WARN Please initialize the log4j  
log4j:WARN See http://logging.apache.com  
Hibernate:  
         Insert  
         into  
               Story  
               (creationDate, lastModifiedDate...  
               values  
               (?, ?, ?, ?)   
Hibernate:  
         Insert  
         into  
               Page  
               (creationDate, htmlIncluded...  
               values  
               (?, ?, ?, ?, ?, ?)*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a01_it_enus/output/html/course_transcript.html#top)

Managing Sessions and Transactions

Learning Objective

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

* *manage sessions and transactions in Java Hibernate*

**1. Manage sessions and transactions**

The Hibernate session does much more for us than just simply giving us access to the database. Obviously it provides us with HQL query or predefined CRUD operations that gets us to the database. It creates the SQL for us and it maps result set, but it also has a certain persistent state, it has a first level caching that happens inside of there. So managing that session, managing that transaction and the session is key. We want to use the concept of a data access object and in my tests for this presentation I have something that sort of kind a looks like it, it's broken up a little bit, but I have Read classes and Create classes and Delete classes. Normally they're around one idea. And in this case they are, they are around story, the idea of a story. But I'm going to have methods inside of there to get story, methods inside of there to delete story, method inside of there to update story. In this case, I have four classes for each of those, just to be able to show all the different ways I can update stories involved there. There's many of them; loads, and gets, and updates, and save, and updates, and all that sort of stuff. But ideally I'd have one class that does all those operations. Now I want to have more than just story classes. I want to have all the other business objects throughout there, so how do I manage session between them? Often I'm going to have sessions that span between them.   
*Java Eclipse is open on the CommandLineMain.java tab. The code is displayed in the editor:  
  
  public CommandLineMain()  
   {  
       factory = ConfigHelper.getSessionFactory();  
       read   = new Read(factory);  
       create = new Create(factory, read);  
       update = new Update(factory, create);  
       delete = new Delete(factory, create);  
   }  
  
   private void run()  
   {  
       try  
       {  
           factory.getCurrentSession().beginTransaction();  
  
//  Create Demo  
//            create.run(Create.PERSIST);  
//            create.run(Create.SAVE);  
//            create.run(Create.SAVE\_OR\_UPATE);  
//            create.run(Create.SAVE\_OR\_UPATE\_EXISTING);  
  
  
//  Read Demo  
//            read.run(Read.GET\_BY\_ID);  
//            read.run(Read.GET\_MISSING);  
//            read.run(Read.LOAD\_BY\_ID);  
//            read.run(Read.LOAD\_MISSING);  
//            read.run(Read.LOAD\_TO\_EXISTING);  
//            read.run(Read.REFRESH);  
//            read.run(Read.SHOW\_LAZY\_LOADING);  
  
The presenter selects the Read.java tab and the code is displayed:  
  
package hibernate.crud;  
  
import hibernate.domain.Story;  
import hibernate.test.StoryGenerator;  
import hibernate.test.StoryToString;  
  
import org.hibernate.Session;  
import org.hibernate.SessionFactory;  
  
public class Read  
{  
   private SessionFactory factory;  
   private Session session;  
   private Create create;  
  
   public static final int GET\_BY\_ID         = 0;  
   public static final int GET\_MISSING       = 1;  
   public static final int LOAD\_BY\_ID        = 2;  
   public static final int LOAD\_MISSING      = 3;  
   public static final int LOAD\_TO\_EXISTING  = 4;  
   public static final int REFRESH           = 5;  
   public static final int SHOW\_LAZY\_LOADING = 6;  
  
   public Read(SessionFactory factory)  
   {  
       this.factory = factory;  
       this.session = factory.getCurrentSession();  
       // We need to create one here to avoid a circular dependency in constructors  
       this.create  = new Create(factory, this);   
   }  
  
   public void run(int testCase)  
   {  
       Story story = null;  
       long id = 1;  
         
         
       if (testCase != GET\_MISSING && testCase != LOAD\_MISSING)  
       {  
           System.err.println("\n!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!");  
           System.err.println("Insert the story to read");  
           story = new StoryGenerator().buildStory();  
           create.callPersist(story);  
       }  
  
       System.err.println("!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!");  
       switch (testCase)  
       {  
           case GET\_MISSING:  
               System.err.println("No data was loaded but...");  
           case GET\_BY\_ID:  
               System.err.println("Call Get");  
               story = getByID(id);  
           break;  
           case LOAD\_MISSING:  
               System.err.println("No data was loaded but...");  
           case LOAD\_BY\_ID:  
               System.err.println("Call load");  
               story = loadByID(id);  
           break;  
           case LOAD\_TO\_EXISTING:  
               System.err.println("Call load to existing object");  
               session.getTransaction().commit();  
                 
               // We have to commit and redo the session to get the previous object saved and ready to go for this one.  
               session = factory.getCurrentSession();  
               session.beginTransaction();  
                 
               story = loadToExisting(new Story(), id);  
           break;  
           case REFRESH:  
               System.err.println("Refresh an object");  
               story = refresh(story);  
           break;  
           case SHOW\_LAZY\_LOADING:  
               System.err.println("Call load but fail as the object is lazy loading enabled");  
               showLazyLoading();  
           return;  
           default:  
               System.err.println("Invalid test case selected");  
           return;  
       }  
       System.err.println("\n!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!");  
       System.err.println("Show the story we just read");  
       StoryToString.printStory(story);  
   }  
  
     
   // Get will return an object, or null if it is not found.  
   public Story getByID(long id)  
   {  
       Story story = (Story) session.get(Story.class, new Long(id));  
       return story;  
   }  
     
    
   // Load requires that the object is present or it will fail.  
   public Story loadByID(long id)  
   {  
       Story story = (Story) session.load(Story.class, new Long(id));  
       return story;  
   }*   
  
So each of these classes, you can see, I pass in either a SessionFactory or in some cases, the session itself to the classes as part of a constructor. And then I might have them inter-depend on each other and I pass them in. And from them I'm going to get the current session that I'm going to be tracking. Now whether I use the factory or the session largely depends on how aware of these guys need to be of the factory or session in the less aware the better. Generally speaking we want to manage the transactions of sessions outside the scopes of these objects. Now we might have a second layer of controller that does manage those and that's what we're kind of doing with our test cases. So in my running application here, you can see this is where I'm beginning transactions. I'm running whatever test case that I have going on at that time. At the end I'm closing out, I'm committing my transactions. And that's a great design. I have data access objects that are reusable. You hand me a session, I do work with it. And ideally I'm not handing that session each method, I had a session that's going to be used for that whole class, it might be set over again but it's going to be used for the whole class. And then I've a controller that manages the beginning and ending of transactions that go across with that as we can see inside of here.   
*The presenter selects the Create.java tab and the code is displayed in the editor:  
  
package hibernate.crud;  
  
import hibernate.domain.Story;  
import hibernate.test.StoryGenerator;  
import hibernate.test.StoryToString;  
  
import org.hibernate.Session;  
import org.hibernate.SessionFactory;  
  
public class Create  
{  
   private Session session;  
   private Read read;  
   private StoryGenerator sg = new StoryGenerator();  
  
   public static final int PERSIST                = 0;  
   public static final int SAVE                   = 1;  
   public static final int SAVE\_OR\_UPDATE          = 2;  
   public static final int SAVE\_OR\_UPDATE\_EXISTING = 3;  
  
   public Create(SessionFactory factory, Read read)  
   {  
       this.session = factory.getCurrentSession();  
       this.read    = read;  
   }  
  
   public void run(int testCase)  
   {  
       Story story = sg.buildStory();  
       System.err.println("\n!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!");  
       switch (testCase)  
       {  
           case PERSIST:  
               System.err.println("Call Persist");  
               callPersist(story);  
           break;  
           case SAVE:  
               System.err.println("Call Save");  
               callSave(story);  
           break;  
           case SAVE\_OR\_UPDATE:  
               System.err.println("Call Save or Update");  
               callSaveOrUpdate(story);  
           break;  
           case SAVE\_OR\_UPDATE\_EXISTING:  
               System.err.println("Call Save or Update on Existing");  
               callSaveOrUpdateExisting(story);  
           break;  
           default:  
               System.err.println("Invalid test case selected");  
           return;  
       }  
         
       session.evict(story);  
         
       System.err.println("\n!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!");  
       System.err.println("Read and print the story from the database");  
       StoryToString.printStory(read.getByID(1));  
   }  
     
  
   public void callPersist(Story story)  
   {  
       // Persist may or may not get the ID at this moment depending on how the Database works...  
       session.persist(story);  
       // Thus at this point, the persistent object may not have an ID set, which is quicker, but not if you need the ID  
   }  
  
   public Long callSave(Story story)  
   {  
       // Save will ensure the ID is returned when it is generated on the DB side, and return the object  
       Long id = (Long) session.save(story);  
       return id;  
   }  
  
  
The presenter then clicks the CommandLineMain.java tab, and scrolls down to reveal the additional code:   
  
  
//  Update Transient Demo              
//          update.run(Update.UPDATE\_FROM\_SCRATCH);  
//          update.run(Update.UPDATE\_WITHOUT\_ALL);  
  
         
//  Update Persistent Demo              
//          update.run(Update.LOAD\_AND\_UPDATE);  
//          update.run(Update.LOAD\_AND\_CALL\_UPDATE);  
//          update.run(Update.LOAD\_AND\_MERGE);  
           
  
//   Delete Demo  
//            delete.run(Delete.DELETE\_A\_PAGE);  
//            //delete.run(Delete.DELETE\_PARENT\_ONLY);  
//            delete.run(Delete.DELETE\_TRANSIENT);  
//            delete.run(Delete.DELETE\_PERSITENT);  
             
             
           System.err.println("Test Complete");  
           System.err.println("!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!");  
           TestHelper.cleanup(factory.getCurrentSession());  
          // Session s = factory.openSession();   
          // S.beginTransaction()";  
         //s.getTransacion().commit();  
factory.getCurrentSession().getTransaction().commit(); factory.getCurrentSession().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);  
       }  
   }  
}*   
  
That controller might be manage by hand, as we're doing it here, or you might be using frameworks like Spring, or Juice, or some dependency injection framework to manage your transactions externally. So past that, it's really important to understand how these things are opened and closed. I want to show you some examples of doing it wrong. So if I come into my Read class here, and let's say I'm overly managing my transaction, so after I read something I want to start a new transaction to go and change something elsewhere. And so I go to my factory and I open up a new session. And then I begin a new transaction at this point. So I just saved an object, now I'm going to do some other work. So if I do this accidentally...don't realize what's going on here...let me go and run this application. It can have some interesting ramifications. So I know I just saved the data but when I run to go load the data, it says there is no story loaded, why? It's because I started a brand new session. When I started a brand new session here, I started a brand new cache for this session. So the old one hasn't been committed and thus it's not in the database. The new one is looking in the database for this information, it's not in my cache and so it's gone. So if I cut up this lines back out and I run it, again, again I got to run my command line main here.   
*The presenter selects Read.java, and then selects the CommandLineMain,java tab. He removes the // from the code lines:  
  
// Session s = factory.openSession();  
// S.beginTransaction()";  
//s.getTransacion().commit();  
  
Then he clicks Run. The output is displayed in the editor and the presenter focuses on the section:   
  
No story loaded  
Test Complete  
  
The presenter selects the Read.java tab and clicks Run. The Run As dialog opens, and the presenter then selects the CommandLineMain.java tab and clicks Run again.*   
  
You can see that the test works as intended. Now this isn't just a ridiculous sample, it's actually something I ran into while trying to build these tests. It's you know...it's not common in code that I inserted the database and then I immediately go and load it again, but it is for a demonstration like this, I want to setup some test, I want to setup some samples. And so believe you me, I've run into this in real coding before as well, but you have to be very careful managing the session that as you go between different objects, I don't randomly create new sessions. The other thing I can look at here is what happens if I start a new session before I cleanly commit. And so in this last example, I'm going to open up a new session here, I'm going to start a new transaction maybe some work is happening inside there and then I'm going to commit it, but notice I'm working in a new session from the one I was working in before. So actually before I do that, I'm sorry, let me show you. In the database I have some values here. So I have 1 and 2, so when I run this again I'm expecting to get number three.   
*The presenter focuses on the following output in the Console:  
  
Welcome to Hibernate! - Tue Feb 25 11:44:46 PST 2014  
  
Page 1   
  
I am the fist page  
  
Page 2   
  
I am the second page  
  
Test Complete  
  
He opens the Command Prompt - mysql -u root -p. It displays the output:  
  
Query OK, 1 row affected <0.00 sec>  
Query OK, 1 row affected, 1 warning <0.00 sec>Query OK, 1 row affected <0.00 sec>  
Query OK, 1 row affected, 1 warning <0.00 sec>Query OK, 1 row affected <0.00 sec>  
Query OK, 1 row affected, 1 warning <0.00 sec>Query OK, 1 row affected <0.00 sec>  
Query OK, 1 row affected, 1 warning <0.00 sec>Query OK, 1 row affected <0.00 sec>  
Query OK, 1 row affected, 1 warning <0.00 sec>Query OK, 1 row affected <0.00 sec>  
Query OK, 1 row affected, 1 warning <0.00 sec>  
  
mysql> select \* from story;  
Empty set <0.00sec>  
  
A table is also displayed with five columns and two rows. The column headings are, id, title, storyDate, creationDate, and LastModifiedDate. The first row is:  
  
id: 1.  
title: Welcome to Hibernate.  
storyDate: 2014-02-25.  
creationDate: 2014-02-25, 00:00:00.  
LastModifiedDate: 2014-02-25 12:24:52.*   
  
So I run it again, and it goes through, it inserts an object, it even loads the object back out for me, it does this like the loads the object back out for me because it's in the Hibernate cache. It's even attempting to put in the database, it just hasn't committed it yet and when the application closes, it rolls it back and setup commits it. And since I committed the wrong session, you can see there's no number three inside of here, it's just not there. If I go through and I fix that solution right here, I get rid of my extraneous session being opened and committed, and I run this then the test case will run fine as it did before, but it will then allow me to see the objects opening in my database, now it's number 4. The reason it's number 4 is because it actually did successfully ask for the number 3, the database committed that, it just didn't commit the save of the other one. So these are actual issues that I spent more than a reasonable amount of time on while building the materials to talk about it, in messing up, and then trying to figure out and hunt this down. So the key around this is to come up with a great design around session management. Use data access objects, use either a framework or a very structured way of knowing when to begin and end commits and sessions, and all that sort of structure to your Hibernate program and you'll be much more successful and hey, if your data is not persisting the database you probably have a bad session somewhere.   
*The presenter clicks Run and the output is displayed in the Console. The presenter then focuses on a section of the output:  
  
Welcome to Hibernate! - Tue Feb 25 11:44:46 PST 2014  
  
Page 1   
  
I am the fist page  
  
Page 2   
  
I am the second page  
  
Test Complete  
  
The presenter opens the Command Prompt and runs the command:  
  
select \* from story;   
  
A table is displayed with five columns and two rows. The column headings are, id, title, storyDate, creationDate, and LastModifiedDate. The first row is:  
  
id: 1.  
title: Welcome to Hibernate.  
storyDate: 2014-02-25.  
creationDate: 2014-02-25, 00:00:00.  
LastModifiedDate: 2014-02-25 12:24:52.  
  
In Java, the presenter amends the code lines:  
  
Session s =, factory.openSession();, S.beginTransaction()";, and  
s.getTransacion().commit();  
  
The new code is now:  
  
          // Session s = factory.openSession();   
          // S.beginTransaction()";  
         //s.getTransacion().commit();  
  
He clicks Run and focuses on the output:  
  
Welcome to Hibernate! - Tue Feb 25 11:44:46 PST 2014  
  
Page 1   
  
I am the fist page  
  
Page 2   
  
I am the second page  
  
Test Complete  
  
The presenter opens the Command Prompt and runs the command:  
  
select \* from story;  
  
A table is displayed with five columns and 3 rows. The column headings are, id, title, storyDate, creationDate, and LastModifiedDate. The first row is:  
  
id: 1.  
title: Welcome to Hibernate.  
storyDate: 2014-02-25.  
creationDate: 2014-02-25, 00:00:00.  
LastModifiedDate: 2014-02-25 12:24:52.*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a01_it_enus/output/html/course_transcript.html#top)

Batch Processing

Learning Objective

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

* *use the executeUpdate command as part of a query to do batch and bulk updates*

**1. Batch processing and bulk updates**

Given that Hibernate focuses much of its mapping intention around individual objects and IDs, you might get the impression that it doesn't work as well when I'm dealing with the lot of objects, as I would in batch processing. That's not really entirely true. Hibernate has the ability to do much more detailed operations. So one of the test classes I have created here, actually does batch operations all the time. I've been doing that in a lot of the tests I've created for Hibernate. And so here's a quick method that takes and generates a hundred different objects, and it wants to put them all on the database. And so if I want to be clever about that, what I want to do is not try and insert all of them at once. Because if I have a file with a million rows in it and try and put them on at once, I'm probably going to have too much data in my Java object...in my Java classes...I'm going to have too much heap memory going on to be able to manage it properly. So instead, what I want to do in Hibernate is as data is coming in from a source, I want to save it regularly and flush it regularly. And so what I'm doing is I'm creating a batch size of 10 or whatever is going on inside of there, and then I'm actually configuring in my hibernate.config to match up to that batch size. So we can actually see my batch size and my hibernate.config is also set to be 10.   
*Java Eclipse is open on the BatchMain.java tab. The code is displayed in the editor:  
  
factory.getCurrentSession().beginTransaction();  
  
           bulk.createABunchForSearching();  
System.err.println("!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!");  
           Date archiveBefore = new StoryGenerator().getRandomDate(new Date());   
           System.err.println("Archive anything before " + archiveBefore);  
           System.err.println("Archive count: " + batch.archiveStories(archiveBefore));  
  
System.err.println("!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!");  
           System.err.println("Update count: " + batch.updateModified());  
  
System.err.println("!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!");  
           System.err.println("Delete count: " + batch.deleteArchived());  
             
           System.err.println("Test Complete");  
           System.err.println("!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!");  
           factory.getCurrentSession().getTransaction().commit();  
       } catch (Throwable t)  
       {  
           t.printStackTrace();  
           factory.getCurrentSession().getTransaction().rollback();  
       } finally  
       {  
  
The presenter selects the Bulkinsert.java tab and the code is displayed in the editor:   
  
public void createABunchForSearching()  
   {  
       StoryGenerator sg = new StoryGenerator();  
       List<Story> stories = sg.generate(100);  
         
       int batchSizeFromConfig = 10;  
       int count = 1;  
       for (Story s : stories)   
       {  
           session.save(s);  
             
           // After every set of batches, we want to flush forcing the save and clearing the session  
           if (count++ % batchSizeFromConfig == 0)   
           {  
               session.flush();  
               session.clear();  
           }  
       }  
            
   }  
}  
  
The presenter selects the hibernate.cfg.xml tab and the code is displayed in the editor:  
  
<!-- Enable Hibernate's automatic session context management -->  
       <property name="current\_session\_context\_class">thread</property>  
  
       <!-- Disable the second-level cache  -->  
       <property name="cache.provider\_class">org.hibernate.cache.internal.NoCacheProvider</property>  
  
       <!-- Echo all executed SQL to stdout -->  
       <property name="show\_sql">true</property>  
  
       <!-- Enables adding of comments to SQL when generated -->  
       <property name="hibernate.use\_sql\_comments">true</property>  
  
       <!-- Required to sync the Hibernate schema to your database, else it will break! -->  
<!--         <property name="hbm2ddl.auto">create</property> -->  
  
       <!-- Used to save occasionally when doing large inserts of data -->  
       <property name="hibernate.jdbc.batch\_size">10</property>  
  
       <mapping class="hibernate.domain.Story"/>  
       <mapping class="hibernate.domain.Page"/>  
       <mapping class="hibernate.domain.StoryCustomStatements"/>  
       <mapping class="hibernate.domain.StoryArchive"/>  
<mapping class="hibernate.domain.StoryBrief"/>  
       <mapping class="hibernate.domain.StoryDescription"/>  
       <mapping class="hibernate.domain.PublicStory"/>  
       <mapping class="hibernate.domain.Newspaper"/>*   
  
And so as I'm doing the inserts, every time I've inserted 10 additional items, every time I hit save on 10 additional items, I'm going to go through and I'm going to flush my session and clear my session, which says to Hibernate, "Put those in the database and then forget about them." So I don't cache them, I don't keep them around, and if again, I'm inserting a million objects, I don't end up with a million objects floating around in memory, slowing down my application and potentially crashing it. Now well beyond that, one aspect of it would be inserting a bunch of objects, but Hibernate can also be used to do bulk insert update and delete operations. So I have another little helper class inside of here that's going to go through and provide some batch operations. So for instance, I have a bunch of stories on my database now, and I want to archive them and don't have to be storage, it could be any data. And so what I want to do is I want to insert to one table based off of selecting from another table. So I have a Story table that's mapped out there for me. And I'm going to take those stories and then I'm going to archive them into a StoryArchive object. Now the first key here is I have to have mapped both Story and StoryArchive. I have to have two classes out there and they both have to be available in my HQL to make this happen.   
*The presenter selects the Bulkinsert.java tab and then selects the BatchMain.java tab. Then he selects the BatchOperations.java tab and the code is displayed in the Editor:  
  
Private Session session;  
     
   public BatchOperations(Session session)  
   {  
       this.session = session;  
   }  
     
   public int archiveStories(Date priorTo)  
   {  
       String hqlInsert = "insert into StoryArchive (id, title, storyDate, creationDate) "  
                          + "select s.id, s.title, s.storyDate, s.creationDate from Story s where s.storyDate < :prior";  
       Query q = session.createQuery(hqlInsert);  
       q.setDate("prior", priorTo);  
       int createdEntities = q.executeUpdate();  
       return createdEntities;  
   }  
  
   public int deleteArchived()  
   {  
       String hqlDelete = "delete Page where storyId in (select id from StoryArchive)";  
       Query q = session.createQuery(hqlDelete);  
       int deletedEntities = q.executeUpdate();  
       return deletedEntities;  
   }  
     
   public int updateModified()  
   {  
       String hqlInsert = "update StoryArchive set lastModifiedDate = CURRENT\_TIMESTAMP";  
       Query q = session.createQuery(hqlInsert);  
       int updatedEntities = q.executeUpdate();  
       return updatedEntities;  
   }*   
  
But passed that, you can see I have a query that selects data from stories: id, title, storyDate, creationDate. And it's going to insert that into my hqlInsert statement. This is not SQL this is HQL, because notice what's happening here's I'm not putting values in. It's just automatically populating it based off of the results of the first query, and then that's based off of a parameter that's going into my insert. And so as I do my query, I insert a date so I'm going to archive everything priorTo and move them over into my table. At that point, I also have a method to delete from the original pages any stories that were archived. And so I don't want to keep those pages around if I've archived them. They're coming out of that, I can go delete them. So I can do a bulk delete here and I can do a bulk page delete as well as a separate item. But I can do a bulk delete of the pages, where the storyId is in anything that was selected out of the storyArchive. So after I've archived them, I can go through and get everything I've archived and delete those out of the page. And so I'm doing a quick join there, I can do the same thing for Story, I'm just not showing it here.   
  
And then the last item here, as I'm going to all of my archived stories, and I'm setting the lastModifiedDate to be the CURRENT\_TIMESTAMP. So anything gets archived, I'm going to modify as it being right now when I've modified it, and gone from there so I know everything was touched. So here's the example of a, again an insert, an update, and a delete on bulk objects from HQL. So let's go ahead and execute our sample right here. And so in our sample we use our bulk insert test file that insert a bunch of objects, you can see they're all getting inserted, and they're being batched out and flushed every 10 inserts. And then once that's all done, I'm going to archive anything that happened before this date that's shown inside of there. And so I've archived 54 objects. Notice in my batch operation when I do my Query, I do an executeUpdate and I get back the number of objects that were modified. And so I'm go to then go through and I'm going to archive my Story. I'm going to update that and I'm going to be setting the CURRENT\_TIMESTAMP on 54 of those, the same number that I did before and then I've deleted a 109 pages that was associated with those old stories that are out there.   
*The presenter selects the BatchMain.java tab and clicks Run. The output is displayed in the Console. The presenter focuses on the code:  
  
Archive before Web Feb 19 11:58:22 PST 2014  
Hibernate: insert into StoryArchive ( id, title, storyDate, creationDate ) select story0\_.id ad col\_0\_0\_, story0\_.title as  
Archive count: 54  
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!  
Hibernate: update StoryArchive set lastModifiedDate=Current\_TIMESTAMP  
Update count:54  
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!  
Hibernate: delete from page where storyId in (select storyarchi1\_.id from StoryArchive storyarch1\_)  
Delete count: 109  
Test Complete  
  
  
The presenter selects the BatchOperations.java tab.*   
  
And so if I go to my database I can look at my storyArchive and you can see it's all been updated, and the lastModifiedDate has been updated to the time I ran on this, they're all the same. There's no variation inside of there. This is all done based off of that batch stamp that's happening there. So just because I'm doing Hibernate I don't have to sacrifice the ability to do batch updates. I don't have to do any special fancy stuff. I can actually use the mappings and HQL to execute this by creating queries, and using executeUpdate as part of the query to be able to make batch and bulk updates.   
*The presenter opens the Command Prompt -mysql -u root -p. A table is displayed with five columns and 100 rows. The column heading names are, id, title, storyDate, creationDate, and lastModifiedDate.*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a01_it_enus/output/html/course_transcript.html#top)

Hibernate and Logging

Learning Objective

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

* *use the log4j framework to configure the logging of information*

**1. Using the log4j framework**

Logging is a key part of all applications, to be able to do some debugging to keep track of the execution of flow, and basically analyze the performance of an application as it runs. So Hibernate logging is very key as it has such a big impact on the interactions with the database. Hibernate, by default, uses a framework called simple log4j and if you don't do anything else, Hibernate will log out to the Console. Simple log4j has a lot of config options and most commonly plugs into log4j itself, so simple log4j, simple API and log4j complex API. And so in order to setup the full log4j inside of Hibernate. I first need to start with getting the right jar files. And so beyond this simple log4j API that comes inside of Hibernate, I can apply the API and the log4j file and the actual log4j implementation. So this log4j is binding over to this actual log4j implementation, and just throwing those in the class path takes care of 95% of everything I need to have happen. The last 5% is configuring log4j with our log4j.properties. And my log4j is very simply just putting it out to standard out. Log4j has the ability to create files and do all sorts of controls on those files. Well outside the scope of what we're going to talk about here, but we're just simply putting it out to a Console.   
*Java Eclipse is open on the hibernate.cfg.xml tab. The code that displays in the editor is:  
  
 <!-- Disable the second-level cache  -->  
       <property name="cache.provider\_class">org.hibernate.cache.internal.NoCacheProvider</property>  
  
       <!-- Echo all executed SQL to stdout -->  
       <property name="show\_sql">true</property>  
  
       <!-- Enables adding of comments to SQL when generated -->  
       <property name="hibernate.use\_sql\_comments">true</property>  
  
       <!-- Required to sync the Hibernate schema to your database, else it will break! -->  
<!--         <property name="hbm2ddl.auto">create</property> -->  
  
       <!-- Used to save occasionally when doing large inserts of data -->  
       <property name="hibernate.jdbc.batch\_size">10</property>  
  
The presenter then navigates to the SL4J library, which is expanded in the Package Explorer. It includes the files: log4j-1.2.17.jar - zzzLibraries/lib/sl4j, slf4j-api-1.7.6.jar - zzzLibraries/lib/sl4j, and slf4j-log4j12 -1.7.6.jar.  
  
The presenter navigates to the log4j.properties tab and the code is displayed:   
  
Direct log messages to stdout  
log4j.appender.stdout=org.apache.log4j.ConsoleAppender  
log4j.appender.stdout.Target=System.out  
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout  
log4j.appender.stdout.layout.ConversionPattern=%d{ABSOLUTE} %5p %c{1}:%L - %m%n  
  
# Root logger option  
log4j.rootLogger=INFO, stdout  
  
# Log everything. Good for troubleshooting  
#log4j.logger.org.hibernate=INFO  
#log4j.logger.org.hibernate.SQL=trace  
#log4j.logger.org.hibernate.engine.query=trace  
log4j.logger.org.hibernate.type=trace  
#log4j.logger.org.hibernate.jdbc=trace*   
  
So our basic logging option inside of hibernate config, really the only good logging out option is whether or not we're going to show our SQL. So I'm going to go run an app here that does a little bit of querying, to show you that my SQL is turned on and it logs the basic SQL as it comes out here. You can see all the selects that's happening to be able to load up data. Now if I go back to that config and I say, I don't want to show the SQL, it basically reduces the amount of stuff that gets spit out as I go and run my application. Now whether that's good or bad is up to you. It's certainly less messages here than it was before, but it also is less information about what's happening. So for the most part in development, you run with this turned on, and perhaps even as you get into the actual real environment, you keep it turned on, because this querying is pretty innocuous. Where the querying becomes dangerous is if it would show actual user data, which this doesn't. You can see it's all question marks inside of here, it doesn't actually show user data. The problem with that is, is how do I debug it? How do I know which parameters are being plugged in? How do I know why my query is failing because I don't see the parameters? That cannot be controlled through hibernate config but it has to be controlled through log4j. So there's a bunch of different types, they're all commented out right now. But here's one example of the Hibernate type.   
*The presenter selects the hibernate.cfg.xm tab and then selects the QueryingMain.java tab. The following code is displayed in the Editor:  
  
package hibernate;  
  
import hibernate.batch.BulkInsert;  
  
import java.util.List;  
  
import org.hibernate.Query;  
import org.hibernate.Session;  
import org.hibernate.SessionFactory;  
  
public class QueryingMain  
{  
   private Session session;  
     
   public static void main(String[] args)  
   {  
       new QueryingMain().run();  
   }  
  
   private void run()  
  
The presenter clicks Run and the output is displayed in the Console. He focuses on a section of the output:  
  
Hibernate: select pages0\_.storyId as storyId7\_3\_0\_, pages0\_.id as id1\_2\_0, pages0\_.id as id1\_2\_1, pages)\_.creationDate....  
  
The presenter returns to the hibernate.cfg.xml tab and amends the codeline:  
  
<!-- Echo all executed SQL to stdout -->        <property name="show\_sql">true</property>  
  
The new code is now:  
  
<!-- Echo all executed SQL to stdout -->  
<property name="show\_sql">false</property>.  
  
The presenter selects the QueryMain.java tab and clicks Run. The output is displayed in the Console.   
  
The presenter navigates to tehhibernate.cfg.xml tab and changes the codeline:  
  
<!-- Echo all executed SQL to stdout --> <property name="show\_sql">false</property>  
  
The code line is now:  
  
<!-- Echo all executed SQL to stdout -->  
       <property name="show\_sql">true</property>.  
  
The presenter clicks Run and the output is displayed in the Console. The presenter focuses on a section of the output:  
  
Hibernate: select pages0\_.storyId as storyId7\_3\_0\_, pages0\_.id as id1\_2\_0, pages0\_.id as id1\_2\_1, pages)\_.creationDateas pageNumb5\_2\_1\_, pages0\_.storyId as storyId 2\_1, pages0\_.words as id6\_2\_1 from pages0\_where pages0\_story.Id=?.  
  
The presenter selects the log4j.properties tab and the code is displayed in the editor:  
  
# Direct log messages to stdout  
log4j.appender.stdout=org.apache.log4j.ConsoleAppender  
log4j.appender.stdout.Target=System.out  
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout  
log4j.appender.stdout.layout.ConversionPattern=%d{ABSOLUTE} %5p %c{1}:%L - %m%n  
  
# Root logger option  
log4j.rootLogger=INFO, stdout  
  
# Log everything. Good for troubleshooting  
#log4j.logger.org.hibernate=INFO  
#log4j.logger.org.hibernate.SQL=trace  
#log4j.logger.org.hibernate.engine.query=trace  
#log4j.logger.org.hibernate.type=trace  
#log4j.logger.org.hibernate.jdbc=trace*   
  
And the Hibernate type, what if I uncomment this and I save this? When I go and run that same application over again, it runs the application this time, where it shows me which values for this query that's going to go out. It shows me which values it's binding into the parameters. And so you can see as data comes in and as data goes out, it's going to bind. Here's the ID, here's the timestamp, here's the Boolean, here's the timestamp. All this page information that's coming in back from the query is getting bound inside of there, and Hibernate will show you that level of detail. This level of allowing can be dangerous, if you have credit card information or social security numbers. It'll be inappropriate to have this turned on in a production-like environment, at least for very long, but it does have the capability as you're doing testing and debugging to turn that on. Again there's a lot of different options here. I can turn on all of these options if I want and see the net result of this, when I go and run my application, as I have this massive output of log information. I'm going to actually make the screen really, really big so you can see it.   
*The presenter clicks Run and the output is displayed in the editor. The presenter then focuses on the following output:  
  
11:48:05, 594 TRACE BasicExtractor:78 - extracted value (is1\_2\_1] : (BIGINT]) - [16]   
11:48:05, 594 TRACE BasicExtractor:78 - extracted value ([creation2\_2\_1\_] : [TIMESTAMP]) - [2014-02-20 11:48:04.0]  
11:48:05, 594 TRACE BasicExtractor:78 - extracted value ([htmlIncl3\_2\_1\_] : [BOOLEAN] - [false]   
11:48:05, 594 TRACE BasicExtractor:78 - extracted value([lastModi4\_2\_1\_] : [TIMESTAMP]) - [2014-02-20 11:48:04.0]  
  
The presenter amends the following code lines by removing the # :  
  
# Log everything. Good for troubleshooting  
#log4j.logger.org.hibernate=INFO  
#log4j.logger.org.hibernate.SQL=trace  
#log4j.logger.org.hibernate.engine.query=trace  
#log4j.logger.org.hibernate.type=trace  
#log4j.logger.org.hibernate.jdbc=trace  
  
The code is now:   
  
# Log everything. Good for troubleshooting  
log4j.logger.org.hibernate=INFO  
log4j.logger.org.hibernate.SQL=trace  
log4j.logger.org.hibernate.engine.query=trace  
log4j.logger.org.hibernate.type=trace  
log4j.logger.org.hibernate.jdbc=trace  
  
He clicks Run and the output is displayed in the Console. The presenter expands the Console screen to full page. An example of the output is:   
11:48:05, 594 TRACE BasicExtractor:78 - extracted value (is1\_2\_1] : (BIGINT]) - [16].*   
  
And you can see all this stuff that gets put out, all of the details along all of the queries, all of the data going in and out of the queries. It has cache information. You can see all the levels of internal operations that's happening inside of there. You can really follow through the Hibernate code step-by-step at this point, to be able to do all the debugging you want to and it's so much, it actually is run by the buffer of my Console. There's more there than I'm even showing because, my Consoleis buffered, it doesn't show everything inside of there. So again you can customize this, whichever you want to. If you don't care about the query engine you can shut that down, you don't want to see the data you can shut that down. You can manage this however you want to, and there's all sorts of levels of this you can turn on and off. So Hibernate is going to give you a lot of details, towards being able to manage the appropriate amount of logging information, and turn it on and off at different levels. Again these were all trace, you can put it in info, you can put it in a debug, but you can get all or as little of information, and redirect it as carefully as you want to given all the config options in Hibernate, as well as the logging frameworks you're working with.   
*The presenter amends the following code by re-adding the #:  
  
log4j.logger.org.hibernate=INFO  
log4j.logger.org.hibernate.SQL=trace  
log4j.logger.org.hibernate.engine.query=trace  
log4j.logger.org.hibernate.type=trace  
log4j.logger.org.hibernate.jdbc=trace  
  
The full code is now:  
  
# Log everything. Good for troubleshooting  
#log4j.logger.org.hibernate=INFO  
#log4j.logger.org.hibernate.SQL=trace  
#log4j.logger.org.hibernate.engine.query=trace  
#log4j.logger.org.hibernate.type=trace  
#log4j.logger.org.hibernate.jdbc=trace*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a01_it_enus/output/html/course_transcript.html#top)

Caching Data

Learning Objective

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

* *use second level caching in Java Hibernate*

**1. Using second level caching**

Hitting the database repeatedly for data can be an expensive and time-consuming operation. Caching data provides a trade-off of local memory use versus database hits. Hibernate allows a second level cache through external providers and many levels of configuration. So let's see how this could help optimize our solution. Hibernate has the ability to add in to all of the other great stuff it provides, a second level caching of data. Now everything is cached at a first level as part of the session. So when you hear the term second level caching, what it's referring to is, using an additional plug-in tool to store data in your environment's memory. So if you have a big server, you can store data in it's huge amount of RAM, or on a file, or something there locally that's going to allow you to reduce the number of queries, and the amount of updates I have to do to the database. And so setting up the cache is a little bit involved, and this has several steps but not that difficult. The first thing is you need to include in, in this case, I've set up a library that points to my caching solution. So I'm using EHCache, which is a very popular default for using with Hibernate, and Hibernate has a plug-in to EHCache to be able to work very closely with this tool. Depending on the version of Hibernate you're using, this plug-in will change a little bit. For the most recent version, this is the two simple jar files you need to plug into your application.   
*Java Eclipse is open on the hibernate.cfg.xml tab. It contains the code:  
  
<?xml version='1.0' encoding='utf-8'?>  
<!DOCTYPE hibernate-configuration PUBLIC  
       "-//Hibernate/Hibernate Configuration DTD 3.0//EN"  
       "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">  
  
<hibernate-configuration>  
  
   <session-factory>  
       <!-- Database connection settings -->  
       <property name="connection.driver\_class">com.mysql.jdbc.Driver</property>  
       <property name="connection.url">jdbc:mysql://localhost:3306/BigNews</property>  
       <property name="connection.username">root</property>  
       <property name="connection.password">test</property>  
  
       <!-- JDBC connection pool (use the built-in) -->  
       <property name="connection.pool\_size">1</property>  
  
       <!-- SQL dialect -->  
       <property name="dialect">org.hibernate.dialect.MySQLInnoDBDialect</property>  
  
The presenter navigates to the Package Explorer and expands EHCache. It contains the files: ehcache-core-2.4.3.jar and hibernate-ehcache-4.3.1.Final.jar.*  
  
Now past that, in my hibernate.config file I've got to turn on caching. So I got my database set up, all that sort of the stuff is the same, but I need to first of all enable the second level caching. Second of all, I need to tell it what solution I'm going to use to plug into caching, and again from that jar file I'm using the Hibernate plug-in to ehcache, and then as a third option I can turn on query caching. And query caching is going to return the results of my queries and cache those internally. So once I've got those set up, then I just simply need to configure my caching solution, and this is very specific to the caching solution, not Hibernate. And so I won't go into much detail here, but for ehcache I have this real simple file that sets up a general place to cache stuff. I'm choosing different options here, in this case, I am choosing a disk...a temporary diskStore. It'll go away after my program is done running, but it's a temporary diskStore in which I'm going to cache this information. So once I've set up the jar dependency, enabled my hibernate.config file, and set up my caching config, I'm ready to run.   
*The presenter scrolls down to reveal additional code:  
  
       <!-- Enable Hibernate's automatic session context management -->  
       <property name="current\_session\_context\_class">thread</property>  
  
       <!-- Enable the second-level cache  -->  
<property name="hibernate.cache.use\_second\_level\_cache">true</property>  
       <property name="hibernate.cache.region.factory\_class">org.hibernate.cache.ehcache.EhCacheRegionFactory</property>  
 <property name="hibernate.cache.use\_query\_cache">true</property>  
  
       <!-- Echo all executed SQL to stdout -->  
       <property name="show\_sql">true</property>  
  
       <!-- Required to sync the Hibernate schema to your database, else it will break! -->  
       <property name="hbm2ddl.auto">update</property>  
  
       <!-- Used to save occasionally when doing large inserts of data -->  
       <property name="hibernate.jdbc.batch\_size">10</property>  
  
       <mapping class="hibernate.domain.Story"/>  
       <mapping class="hibernate.domain.Page"/>  
   </session-factory>  
</hibernate-configuration>  
  
The presenter selects the ehcache.xml tab. Te code is displayed in the editor:  
  
<?xml version="1.0" encoding="UTF-8"?>  
  
<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
   xsi:noNamespaceSchemaLocation="http://ehcache.org/ehcache.xsd" updateCheck="false">  
  
   <diskStore path="java.io.tmpdir/tempCache" />  
  
   <defaultCache eternal="false" maxElementsInMemory="1000"  
       overflowToDisk="false" diskPersistent="false" timeToIdleSeconds="0"  
       timeToLiveSeconds="600" memoryStoreEvictionPolicy="LRU" statistics="true" />  
</ehcache>*   
  
So I have a really simple demo here. I have a demo that's going to load up a bunch of objects, that's going to do a query on a bunch of objects. And then I'm going to do the exact same query a second time, and so we're going to see what happens in the caching environment. We know, and traditionally what would happen, is I hit the database, load it, go back to the database, load it again. So now when I execute this in my cache world, there's going to be a lot of output here. But I scroll down in my demo and there's so much being logged. I have debugging turned up pretty high in the cache so we can see that, but here now we're at the spot where we're doing a query. So we're running the query the first time. And you can see all these different selects that are happening to load up all of the stories. And so there's 1, 2, 3, 4, 5, 6, 7 stories that are being loaded and all the pages that are associated with the stories that are being loaded into our solution. Now the second time I'm going to go run it, you can see I'm running it again, notice there are zero lines of code after that. I am not going to Hibernate, I am not making a query, all that data was just loaded in my solution. Not only was that data loaded in the solution, the exact query with the exact same results was loaded in the solution.   
*The presenter selects the CacheMain.java tab and the code is displayed in the editor:  
  
 System.err.println("!!!!!!!!!!!!!!!!!!!!!!!!!!!!!");  
           System.err.println("Run it once");  
           {  
               Criteria c = session.createCriteria(Story.class);  
               c.add(Restrictions.like("title", "Bamboo", MatchMode.END));  
               c.setCacheable(true);  
               c.setCacheRegion("Story");  
               List<Story> all = c.list();  
               StoryToString.printAllStories(all);  
           }  
  
  
           System.err.println("!!!!!!!!!!!!!!!!!!!!!!!!!!!!!");  
           System.err.println("Run it again");  
           {  
               Criteria c = session.createCriteria(Story.class);  
               c.add(Restrictions.like("title", "Bamboo", MatchMode.END));  
               c.setCacheable(true);  
c.setCacheRegion("Story");  
               List<Story> all = c.list();  
               StoryToString.printAllStories(all);  
           }  
  
The presenter clicks Run and the output is displayed in the Console. He focuses on a section of the output:   
  
Hibernate: select pages0\_.storyId as storyId7\_1\_0\_, pages0\_.id as id1\_0\_0, pages0\_.id as id1\_0\_1, pages0\_.creationDate as...  
Hibernate: select pages0\_.storyId as storyId7\_1\_0\_, pages0\_.id as id1\_0\_0, pages0\_.id as id1\_0\_1, pages0\_.creationDate as...  
Hibernate: select pages0\_.storyId as storyId7\_1\_0\_, pages0\_.id as id1\_0\_0, pages0\_.id as id1\_0\_1, pages0\_.creationDate as...  
Hibernate: select pages0\_.storyId as storyId7\_1\_0\_, pages0\_.id as id1\_0\_0, pages0\_.id as id1\_0\_1, pages0\_.creationDate as...  
Hibernate: select pages0\_.storyId as storyId7\_1\_0\_, pages0\_.id as id1\_0\_0, pages0\_.id as id1\_0\_1, pages0\_.creationDate as...  
Hibernate: select pages0\_.storyId as storyId7\_1\_0\_, pages0\_.id as id1\_0\_0, pages0\_.id as id1\_0\_1, pages0\_.creationDate as...  
  
The presenter then scrolls through the output and then focuses on the code:   
  
Run it again.*   
  
So I can return those objects immediately and without having to hit the database, which dramatically improves my performance. And so looking back at my solution here, you can see I run it once, I run it again, and it works just that quickly. Now caching can be configured in a lot of different levels. I can set up caching for the entire project more likely because my cache would be filled up in a hurry. I will enable caching at the individual class level. And so my Story class in this case, was set up to be cacheable. And beyond that I can choose to set up a caching specific region for this set of objects. A region in caching is just saying, "hey, this region holds specific types of objects." Not the general everything else, it just holds this, which gives you certain amount of optimization for those objects even beyond that. So I can set up a region just to hold my story objects. That region setup is part of the caching config, but then I can point to it very easily inside of my annotations, as you can see here. So I can cache based off of a whole class. I can cache off of a field that's coming back in a collection, a list, or set. I can cache on a specific query and I can do that either in annotations or in XML config. Doesn't matter, those are just details at this point, but the cool thing is that very simple three-step plug-in. I can enable that big performance-enhancing capability, of using a second level cache within Hibernate.   
*The presenter minimizes the Console window. He selects the Story.java tab and the code is displayed in the editor:  
  
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;  
     
   @Basic  
   private String title;  
     
   @Basic  
   private Date   storyDate;  
     
   @Temporal(TemporalType.DATE)  
   private Date   creationDate;  
     
   @Temporal(TemporalType.TIMESTAMP)  
   private Date   lastModifiedDate;*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a01_it_enus/output/html/course_transcript.html#top)

Unique Item Query

Learning Objective

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

* *use the unique item query in Java Hibernate*

**1. Using the unique item query**

Well it's great that we can sometimes search on the ID, the primary key. Often we have other search criterias that aren't searching the primary key, but still expect to come up with a unique result, the business rules might dictate that. So I've come up with a little business scenario here that's saying, "Hey, I want to search inside of our real estate table." And one of the rules of the real estate table is between the street, city, state, zip. I should have one, and only one, row that matches those. I want to make sure I have unique addresses out there. And so if I do a query on those four things, street, city, state, zip, I'm expecting a unique result. And so Hibernate gives me an alternative to the list method on queries and criteria objects, called uniqueResult. Again, available on both criterias and on query objects, HQL queries. And the uniqueResults method returns to me the unique object that matches one, and only one, of that results or nothing if nothing happens to match that results, because that's okay too. But will fail if I have more than one result. Giving me the extra backup as a developer to know I should be getting one of those out there. It gives me a little bit of the efficiency gain, but also gives me that reassurance that, hey, if there's a mistake in the database my code will not continue to use this, and further propagate the mistake. So I have a little test case here that creates some interesting scenarios.   
*Java Eclipse is open on the UniqueMain.java tab. The code that is displayed in the editor:   
  
RealEstate r3 = new RealEstate();  
           r3.setValue(4444444);  
           r3.setAddress(unique);  
           s.saveOrUpdate(r3);  
       }  
   }  
  
   private void loadUnique(Session s, Address address)  
   {  
       Query q = s.createQuery("from RealEstate where street = :street and city = :city and state = :state and zip = :zip");  
       q.setString("street", address.getStreet());  
       q.setString("city", address.getCity());  
       q.setString("state", address.getState());  
       q.setString("zip", address.getZip());  
       RealEstate uniqueItem = (RealEstate) q.uniqueResult();  
       if (uniqueItem != null)  
       {  
           System.err.println("Got a unique item!");  
       } else  
       {  
           System.err.println("Got nothing");  
       }  
   }  
     
}  
  
The presenter scrolls up the page and displays additional code:  
  
   private void run()  
   {  
       try  
       {  
           Session s = factory.getCurrentSession();   
           s.beginTransaction();  
  
           Address missing = new Address();  
           missing.setStreet("nope");  
           missing.setCity("nope");  
           missing.setState("nope");  
           missing.setZip("nope");  
  
System.err.println("!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!");  
           fillData(s);  
             
           System.err.println("!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!");  
           System.err.println("Get a unique item");  
           loadUnique(s, unique);  
  
           System.err.println("!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!");  
           System.err.println("Get a missing item");  
           loadUnique(s, missing);  
             
           System.err.println("!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!");  
           System.err.println("Get a duplicate item");  
           loadUnique(s, common);  
             
           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);  
       }  
   }  
     
   private void fillData(Session s)  
   {  
       common = new Address();  
       common.setStreet("same street");  
       common.setCity("same city");  
       common.setState("same state");  
       common.setZip("12345");  
         
       {  
           RealEstate r = new RealEstate();  
           r.setValue(123456);  
           r.setAddress(common);  
           s.saveOrUpdate(r);  
       }  
  
       {  
           RealEstate r2 = new RealEstate();  
           r2.setValue(654321);  
           r2.setAddress(common);  
           s.saveOrUpdate(r2);  
       }  
  
       {  
           unique = new Address();  
           unique.setStreet("my street");  
           unique.setCity("my city");  
           unique.setState("my state");  
           unique.setZip("54321");*   
  
So I add in the unique item and then I search on it and I should get it. I add in a query that searches on something that doesn't exist in the database, and yeah there's nothing that looks like that object, and then I have duplicate objects that are loaded in there to my database. So I'm going to search on that to see what the net results is. So let's execute that. So the net result of this is I insert the items in the database, and then I search on the unique item, which I return, I Got a unique item! And then I do a search on the missing item and as a result of that I get nothing back, I got null. And it's okay, I did a check on that and I printed out, get null. When I do a search that results in more than one though, you can see I get an exception, query did not return a unique result, and it tells me the number returns just in case. So what do I do about that? Well there's not much I can do in my solution. I really probably am not going to self correct that, but I can send an e-mail, make a log message, do something to warn some administrator, some database administrator, some business person, that the state expected my application is not what it is, and they can go off and correct that appropriately. There might be cases where you can be intelligent and fix it, but at least I can be warned, I can know what's going on. So this little extra feature that Hibernate throws in certainly gives me a better opportunity to maintain the integrity of my data, and not just accidentally make mistakes, because I picked the first one that happens to comeback, but I'm expecting only one.   
*The presenter clicks Run and the output is displayed in the Console. The presenter focuses on a section of output:  
  
Hibernate: insert into RealEstate (value, street, city, state, zip) values (?, ?, ?, ?, ?)   
Hibernate: insert into RealEstate (value, street, city, state, zip) values (?, ?, ?, ?, ?)   
Hibernate: insert into RealEstate (value, street, city, state, zip) values (?, ?, ?, ?, ?)   
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!  
Get a unique item   
Hibernate: select realestate0\_.id as id1\_3\_, realestate0\_.value as value2\_3\_, realestate0\_.street as street3\_3\_, realestate...  
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!  
Got a unique item!  
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!  
Get a missing item  
Hibernate: select realestate0\_.id as id1\_3\_, realestate0\_.value as value2\_3\_, realestate0\_.street as street3\_3\_, realestate...  
Got nothing  
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!  
Get a duplicate item  
org.hibernate.NonUniqueResultException: query did not return a unique result: 2*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a01_it_enus/output/html/course_transcript.html#top)

Defining Custom Types

Learning Objective

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

* *define custom types in Java Hibernate*

**1. Defining custom types in Hibernate**

When basic Java data types aren't enough for us, that we need to decorate that value sometimes, we might feel like we're limited within Hibernate but we're certainly not. For example, let's talk about a simple data type such as a zip code, which is normally five digits, but sometimes you can have the plusFour inside of there, so it makes it nine digits and perhaps with a dash or whatnot. But I can create, as we see here, a class to represent that zip type. It can store the zip or it can store the zip plusFour, and so I can create constructors that you get both or you get allInOne, where it's going to go through and parse out that dash, and figure out all the numbers and all that fancy stuff. The details aren't important, the point is I can create my own custom type to represent this otherwise simple String. Well going to Hibernate, in the database, it's just going to be a String, but Hibernate needs to know to use your data type instead of using the basic String. So I can create a little bridge by implementing the userType class out of Hibernate, to map from the database type to my ZipType. Now there's a series of steps I have to do in order to make that happen. So there's things like assembling and disassembling that happened, which is about putting things into the cache. Now those typically don't have much work to do, but there might be something custom you want to do inside of there.   
*Java Eclipse in open on the Zip.java tab. The code is displayed in the editor:  
  
package hibernate.customType;  
  
public class Zip  
{  
   private String zip;  
   private String plusFour;  
  
   public Zip(String zip, String plusFour)  
   {  
       this.zip = zip;  
       this.plusFour = plusFour;  
   }  
     
   public Zip(String allInOne)  
   {  
       int dashLocation = allInOne.indexOf('-');  
       if (dashLocation > 0)  
       {  
           zip = allInOne.substring(0, dashLocation);  
           plusFour = allInOne.substring(dashLocation + 1);  
       } else  
       {  
           zip = allInOne;  
       }  
   }  
  
The presenter selects the ZipType.java tab and the code is:  
  
package hibernate.customType;  
  
public class ZipType implements UserType  
{  
  
   @Override  
   public Object assemble(Serializable cached, Object owner) throws HibernateException  
   {  
       System.err.println("assemble" + cached);  
       return cached;  
   }  
  
   @Override  
   public Object deepCopy(Object o) throws HibernateException  
   {  
       Zip z = (Zip) o;  
       Zip newZ = new Zip(z.getZip(), z.getPlusFour());  
         
       return newZ;  
   }  
  
   @Override  
   public Serializable disassemble(Object o) throws HibernateException  
   {  
       System.err.println("disassemble" + o);  
       return null;*   
  
Within the cache, it's going to often do a deepCopy, which just simply is saying, "hey, give me a copy of this," like you would in a clone method, typically inside of Java. The equals method is going to be very similar to, but might be slightly different to the equals method from your class. And it gives you an opportunity to find that separately inside of here. Or you can just use the equal method from your class as you're defining. hashcode very similar; it asks whether this object isMutable or not. And then there's some getters and setters that we can put to check for nulls, as it's putting things into or taking things out of prepared statements. And so there's some work that they have to do inside of there within prepared statements. So that's the basics of setting up this bridge, from Hibernate to your specific class inside of there. Now how do you plug that in? Well in your Java class you want to use your type just like you normally would. So instead of having a string for the zip code here, I now have my ZipType, and I can use the @Type annotation here to point to my custom ZipType inside of there. And this is not the type itself, it's the mapping that this ZipType class over here, this mapping that tells me to have Hibernate point to my zip class.   
*The presenter scrolls down to reveal additional code:   
  
@Override  
   public boolean equals(Object o1, Object o2) throws HibernateException  
   {  
       if (o1 == null && o2 == null)  
       {  
           return true;  
       } else if (o1 == null || o2 == null)  
       {  
           return false;  
       }  
             
       return o1.equals(o2);  
   }  
  
   @Override  
   public int hashCode(Object o) throws HibernateException  
   {  
       return o.hashCode();  
   }  
  
   @Override  
   public boolean isMutable()  
   {  
       return true;  
   }  
  
   @Override  
   public Object nullSafeGet(ResultSet rs, String[] names, SessionImplementor si, Object owner) throws HibernateException, SQLException  
   {  
       if (names[0] == null)  
       {  
           return null;  
       }  
       String value =(String) StringType.INSTANCE.get(rs, names[0], si);  
       return new Zip(value);  
   }  
  
   @Override  
   public void nullSafeSet(PreparedStatement ps, Object value, int index, SessionImplementor si) throws HibernateException, SQLException  
   {  
       if (value==null)   
       {  
           ps.setNull(index, Types.VARCHAR);  
           return;  
       }  
              
       ps.setString(index, value.toString());          
   }  
  
   public String getAsSingle()  
   {  
       if (plusFour != null)  
       {  
           return zip + "-" + plusFour;  
       }  
       return zip;  
   }  
     
   public String getZip()  
   {  
       return zip;  
   }  
  
   public void setZip(String zip)  
   {  
       this.zip = zip;  
   }  
  
   public String getPlusFour()  
   {  
       return plusFour;  
   }  
  
   public void setPlusFour(String plusFour)  
   {  
       this.plusFour = plusFour;  
   }  
     
   @Override  
   public String toString()  
   {  
       return getAsSingle();  
   }  
}  
  
He selects the ComplexPersonWithZip.java tab and the code is displayed:   
  
package hibernate.domain.customType;  
  
import hibernate.customType.Zip;  
  
import org.hibernate.annotations.Type;  
  
@Entity  
@Table(name="ComplexPerson")  
public class ComplexPersonWithZip  
{  
   @Id  
   @Column(name="remoteId")  
   private long id;  
     
   @Basic  
   private String name;  
     
   @Basic  
   @Column(name="ssn")  
   private String ssn;  
     
   @Type(type="hibernate.customType.ZipType")  
   private Zip zip;  
     
   public long getId()*   
  
So I can go through and run a quick example of this guy here. And this solution will go through and it's going to create an instance of a complex ZipType, using the Hibernate tools as it normally would, nothing special inside of there. The Hibernate behind the scenes is making calls into my mapping, my custom user type, to be able to move those things back and forth. And so I have a little bit of some print line to show inside of here, where I'm pulling out the last four digits of that solution. So if I look at this complex solution in the database, I have a zip code as you would expect it, it's essentially 10 digits, nine of those are numbers and one of them is a dash. And it just shows up as a String. Inside of Java though, I'm representing that data as the ZipType, and it's shown as the zip plusFour and that's where my message is coming out right here, where I can go to my zip and say get zip plusFour as I load that object back in.   
*The presenter selects the CustomtypeMain.java tab and the code displays in the Editor:   
  
private void run()  
   {  
       try  
       {  
           factory = ConfigHelper.getSessionFactory();  
           Session session = factory.getCurrentSession();  
           session.beginTransaction();  
  
           ComplexPersonWithZip newC = createComplexPerson();  
           session.persist(newC);  
           session.getTransaction().commit();  
           session = factory.getCurrentSession();  
           session.beginTransaction();  
             
           session.evict(newC);  
             
The presenter clicks Run and the output is displayed in the Console. The presenter focuses on a section of the output:  
  
6789  
  
The presenter opens up a Command prompt and runs the command:  
  
selects \* from complexperson.  
  
The output is a table with four columns and one row:  
  
remoteId: 1.  
ssn: 12345671.  
zip: 12345-6789.  
name: CP 1.  
  
The presenter moves back to Eclipse, where the CustomTypeMain.java tab is selected.*   
  
So this custom type allows me to treat data however it's represented in the database, as a special type of Java object and be able to have methods and manipulations on that data, as I see fit. I'm not limited to basic strings, or dates, or numbers inside of the database, I can represent them more interestingly. We saw this with an annotation, just as a quick note. I can do the exact same thing within XML mapping on any property, by just specifying the type attribute. And so I can point to that same ZipType I defined in my annotation, with hibernate.customtype.zipType as being the type of this object. So Hibernate gives you a massive amount of flexibility, not only in mapping database options, but how do I represent that data on the Java side as Java domain objects as well.   
*The presenter selects the ComplexPersonWithZip.hbm.xml tab and the code is displayed in the editor:  
  
<?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.customType" >  
   <class name="ComplexPersonWithZip" table="complexPerson">  
       <id name="id" column="remoteId"/>  
       <property name="name"/>  
       <property name="ssn" index="xml\_person\_ssn\_index"/>  
       <property name="zip" type="hibernate.customType.ZipType"/>  
   </class>  
</hibernate-mapping>*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a01_it_enus/output/html/course_transcript.html#top)

Filtering Data Results

Learning Objective

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

* *filter data in Java Hibernate*

**1. Filtering data in Java Hibernate**

Sometimes our business rules dictate that we don't want to show all of the data that we have in the database in certain circumstances. Perhaps there's historical data we don't want to show, or certain data's protected because of permissions. And one slip in one query could mean I'm not showing something, I am showing something that I shouldn't be showing. So Hibernate gives us the option to filter data that's coming back. And that filter can be applied as we see here on an entire class, where everything that comes back from this class, goes to this filter assuming we turn the filter on as we'll see. And so the filter itself is defined through the @Filter annotation, which can be applied either on the class or on a specific collection. And in this case, I'm creating a filter that I'm defining as onlyOlderThan. So I only want stories that come older than some date that's coming in, and that data will be passed in later. And so as part of my filter, I just put some condition on the query, that will be applied every time I query for this PublicStory class. And the olderThan parameter has to be defined through this filter definition annotation. And so the name of the annotation definition matches the name of the filter that I have going on here, and then the parameter definition comes in to say I want to pass in a variable called olderThan, of type "date" to be able to basically query on stories olderThan that specific "date". That's the business logic, but the Hibernate mechanics is to create this parameter that gets passed in.   
*Java Eclipse is open on the PublicStory.java tab. The code that is displayed in the editor is:  
  
package hibernate.domain;  
  
import java.util.Date;  
  
@Entity  
@Table(name="Story")  
@FilterDef(name="onlyOlderThan",  
          parameters=@ParamDef( name="olderThan", type="date"))  
@Filter(name="onlyOlderThan", condition="storyDate < :olderThan")  
public class PublicStory  
{  
   @Id  
   @GeneratedValue  
   private long   id;  
     
   @Basic  
   private String title;  
  
The parameter that the presenter refers to is: @ParamDef.*   
  
Now this filter doesn't do anything by itself. I have to set up the filter and tell it that I want it to be used. So let's go take a peek at the code. So over here, I have a Criteria search that I'm going to do. I could be doing a query using HQL just as easily. And so on that Criteria object, on that query object I get back from the session, I have to enable the filter and I'm going to enable the filter by name. So again, it doesn't automatically filter everything all the time. I have to enable it, but it will ensure once I've enabled the filter on that session, that it always works for every query I do. Now if I have a parameter as part of my filter, as I do in this case, I have to get that ageFilter back, the one that is the object that I created right here and I have to set that parameter. So the olderThan parameter that was defined there...I'm going to pass in the time, which is basically for the business logic in this case, I want it to only show all stories older than seven days from now. And so I then can go do and execute my query. I'm not adding any where logic, I'm not adding any Java logic to filter anything there. Hibernate's taking care of all of that by generating the query appropriately.   
*The presenter selects the FilterMain.java tab and the code is displayed:   
  
Calendar cal = Calendar.getInstance();  
           cal.add(Calendar.DATE, -7);  
             
           Criteria c = s.createCriteria(PublicStory.class);  
           Filter ageFilter = s.enableFilter("onlyOlderThan");  
           ageFilter.setParameter("olderThan", cal.getTime());  
           List<PublicStory> all = (List<PublicStory>) c.list();  
  
           for (PublicStory p : all)  
           {  
               System.out.println(p.getStoryDate() + " " + p.getTitle());  
           }*   
  
So if I execute this, I run and I get a bunch of stories that come back from it. We can see here all these red stories is what's getting printed out as return. So when I did my query here, it's selecting out all the different stuff that I would normally select in here, but it adds in the where clause that the story date must be less than. So this is coming from my filter, if we look back here the filter is saying the storyDate is less than, and that's what's being added because of the filter that I'm applying. So this filter, as we can see, the result's down below, only returns stories that were of a date that was a week ago or more. Now just to prove that, I can come inside here, I can comment out the addition of the filter and run this without having turned on the filter.   
*The presenter clicks Run and the output is displayed in the Console. The presenter focuses on a section of the output:  
  
Hibernate: /\* criteria query \*/ select this\_.id as id1\_7\_0\_, this\_.titile2\_7\_0\_, this\_.storyDate as storyDat3\_7\_0\_, this\_.creationDate as creation4\_7\_0\_, this\_.lastModified as lastModi5\_7\_0\_ from story this\_where this\_.storyDate < ?  
  
The presenter selects the PublicStory.java tab.   
  
He scrolls through the Console output. An example of the output is:  
  
2014-02-16-07:35:52:0 Mayor hugs Hedgehog.  
  
The presenter amends the code by adding the // to the code:  
  
Filter ageFilter = s.enableFilter("onlyOlderThan");  
ageFilter.setParameter("olderThan", cal.getTime());  
List<PublicStory> all = (List<PublicStory>) c.list();  
  
The code is now:  
  
//            Filter ageFilter = s.enableFilter("onlyOlderThan");  
//            ageFilter.setParameter("olderThan", cal.getTime());  
           List<PublicStory> all = (List<PublicStory>) c.list();*   
  
And executing this code you'll see I get a whole lot of stories, there's actually a hundred stories that get generated here in my test case, and every single one of them comes back. If I look at the query itself, the criteria query that's going on, what's happening is there is no where clause when I've taken that out. And so the filtering is a great way of ensuring of all the different queries, no matter how complex other stuff is happening, however involved my SQL is, whatever other criteria I'm putting on it, it'll ensure that this data is not returned. Which is great if I have, as I said, a security restriction or some other category that I want to add inside of this. And that can be happening no matter how it's defined; I can define this in annotations or XML, I just simply need to add in the Filter appropriately here at this point.   
*The presenter clicks Run and the output is displayed in the Console. The presenter focuses on a section of output:  
  
Hibernate: /\* criteria query \*/ select this\_.id as id1\_7\_0\_, this\_.titile2\_7\_0\_, this\_.storyDate as storyDat3\_7\_0\_, this\_.creationDate as creation4\_7\_0\_, this\_.lastModified as lastModi5\_7\_0\_ from story this  
  
The presenter amends the code by removing the // from the code:  
  
// Filter ageFilter = s.enableFilter("onlyOlderThan");, and the   
// ageFilter.setParameter("olderThan", cal.getTime());   List<PublicStory> all = (List<PublicStory>) c.list(); code lines.  
  
The code is now:  
  
  
      Filter ageFilter = s.enableFilter("onlyOlderThan");  
      ageFilter.setParameter("olderThan", cal.getTime());  
           List<PublicStory> all = (List<PublicStory>) c.list();*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a01_it_enus/output/html/course_transcript.html#top)

Allowing Hibernate to Create Tables

Learning Objective

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

* *allow Java Hibernate to create a table structure*

**1. Allowing table creation in Hibernate**

Hibernate is not only smart enough to map queries for you. Once you've done the heavy lifting of creating the mapping, you can also create the tables. With a simple config change, you can run many options to check your ddl against your mapping. Hibernate has the ability to create our table structure, as well as creating all the queries and mappings back and forth. Now the use of this is going to be highly up to you, but we'll show you how it's done, and then we'll talk a little bit about where it's practical and impractical to be used. Now the key to this is one simple config property in the hibernate.config file, called the hbm2ddl.auto feature. So that's the Hibernate mapping to the ddl being the table creation script. And this has a bunch of different possible values. We're using the value create here and under create it's assuming you start with an empty database and it's going to create that database and all the tables for you. You can also do create dash drop, which creates it and then drops it, or update, or just simply validate, that it will go off and double-check it on here. Or you can leave the property off all together and Hibernate ignores the ddl.   
*Java Eclipse is open on the hibernate.cfg.xml tab. The file contains the code:  
  
   <! -- Enables adding of comments to SQL when generated -->  
    <property name="hibernate.use\_sql\_comments>true</property>  
      
  
    <!-- Option to sync the Hibernate schema to your database -->   
    <property name ="hbm2ddl.auto">create</property>  
      
      
    <!-- Used to save occasionally when doing large inserts of date -->  
    <property name="hibernate.jdbc.batch\_size">10,/property  
      
    <mapping class="hibernate.domain.Story"/>  
    <mapping class="hibernate.domain.Page"/>  
  
The presenter selects the Page.java tab. The file contains the code:  
  
@Entity  
@Table(name="Page")  
@NamedNativeQuery(name = "resetPageSequence", query="ALTER TABLE Page AUTO\_INCREMENT = 1")  
public class Page  
{  
   @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  
   @JoinColumn(name= "storyId")  
   private Story story;  
  
   @Temporal(TemporalType.TIMESTAMP)  
   private Date   creationDate;  
     
   @Temporal(TemporalType.TIMESTAMP)  
   private Date   lastModifiedDate;*   
  
So we have our Story class that's mapped with it's annotations out here, and it has its ID and its fields, and it actually has a relationship with this Page class out here. And so the Page class also maps to its own table and it has its ID, and it has a relationship back to the Story class. And that actually has a foreign key relationship as part of it as well, as the pages are subordinate to the Story. You can't have a page of a Story without having a Story object. So in this sample, if we go through and look at our existing state of the database, I can say for our database if I show the tables I have an empty set of tables, there's nothing out here. So if I run my little test here, I can execute it and it'll do its stuff. And the test ran fine and it's all great but notice as part of the query here, if the tables existed it would have dropped them, which they didn't, and then it's going to create those tables again. And it actually shows you here the create script it used to create the tables. And then actually you can also see that there is an alter script to add in the foreign key.   
*The presenter selects the Story.java tab and the code is displayed in the editor:   
{  
   @Id  
   @GeneratedValue  
   private long   id;  
     
   @Basic  
   private String title;  
     
   @Basic  
   private Date   storyDate;  
     
   @Temporal(TemporalType.DATE)  
   private Date   creationDate;  
     
   @Temporal(TemporalType.TIMESTAMP)  
   private Date   lastModifiedDate;  
  
   @OneToMany(cascade=CascadeType.ALL)  
   @JoinColumn(name= "storyId", updatable=false)  
  
The presenter opens the Command prompt and runs the command:   
  
show tables;   
  
The output is:  
  
Empty set <0.00 sec).  
  
The presenter returns to the editor in Java Eclipse and selects the CommandLineMain tab. The file contains the code:  
  
   private SessionFactory factory;  
     
   private Create         create;  
   private Read           read;  
   private Update         update;  
   private Delete         delete;  
  
   public static void main(String[] args)  
   {  
       new CommandLineMain().run();  
   }  
  
   public CommandLineMain()  
   {  
       factory = ConfigHelper.getSessionFactory();  
       read   = new Read(factory);  
       create = new Create(factory, read);  
       update = new Update(factory, create);  
       delete = new Delete(factory, create);  
  
The presenter clicks Run, and the output is displayed in the Console. He focuses on a section of the output:  
  
Hibernate: drop table if exists Page  
Hibernate: drop table if exists Story  
Hibernate: create table Page (id bigint not null auto\_increment, creationDate datetime, htmlIncluded bit not null, lastModified...  
Hibernate: create Story Page (id bigint not null auto\_increment, creationDate datetime, htmlIncluded bit not null, lastModified...  
Hibernate: alter Page add constraint FK\_7qmlqw26wvuw5ow4xa814ispy foreign key (storyId) references Story (id)*   
  
Now I don't want you to take my word for it and the stack traces word for it, I don't want to just look at the log, let's go through and look at the database, then we can say show the tables again. So now you can see we have a page table, we have a story table and we can look at the detail of this. I can describe the table story and like our story over here in our Java file, it has an id, a title, a date, creationDate, lastModifiedDate...and you have all those items inside of here, not ordered exactly the same, but id, creation, last modified, storyDate, and title. And that'll have data types equivalent to the Java data types that are out there. You can see it's using bigint over here, and date, and datetime appropriately based off of how I defined those things. Like over here in Java some of these are dates, and some of these are time stamps, and some of these are just straight old-fashioned Java dates. And that will affect what data type Java chooses for you to use over on the other side. The page similarly has the same sort of mappings, the same sort of dates we can see this all out here, but a little bit more interesting. Let's look at the create table script on page, where you can see the create table script shows you that it has a foreign key CONSTRAINT out here.   
*The presenter navigates to the Command Prompt and runs the command:  
  
show tables;.  
  
The output is a table with one column and two rows. The column header is Tables\_in\_bignews, and the rows include page and story.  
  
The presenter runs the command:  
  
describe story;.  
  
The output is a table with six columns and five rows. The column headers are: Field, Type, Null, Key, Default, and Extra.  
  
The first row is:  
Field: id.  
Type: bigint.  
Null: NO.  
Key: PRI.  
Default: NULL.  
Extra: auto\_increment.  
  
The presenter navigates to the Story.java tab in Eclipse and then returns to the Command Prompt.  
He runs the command:  
  
describe page;.  
  
The output is a table with six columns and seven rows. The column headings are: Field, Type, Null, Key, Default, and Extra.  
  
The first row is:  
Field: id.  
Type: bigint.  
Null: NO.  
Key: PRI.  
Default: NULL.  
Extra: auto\_increment.  
  
The presenter runs the command:   
  
show the create table page;.  
  
He examines a section of the output:   
  
CONSTRAINT 'FK\_7qmlqw26wvuw5ow4xa814ispy' FOREIGN KEY <'storyId'> REFERENCES 'story' <'id'>*   
  
And that FOREIGN KEY has been created to the storyId, to the id and table story. So the Hibernate mapping does a very detailed job of creating not just the basic table structure, but the advanced table structure of foreign keys in those relationships. So this type of testing is fantastic for initial development, as I'm starting up an application quickly, maybe I don't have ddl yet, maybe I haven't gotten database administrators involved yet. And so I can start my testing and start testing within my Java objects, and just have that database being created and dropped at any given time. This feature is less useful for existing databases that exist prior to my Java, or that are being maintained by an external database administration team. But still I can at least turn on that validate option, to make sure my database model is aligning to that database appropriately, and I don't have any tricks upfront. It'll tell me upfront if it's wrong. So it's a great feature to be aware of and to use appropriately.

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a01_it_enus/output/html/course_transcript.html#top)

Multiple Mappings of Tables

Learning Objective

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

* *implement multiple mappings of tables in Java Hibernate*

**1. Implementing multiple table mapping**

Hibernate's great at mapping full objects and giving you all the possible columns, and attributes, and stuff that goes back and forth between the database and the Java object. And so we can see here I have a Story object and it has everything mapped out here. So it has a title, a date, other details that are out there, plus a whole list of pages that can come up for a story. Now if I'm running a website, if I'm running a mobile app, I don't want to load every single page of every single story, in order to give somebody a preview of the stories. So I might want to have a second mapping of the same object. For instance, here our StoryBrief object it has less data. So here I have the id, the title, the storyDate, I don't care when it was created, I don't care when it was last modified, and I only care about a single page, not a list of pages, just the first page, I just want to give you a taste for what the story is about. So this mapping allows me to actually go back to that Story table out there. And so my StoryBrief object class name here can tie to the same Story table in the back-end, and query on that Story in a very different way. And you can see here, I'm actually getting a little bit clever, I'm using the where clause to limit my query to just where pageNumber is 1. I don't care if there's 10 pages, or 5 pages, or just 1 page, I only care about the firstPage.   
*Java Eclipse is open on the Story.java tab. The file contains the code:  
  
   @Id  
   @GeneratedValue  
   private long   id;  
     
   @Basic  
   private String title;  
     
   @Basic  
   private Date   storyDate;  
     
   @Temporal(TemporalType.DATE)  
   private Date   creationDate;  
     
   @Temporal(TemporalType.TIMESTAMP)  
   private Date   lastModifiedDate;  
  
   @OneToMany(cascade=CascadeType.ALL)  
   @JoinColumn(name= "storyId", updatable=false)  
   private List<Page> pages = new ArrayList<Page>();  
  
   public void addPage(Page page)  
   {  
       page.setStory(this);  
       page.setPageNumber(pages.size() + 1);  
       pages.add(page);  
   }  
  
The presenter selects the StoryBrief.java tab and the code is displayed in the editor:  
  
   @Id  
   @GeneratedValue  
   private long   id;  
     
   @Basic  
   private String title;  
     
   @Basic  
   private Date   storyDate;  
  
   @OneToOne  
   @JoinColumn(name="id", referencedColumnName="storyID")  
   @Where(clause="pageNumber = 1")  
   private Page firstPage;*   
  
I can take this even further where I can go to just the description of the story. I don't care about any of the pages just give me the ID of the story, the title, and the date. And so I can show you a list of all the stories I have. You can click on it and then I can load it later by ID and show you all the pages. So having this in this case, three-tiered approach to stories, allows me to align my retrieval of data from the database, to the model of how I'm using the data on any given page. Doesn't matter if it's web, or desktop, or batch, it really works well to maximize the efficiency of the use of Hibernate, to align the use of my data model to the query results that are being returned. So let's execute this and see how conveniently this lays out.   
*The presenter selects the StoryDescription.java tab and the code is displayed in the editor:  
  
@Entity  
@Table(name="Story")  
public class StoryDescription   
{  
   @Id  
   @GeneratedValue  
   private long   id;  
     
   @Basic  
   private String title;  
     
   @Basic  
   private Date   storyDate;  
  
   public long getId()  
   {  
       return id;  
   }  
   public void setId(long id)  
   {  
       this.id = id;  
  
The presenter selects the MultipleMappingMain.java tab and the code is displayed:  
  
new BulkInsert(s).createABunchForSearching();  
             
          System.err.println("!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!");  
           System.err.println("See all the short descriptions");  
           List<StoryDescription> descriptions = (List<StoryDescription>) s.createCriteria(StoryDescription.class).list();  
           for (StoryDescription d : descriptions)  
           {  
               System.err.println(d.getTitle() + " " + d.getStoryDate());  
           }  
             
           System.err.println("!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!");  
           System.err.println("Load a short form with just the first page");  
           StoryBrief brief = (StoryBrief) s.load(StoryBrief.class, new Long(1));  
           System.err.println(brief.getTitle() + " " + brief.getStoryDate());  
           System.err.println(brief.getFirstPage().getWords());  
           System.err.println();  
  
            System.err.println("!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!");  
           System.err.println("Now the full form");  
           Story full = (Story) s.get(Story.class, new Long(1));  
           StoryToString.printStory(full);  
             
           System.err.println("Test Complete");  
           System.err.println("!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!");  
           TestHelper.cleanup(factory.getCurrentSession());  
           s.getTransaction().commit();  
       } catch (Throwable t)  
  
The presenter clicks Run and the output is displayed in the Console.*   
  
So if I bring this up and give us some example, here is an example of our full detail of the story. You see it's showing all three pages and so what it did, it went select inside of here, it went through and loaded all the page data that's associated with that. Now before that I loaded just the first page off of there. And so when I did this query, the query looks much the same. I'm still joining between those tables, but the amount of data I'm pulling back is limited to just that first page. And then I dive into here but you can actually see inside of here, there's a limitation on the page that joins everything else where...and I'm not finding it right now...because the Hibernate does that, do the job of a query, but it just ensures that it comes down to just page 1. You can see it's just loading up page 1 inside of there. And then finally the last one here, is loading up a bunch of data and it's doing a very basic criteria search, where it's only loading back these few couple of columns, no joins involved, just limiting my story data that's coming inside of there. So as you're doing your Hibernate don't feel that you're limited to a single mapping, don't feel that you're stuck within the Story object. If there's other uses of data inside of your application, inside of your model, model it appropriately, create a second object, name it appropriately, what's stored back there and only return back the attributes and data that you need for that model.   
*The presenter expands the Console and focuses on a section of the output:  
  
Mayor fist bumps woman - 2014-02-23 00:00:00.0  
  
Page 1   
  
Mayor fist bumps woman - Page 1   
  
  
Page 2   
  
Mayor fist bumps woman - Page 2  
  
  
Page 3  
  
Mayor fist bumps woman - Page 3  
  
creation: 2014-02-23 modified: 2014-02024 08:45:36.0  
Test Complete  
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!  
  
The presenter scrolls up in the Console and focuses on the code:   
  
Load a short form with just the first page   
Hibernate: se;select storybrief0\_.id as id1\_3\_0\_, storybrief0\_.storyDate as storyDat4\_3\_0\_, storybrief0\_.title as title5.......  
  
He scrolls up and examines the output:  
  
Hibernate: /\* insert hibernate.domain.Page \*/ insert into Page (creationDate, htmlIncluded, lastModifiedDate, PageNum...  
Hibernate: /\* insert hibernate.domain.Page \*/ insert into Page (creationDate, htmlIncluded, lastModifiedDate, PageNum....  
Hibernate: /\* insert hibernate.domain.Story \*/ insert into Story (creationDate, lastModifiedDate, storyDate, title) v.....  
  
Then he returns to the Eclipse interface and selects the Story.java tab. He scrolls up to display additional code:  
  
  parameters=  
                                   {@StoredProcedureParameter(name="topic", mode=ParameterMode.IN, type=String.class)}  
                             ),  
   @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   
{*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a01_it_enus/output/html/course_transcript.html#top)

Basic Mapping – Annotations

Learning Objective

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

* *implement a basic mapping using annotations in Java Hibernate*

**1. Applying basic mapping using annotations**

The core to your Hibernate application is mapping your database table to your classes and vice versa. And so we have a database table existing. Now you can start from classes, but we're going to start from a database table for this example. And so our table is just called Generic and it has an id, some text, a date, a double for some money out there, and a Boolean, just as a sample representative data that you could have possibly out there in the database. Not everything, but a few nice attributes out there. And so inside of our Java class I want to create attributes that map to each one of these columns. That's basically the way Java works, is I have classes with attributes, and databases have rows with columns and so they map together nicely. And so inside of our class, the first thing I need to start with is an Entity tag to tell Hibernate, "Hey, this is something that does get persisted, it's something that ties to the database." And since the class is named the same thing as a table, that's all I have to do. It uses that convention to simply automatically map from there.   
*Java - Eclipse is open on the Generic.java tab. The file contains the code:  
  
package hibernate.domain;  
  
import java.util.Date;  
  
@Entity  
public class Generic  
{  
   @Id  
   private long id;  
     
   @Basic  
   private String text;  
     
   @Basic  
   @Column()  
   private Date aDate;  
     
   @Basic  
   private double sampleMoney;  
     
   @Basic  
   private boolean aBoolean;   
  
The presenter opens the Command Prompt. The describe generic command has been run and the output is a table with six columns and five rows. The column headings are, Field, Type, Null, Key, Default, and Extra.  
  
The first row is:  
Field: id.  
Type: bigint<20>.  
Null: NO.  
Key: PRI.  
Default: NULL.  
Extra: an empty space.  
  
The presenter returns to the Java - Eclipse interface.*   
  
Past that, I have my attributes and I have to have an ID with an id tag. Again all of these align directly, where the name of the attribute matches the name of the column of the database. And so the ID becomes the primary key of this. And here I have text and so I can keep it as a Basic tag, which automatically maps it, or you can add a Column tag to add more features, more information about that later on. I can go through and I can put in dates, I can put in money, I can put in the Booleans, everything I want inside of there, and I can even have columns I don't care about making them Transient. This means it's not stored in the database. Passed this, my class is required to have an empty constructor, which in this case, I have no constructor and meets the needs of that, but if I add a constructor, I also have to have an empty constructor. And then getters and setters for each one of these. And note for the boolean, it's not get a boolean, it's isaBoolean that gets put as the getter and setter. That's the convention that comes up with Booleans inside of there. And then ideally you also include equals and hashCode. It's not absolutely required, but it's useful. So that way you can have a full experience of where these are used inside of the collections. So from there I have a class that's mapped over and I can go through and I can execute this. I have all these things up and ready, and then I can go through and I can insert the values into my database.   
*The presenter scrolls down to display additional code:  
  
@Transient  
   private int countNotPersisted;  
     
   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;  
   }  
  
   public Date getaDate()  
   {  
       return aDate;  
   }  
  
public void setaDate(Date aDate)  
   {  
       this.aDate = aDate;  
   }  
  
   public double getSampleMoney()  
   {  
       return sampleMoney;  
   }  
  
   public void setSampleMoney(double sampleMoney)  
   {  
       this.sampleMoney = sampleMoney;  
   }  
  
   public boolean isaBoolean()  
   {  
       return aBoolean;  
   }  
  
   public void setaBoolean(boolean aBoolean)  
   {  
       this.aBoolean = aBoolean;  
   }  
  
   public int getCountNotPersisted()  
   {  
       return countNotPersisted;  
   }  
  
   public void setCountNotPersisted(int countNotPersisted)  
   {  
       this.countNotPersisted = countNotPersisted;  
   }  
  
   @Override  
   public int hashCode()  
   {  
       final int prime = 31;  
       int result = 1;  
       result = prime \* result + (int) (id ^ (id >>> 32));  
       return result;  
   }  
  
   @Override  
   public boolean equals(Object obj)  
   {  
       if (this == obj)  
           return true;  
       if (obj == null)  
           return false;  
       if (getClass() != obj.getClass())  
           return false;  
       Generic other = (Generic) obj;  
       if (id != other.id)  
           return false;  
       return true;  
   }  
}*   
  
I'm going to go back to my Main, I'm going to run my Main so you can see I'm creating an instance of each one of these classes. I'm filling them with data. I'm doing a save of each one of these guys and then throwing all of those in my database. And then I can see the values from my database, where I've inserted here a Simple Object, not everything was filled, and then a Full Object, where everything indeed did get filled. And so we've been able to very quickly take a class, create a very basic Java class to map to that database, using these simple annotations that work simply because everything aligns really well. We're adhering to the conventions, and then with some really simple calls into the Hibernate framework, we can just simply save those objects and they get thrown in the database. So that's a very basic mapping, a very basic intro, and then we can build on this mapping to do more interesting things from there.   
*The presenter selects the GenericMain.java tab. The file contains the code:  
  
private SessionFactory factory;  
  
   public static void main(String[] args)  
   {  
       new GenericMain().run();  
   }  
  
   private void run()  
   {  
       try  
       {  
           factory = ConfigHelper.getSessionFactory();  
           Session s = factory.getCurrentSession();   
           s.beginTransaction();  
  
           {  
               Generic g = new Generic();  
               g.setText("Simple Object");  
               s.save(g);  
           }  
             
           {  
               Generic g = new Generic();  
               g.setId(1);  
               g.setText("Full Object");  
               g.setaBoolean(true);  
               g.setaDate(new Date());  
               g.setSampleMoney(199.99);  
               g.setCountNotPersisted(5);  
               s.save(g);  
                 
               g.setText("Update Full Object");  
               s.update(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());  
           }  
             
           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 Run and the output displays in the Console.  
  
In the Command Prompt, he runs the command:  
  
source showGenerics.txt.  
  
The output is a table with five columns and two rows. The column headings are: id, text, eDate, sampleMoney, and eBoolean.  
  
The first row is:  
  
id: 0.  
text: Simple Object.  
eDate: Null.  
sampleMoney: 0.00.  
eBoolean: an empty field.  
  
The second row is:  
  
id: 0.  
text: Full Object.  
eDate: 2014-02-25.  
sampleMoney: 199.99.  
eBoolean: a boolean.  
  
The presenter then navigates back to the GenericMain.java tab on the Eclipse interface.*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a01_it_enus/output/html/course_transcript.html#top)

Basic Mapping – XML

Learning Objective

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

* *create XML mapping between a class and a table in Java Hibernate*

**1. XML mapping between tables and classes**

The core of Hibernate is mapping between your class files into the database and XML mapping is the original version that was released with Hibernate. It was later replaced with annotations but it's still a valid part of creating Hibernate mappings. And so we're going to explore a basic, simple little Hibernate mapping in XML between a very simple class and a very simple table. And so the mapping here in XML first has to point to a class. And you can see the mapping here has a class tag that will tell you what the name of the class is, and you can see the name here equates to our Java class over here, just called generic. And then it equates to a table out here. The table name might be the same at as the class name and it might not. If it is the same, you can omit this table property, but if it's not the same, I can point to it separately here. And so in this case I have a case called, a table called generic and it has an auto id generator in there so I have the table called generic\_auto.   
*Java - Eclipse is open on the Generic.hbm.xml tab. The file 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="Generic" table="generic auto">  
       <id name="id">  
           <generator class="native"/>  
       </id>  
       <property name="text"/>  
       <property name="aDate"/>  
       <property name="sampleMoney"/>  
        
         </property>  
         <property name ="aBoolean"/>  
   </class>  
</hibernate-mapping>  
  
The presenter selects the Generic.java tab. The file contains the code:  
  
package hibernate.domain;  
  
public class Generic  
{  
   private long id;  
   private String text;  
   private String aDate;  
   private String sampleMoney;  
   private String aBoolean;  
   private double countNotPersisted;  
  
   public long getId()  
   {  
                return id;  
   }  
  
   public void setId(long id)   
     
The presenter returns to the Generic.hbm.xml tab.*   
  
Past that every class must have an ID. And so I have to show what the primary key of the table is and what the primary key of the class is. Now it doesn't really require it to be a primary key on the database side, but it does help if it is, it solves a lot of problems outside of there. And then past that you just map the rest of the attributes. So if we go back to the Java class here for a second, you can see I have an ID and then a bunch of different attributes that come up here– so text, and dates, and money amounts, or doubles, and Booleans, and however I want to represent the data that goes on inside of here. It's all based on your business, your model your objects. Here's a smattering of some example types that you can be using and not even comprehensive inside of there, but that all gets reflected over here in the items that go to the database. Now speaking of the database, we can do a quick look at the database table and you can see here are the columns of the database table that align very nicely with the rows and the attributes that come up into the Java class. In this case we actually are naming them pretty much exactly the same thing – not a requirement, but it is convenient because there's a automatic mapping that happens when they are named the same thing. It just assumes that and it moves forward with it.   
*The presenter selects the Generic.java tab again. Then he presenter opens the Command Prompt and runs the command:   
  
describe generic\_auto.  
  
The output is a table with six columns and five rows. The column headings are: Field, Type, Null, Key, Default, and Extra.  
  
The first row is Field: id.  
Type: bigint(20).  
Null: NO.  
Key: PRI.  
Default: Null.  
Extra: auto\_increment.  
  
The presenter navigates back to the Generic.hbm.xml tab on the Eclipse interface.*   
  
So past that we are ready to go, we are ready to test this. We have a bunch of attributes that are mapped and we can have a little simple class that we just are going to build two different objects right here in Java and then we're going to save those into our database. So if we go through and run our sample here, it's going to load up our mapping, it's going to initialize the Hibernate, and then you can see Hibernate has generated some insert statements here. Two of them that's inserting data into our solution and very simply saving that data, and then loading those back. So we save them each there, and then we can load those back in through a search, and we're printing out the two simple objects that we just created. If we go over our database side, we can take a quick search on this, alright. I have to actually run that test again because I just ran my clean up script instead.   
*The presenter selects the GenericMainXML.java tab. The file 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 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();  
  
           {  
               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  
       {  
           // To kill the run else Hibernate leaves it active.  
           factory.close();  
           System.exit(0);  
       }  
   }  
}  
  
  
The presenter clicks Run and the output is displayed in the Console. The presenter then focuses on the following output:  
  
Hibernate: insert into generic auto (text, aDate, sampleMoney, aBoolean) values (?, ?, ?, ?)   
Hibernate: insert into generic auto (text, aDate, sampleMoney, aBoolean) values (?, ?, ?, ?)   
  
The presenter opens the Command Prompt and runs the command:  
  
source setup.txt;.  
  
The output is a list of data including:  
  
Query OK, 1 row affected <0.00 sec>   
Query OK, 1 row affected <0.00 sec>   
Query OK, 1 row affected <0.00 sec>   
Query OK, 1 row affected <0.00 sec>   
Query OK, 0 rows affected <0.00 sec>   
Query OK, 0 rows affected <0.00 sec>   
Query OK, 0 rows affected <0.00 sec>   
  
The presenter navigates back to Eclipse and clicks Run.*   
  
So we run our test again and there are two simple objects that we just created. And now I run the script to show the table and you can see there is our two simple objects that got shown in the table. The one object didn't get field properties for dates, money, or Booleans. But notice the date is NULL because there's no default for date and it shows up as NULL in the database, and the other two have a default of zero, or in this case the Boolean, false so they show up as values in the database not as null values because they were something in Java. If I want them truly be null I'll have to map that separately and not use simple variables...simple primitive types...it use the object types instead. But then you can see the second one has all those values filled in exactly as I filled in inside of my solution. So very simple, very easy mapping you can start it with, and it does all the basic operations I want to right out-of-the-box once I have the mapping done.   
*The presenter opens the Command Prompt and runs the command:  
  
source showGenerics.txt.  
  
The output is a table with five columns and two rows. The column headings are: id, text, aDate, sampleMoney, and aBoolean.  
  
The first row is id: 1.  
text: Simple Object.  
aDate: Null.  
sampleMoney: 0.00.  
aBoolean: 0.*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a01_it_enus/output/html/course_transcript.html#top)

Associations and Foreign Keys – Annotations

Learning Objective

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

* *expand an object model in Java Hibernate*

**1. Expanding object models in Hibernate**

It's vital in your object model to have objects that are associated with other objects. It's just simply part of the way you design things. And the same thing is true in your database; you have associated classes through foreign keys, associated tables through foreign keys. And so inside of our mapping we have to be able to represent the mapping between those relationships. They map fairly well conceptually; we just go to figure how to do them back and forth. So in this example we have a story that's made up of a bunch of pages. And we have a very tight composite relationship where that page belongs to a story, the story is made up of pages. And so our basic entity is set up for the story. We have an ID, we have title and dates that are associated with that, and then we have a list of pages that we're looking at. And the list has a specific connotation, it has an order that's associated with that. We need those pages to come back in page order, they it wouldn't make sense out of order there; so that's why we represented this as a list. And the way we can represent this is as a OneToMany relationship with this page. This is a database and a Java notion, OneToMany, and by modeling it through that in Hibernate we're really creating a nice bond between these two classes and between these two tables.   
*Java Eclipse is open on the Story.java tab. The file contains the code:  
  
})  
public class Story   
{  
   @Id  
   @GeneratedValue  
   private long   id;  
     
   @Basic  
   private String title;  
     
   @Basic  
   private Date   storyDate;  
     
   @Temporal(TemporalType.DATE)  
   private Date   creationDate;  
     
   @Temporal(TemporalType.TIMESTAMP)  
   private Date   lastModifiedDate;   
  
OneToMany(cascade=CascadeType.ALL)  
   @JoinColumn(name= "storyId", updatable=false)  
   private List<Page> pages = new ArrayList<Page>();  
  
   public void addPage(Page page)  
   {  
       page.setStory(this);  
       page.setPageNumber(pages.size() + 1);  
       pages.add(page);  
   }  
     
   public long getId()  
   {*   
  
Now the classes make sense. I have that relationship simply by having the attribute pages and having a list of pages that's easy in Java. On the database side though, I need to map that through foreign keys. So I have to have a JoinColumn that sets up between there. So if I go look at my database I can say, "Hey, what does my story look like and what does my page look like?" So I can see they have all the basic columns out there, the id of the Story column is here with the primary key. The id of the Page table is here with the primary key, but the storyId is that foreign key that ties these two things together. There's a foreign key designed to say that, "Hey, every page has to have a story that's related to," and that story is tied through by this foreign key. So that relationship is reflected here. So for the pages, if I want to load those pages in, I go look for all of the pages where the storyId is equal to my id. And this column's essential because it lets me know when I load my story which pages to pull in. Now that's half of the relationship.   
*The presenter opens the Command Prompt and runs the command:  
  
describe Story;.  
  
The output is a table with six columns and five rows. The column headings are: Field, Type, Null, Key, Default, and Extra.  
  
The first row is:  
  
Field: id.  
Type: binit(20).  
Null: NO.  
Key: PRI.  
Default: NULL.  
Extra: auto\_increment.  
  
The presenter runs the command:  
  
describe Page;.  
  
The output is a table with six columns and seven rows. The column headings are: Field, Type, Null, Key, Default, and Extra. The first row is:  
  
Field: id.  
Type: binit(20).  
Null: NO.  
Key: PRI.  
Default: NULL.  
Extra: auto\_increment.*   
  
This relationship right now could be considered one directional but we want a bidirectional relationship. So we have to also set up a similar relationship over on the Page side. So for the page again we have an Entity, we have a Table, we mark that all as it supposed to be marked from there. But on the Page side we also needed to tie back to the story. And for this case each pages belongs to one and only one story, there's many pages that could belong to that story but it's only one that's out there. And so the JoinColumn is still essential. I tie to the storyId that's what I need to know by. If I want to go and find out what story I belong to, I look at my storyId and the Id of that story over there, and then through that many-to-one relationship I'm be able to tie myself back. That's the other half of this bidirectional relationship that's going on side of there. I don't know why that space is there, it actually doesn't change the behavior but I'm taking it out. So the story now in my object model is just simply an attribute. I have getters and setters on Story, it's a full object I'm pointing to, if I want to know about the story I just simply say get Story and have access to all its attributes. And so from a Java model it looks exactly the same as I would always model it in Java, there's no extra Id's floating around or anything I need to worry about.   
*The presenter selects the Page.java tab and the code is displayed in the Editor:  
  
package hibernate.domain;  
  
@Entity  
@Table(name="Page")  
@NamedNativeQuery(name = "resetPageSequence", query="ALTER TABLE Page AUTO\_INCREMENT = 1")  
public class Page  
{  
   @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  
   @JoinColumn(name= "storyId")  
   private Story story;  
  
   @Temporal(TemporalType.TIMESTAMP)  
   private Date   creationDate;*   
  
The other thing I want to point out is the OrderColumn here on pageNumber. I'm tying the OrderColumn to that pageNumber and I'm saying, "Hey, this guy is the thing that gets loaded." I'm going to basically do an order by on pages when I load them. So I don't want the pages to come back in random order, I want them to come back in pageNumber order and that makes a lot of sense for my mapping of stories. It both fits the business model and it also fits the way I need to represent it in the database. So I can go through and I can run a little test to insert some stories into my database. And so I'm going through I'm running it, I'm creating some stories randomly, and then I'm loading up so you can see. So from the Java data model I have a story loaded from the database, it has a title and a date associated with it. And then there's two pages that gets loaded with this story. The first page and the second page and notice they came back in page order. Now honestly, 50-50 shot, it could have come back there anyway but that OrderColumn on the page ensured that came back in that order.   
*The presenter selects the CommandLineMain.java tab and the code is displayed in the editor:  
  
factory = ConfigHelper.getSessionFactory();  
       read   = new Read(factory);  
       create = new Create(factory, read);  
       update = new Update(factory, create);  
       delete = new Delete(factory, create);  
   }  
  
   private void run()  
   {  
       try  
       {  
           factory.getCurrentSession().beginTransaction();  
  
//  Create Demo  
//            create.run(Create.PERSIST);  
//            create.run(Create.SAVE);  
//            create.run(Create.SAVE\_OR\_UPATE);  
//            create.run(Create.SAVE\_OR\_UPATE\_EXISTING);  
  
He clicks Run and the output is displayed in the Console. The presenter focuses on the following output:  
  
Welcome to Hibernate!: 2014-02-25 00:00:00.0  
  
Page 1  
  
I am the first page  
  
Page 2   
  
I am the second page.*   
  
So beyond that let's look at our database. And we can see I got the object that was loaded for the story, I got the two objects that were loaded for the page to become rows inside of my database. And all of those things are set up where their pageNumber order and all of the values are being set appropriately. So our foreign key associations can very easily be modelled inside of our annotations using the one-to-many and the many-to-one attributes, the annotations inside of there and then basically tying those back and forth by using the JoinColumns to let it know which database items are out there. And then both models were happy and function as they always would.   
*The presenter opens the Command Prompt and runs the command:  
  
source showStory.txt.  
  
Two tables are displayed. The first table has five columns and one row. The row data is read as:  
  
id: 1.  
title: Welcome to Hibernate.  
storyDate: 2014-02-25.  
creationDate: 2014-02-25 00:00:00.  
LastModifiedDate: 2014 08:19:11.   
  
The second table contains seven columns and two rows. The first row is:  
  
id: 1.  
storyId: 1.  
pageNumber: 1.  
words:  I am the first page.  
htmlIncluded: blank.  
creationDate: 2014-02-25 00:00:00.  
LastModifiedDate: 2014 08:19:11.  
  
The presenter navigates back to the Eclipse interface and selects the Story.java tab.*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a01_it_enus/output/html/course_transcript.html#top)

Associations and Foreign Keys – XML

Learning Objective

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

* *use XML mapping options to allow the framework to automatically maintain the relationship between Java classes and database tables in Hibernate*

**1. Enabling associations and foreign keys**

Our data model and our object model has to be interesting in ways where it extends past single classes, single tables, and into relationships between classes and between tables. And so as an example we have a story here. And the story is a pretty basic class, and it has a title and some dates associated with it. But most importantly, it has a relationship with a set of pages. Now it could be a list, it could be a map, it doesn't really matter, but in this case, we're using a set of pages as the example. And so a page object is a separate class, it has its own characteristics, its own attributes, it could have additional relationships. But in this case it doesn't, but it does tie back to its parent story. We have a bidirectional relationship here, going back and forth between stories and pages. A page belongs to a story, a story has a bunch of pages that's associated with that. And so we need to be able to map that relationship that also exists within our database. So if we take a look real quick at our database mapping, we can see we have the story out here and we have the id, the title, all that sort of stuff that goes associated with that. And then in our page, we have this storyId that ties back as a foreign key relationship to the story. We don't see that this described, but it's there let me tell you. And so we need to be able to maintain that relationship between these. And being a foreign key, there's other implied requirements for being able to manage these things back and forth in this bidirectional relationship.   
*Java Eclipse is open on the Story.java tab. The file contains the code:  
  
package hibernate.domain;  
  
public class Story  
{  
   private long   id;  
   private String title;  
   private Date   storyDate;  
   private Date   creationDate;  
   private Date   lastModifiedDate;  
   private Set<Page> pages = new HashSet<Page>();  
  
   public void addPage(Page page)  
   {  
       page.setParent(this);  
       page.setPageNumber(pages.size() + 1);  
       pages.add(page);  
   }  
  
  
The presenter selects the Page.java tab. The file contains the code:  
  
package hibernate.domain;  
  
import java.util.Date;  
  
public class Page  
{  
   private Long id;  
   private Story parent;  
   private int pageNumber;  
   private String words;  
   private boolean htmlIncluded;  
   private Date   creationDate;  
   private Date   lastModifiedDate;  
  
   public Long getId()  
   {  
       return id;  
   }  
   public void setId(Long id)  
   {  
  
The presenter opens the Command Prompt, which displays two tables as output. The first table has six columns and five rows. The column headings are, Field, Type, Null, Key, Default, and Extra.  
  
In the first table, the first row's data is:  
  
Field: id.  
Type: binit(20).  
Null: NO.  
Key: PRI.  
Default: NULL.  
Extra: auto\_increment.  
  
The second table has six columns and seven rows. The first row's data is:  
  
Field: id.  
Type: binit(20).  
Null: NO.  
Key: PRI.  
Default: NULL.  
Extra: auto\_increment.*   
  
So if we look at our Hibernate mapping for Story, we have mostly everything is the same. We have our id, we have our properties, those are all mapped together, but now when we go map the pages we're using the set tag. And the set tag is pointing to the attribute pages, which is inside of here, the attribute and the getters and setters that's associated with that. So the pages attribute over here. It ties to the table page and we are going to cascade all of our actions, because remember we said this is a bidirectional association. Anytime we add a story, we want to add all of its pages. If we delete a story, we want to delete all of its pages as well, and so on and so forth, updates, and things like that. Few other attributes inside of here that we can talk about. We can choose to lazy load these things or not. That's basically when I load a story, do I automatically load those pages, and those are different options we can select as we go forward. But the most important thing about this relationship is we need to know what the key column is. What is the column in that other field that ties back to us? If we want to go load all the pages, we need to do some sort of join and some sort of column and that column is the storyId, that's the foreign key that's out there. So we know we need to load in all these pages by the foreign key and then it maps to everything else. The page class, we have a one-to-many relationship. I am one story, I have many pages that relate to me and so I'm going to do the rest of the mapping by looking at the page mapping.   
*The presenter navigates back to the Eclipse interface and selects the Storyhbm.xml tab. The output in the Console is:   
  
<?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="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>  
  
The presenter navigates back to the story.java tab and then back to the Story.hb.xml tab.*   
  
This is half of our bidirectional association. We don't have to have a bidirectional association. This mapping would work if the pages didn't care in the slightest about the stories, but it is essential for our half, the story half. On the page half then, pretty much normal old mapping, id and fields as it normally would be. But for that mapping back to the parent, we're going to do the many-to-one here. So I only have one object I'm going to tie to, that object is named parent. And you can see inside of our page we have a attribute named parent, getters and setters named parent, that goes along with that and that points to the Story class. And the key column once again that we care about in this relationship is the storyId. So inside of myself, the page class, the storyId gives me that foreign key relationship that ties to the story and its ID. And so I know I'll load the parent that's associated with the story, that's the key I want to. Again it's not required for a bidirectional association, but by putting both the set with the one-to-many, and the page with the many-to-one we've tied these things together.   
*The presenter selects the Page.hbm.xml tab and the code displays:  
  
<?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="Page" table="page">  
       <id name="id" column="id" type="long">  
           <generator class="native"/>  
       </id>  
       <property name="words"/>  
       <property name="pageNumber" insert="true" update="true"/>  
       <property name="htmlIncluded"     type="boolean"/>  
       <property name="creationDate"     type="timestamp"/>  
       <property name="lastModifiedDate" type="timestamp"/>  
         
       <many-to-one name="parent" class="hibernate.domain.Story">  
           <column name="storyId" not-null="true" />  
       </many-to-one>  
   </class>  
</hibernate-mapping>  
  
The presenter selects the Page.java tab and the  code is displayed:  
  
package hibernate.domain;  
  
public class Story  
{  
   private long   id;  
   private String title;  
   private Date   storyDate;  
   private Date   creationDate;  
   private Date   lastModifiedDate;  
   private Set<Page> pages = new HashSet<Page>();  
  
   public void addPage(Page page)  
   {  
       page.setParent(this);  
       page.setPageNumber(pages.size() + 1);  
       pages.add(page);  
   }  
     
   public long getId()  
   {  
       return id;  
   }  
   public void setId(long id)  
   {  
       this.id = id;  
  
The presenter navigates back to the Page.hbm.xml tab.*   
  
Now as a quick demo here you can see we're creating a story, we're creating a couple of pages, and then we're saving it. When we do the save, we don't have to save the pages individually. There's no looping and saving the pages, I will automatically cascade the save the pages when I save the story. So when I go through and then run this, it loads up the mapping, it executes our queries. And then again our test is loading them back up to see that they're loaded, but we can see it inserts the story because it needs that primary key to get set first. And then I can insert the pages that have the foreign key relationship to the story. And so I can come back over here and I can go to my database and I can show you there, that our story was inserted and then our two pages are inserted with the storyId tying to the Id that was set up above. So very simply, we are able to map back and forth. Okay, I say simply, there is sometimes some things there to get worked out but if you get all the key pieces correct – the class mappings, the key mappings in each direction – it does tend to be fairly effortless and maintain itself over time once you get everything set up the first time.   
*The presenter selects the XMLCommandLineMain.java tab and the code is displayed:  
         
       System.err.println(output);  
   }  
     
   private void addItems()  
   {  
       Story story = new Story();  
       story.setTitle("Welcome to Hibernate!");  
       Date now = new Date();  
       story.setStoryDate(now);  
       story.setCreationDate(now);  
       story.setLastModifiedDate(now);  
         
 Page page1 = new Page();  
       page1.setWords("I am the first page");  
       page1.setCreationDate(now);  
       page1.setLastModifiedDate(now);  
       story.addPage(page1);  
  
       Page page2 = new Page();  
       page2.setWords("I am the second page");  
       page2.setCreationDate(now);  
       page2.setLastModifiedDate(now);  
       story.addPage(page2);  
  
       Session session = factory.getCurrentSession();  
       session.beginTransaction();  
       session.persist(story);  
       session.getTransaction().commit();  
   }  
}  
  
The presenter clicks Run and the output is displayed in the Console. The presenter then opens the Command Prompt and runs the command:   
  
source showStory.txt;.  
  
Two tables are displayed. The first table has five columns and one row.  
  
The data is:  
  
id: 1.  
title: Welcome to Hibernate.  
storyDate: 2014-02-25.  
creationDate: 2014-02-25 07:18:02.  
LastModifiedDate: 2014 00:18:02.  
  
The second table contains seven columns and two rows.  
  
The data for row 1 is:  
  
id: 1.  
storyId: 1.  
pageNumber: 1.  
words:  I am the first page.  
htmlIncluded: blank.  
creationDate: 2014 00:18:02.  
LastModifiedDate: 2014 00:18:02.  
  
The presenter closes the Command Prompt and the Eclipse interface is displayed.*

[Back to top](http://xlibrary.skillport.com/courseware/Content/cca/jl_hbnt_a01_it_enus/output/html/course_transcript.html#top)

© 2016 Skillsoft Ireland Limited