




**Hibernate, ORM, and JPA  
for  
Portland Java User Group**

**nTier Training, Inc.**

info@nTierTraining.com 

www.nTierTraining.com 

**Notes**



**LEARNINGPATTERNS**

**Part 1**


**Hibernate the Traditional Way**

- Hibernate Overview
- Using Hibernate
- Mapping a Simple Class
- Inserting, Updating, and Deleting
- Querying and HQL

**Objectives**

- ◆ Describe the issues of object-relational mapping (ORM)
- ◆ Describe the overall goals and architecture of Hibernate
- ◆ Map a simple class to a DB using Hibernate
- ◆ Learn how to retrieve, insert, update, and delete Hibernate persistent objects
- ◆ Be familiar with querying using Hibernate Query Language (HQL)

2



**LEARNINGPATTERNS**

## Hibernate Overview

**Hibernate Overview**  
Using Hibernate  
Mapping a Simple Class  
Inserting, Updating, and Deleting  
Querying and HQL

### The Issues with Persistence Layers

- ◆ Data is a core element of many applications
  - Data is often stored in relational databases
- ◆ A great deal of effort is expended writing persistence layers to store and retrieve data from the database
  - These layers are complicated to write
  - Often the "homegrown" persistence layer is buggy and incomplete
  - Changes to the database schema are often expensive to propagate to the persistence layer
- ◆ When using an OO language like Java, the situation is made more complicated

4

## Object-Relational Mapping (ORM) Issues

- ◆ When using an OO language with a relational database, you are working with different models of using data
  - OO: Interacting objects traversed via relationships
  - Relational: Tables that are joined via foreign key relationships
- ◆ These are very different models
  - Translating from one to the other requires significant effort
  - There are many issues to deal with
- ◆ Example: Inheritance: A core feature of OO models
  - Not supported in relational model
- ◆ Relational associations: only foreign key relationships
  - Object model: 1-1, 1-N, N-N

5

## Issues with JDBC Alone

- ◆ You don't store/load **objects** to the database
  - Everything has to be done with SQL
- ◆ For example, to persist an object of some class
  - You need to write SQL statements to do this
- ◆ To load an object from the database
  - You get result rows from the database, which must be converted into objects
    - Coding this is a tedious and error-prone process

6

## Hibernate Overview

- ◆ Open source, ORM framework for Java
  
- ◆ Overall goals
  - **Relieve the developer from 95%** of common data persistence related programming tasks
  - Provide **ORM capability** for using a **Java OO domain model** to manage a relational database
  - Provide a **light-weight POJO based framework** for Java persistence, including **inheritance** and **polymorphism**
  - Provide a **query language** that helps **remove or encapsulate vendor specific SQL** code
  - Help with **translation of result-set tables to object graph**

7

## Hibernate Benefits

- ◆ Simplicity and flexibility
  - POJO based!
  - ORM is completely metadata driven (annotations or XML)
  - No need to write JDBC code
  - Common defaults that allow metadata to be minimized
  - Persistence API is totally separate from entity classes
  - Can be used in Java SE environments (without app server)
- ◆ Completeness
  - Supports full range of OO features
    - Inheritance, custom types, collections, associations
  - Powerful query language
- ◆ Performance
  - Minimizes number of database interactions
  - Object caching

8

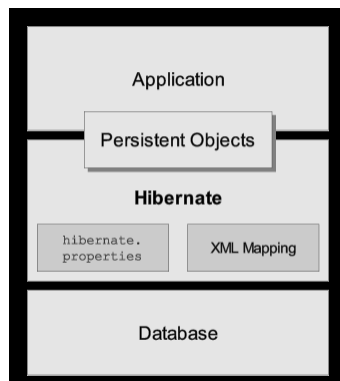
## Hibernate Environments

- ◆ Hibernate can be used in Java SE or Java EE programs
  - It is packaged as a set of jar files
  - There are only minor configuration and programming differences between using it in a Java SE and Java EE environment
  
- ◆ The Java Persistence API derived a lot of its structure from Hibernate
  - Java Persistence is a required part of EJB 3.0, and uses annotations to specify mapping metadata about persistent classes
  - Hibernate supports the standard Java Persistence API annotations, in addition to defining its own
    - Using Hibernate annotations gives a smooth migration path to standard Java Persistence

9

## Hibernate Architecture – High Level View

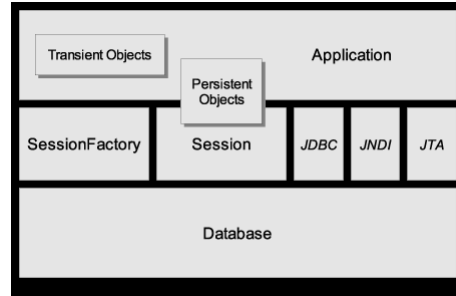
- ◆ Hibernate uses database and configuration data to provide persistence services and objects to the application
  
- ◆ Many ways to architect this – we'll show two common ways



10

## Hibernate Architecture – Detailed View 1

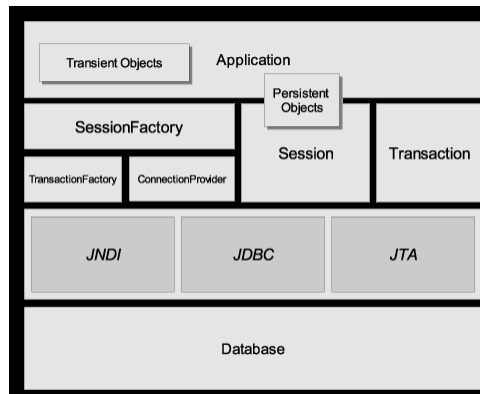
- ◆ In this scenario, the application manages some details itself
  - JDBC connections and transactions, JNDI, etc.
  - Uses minimal subset of Hibernate API
  - This can be considered Hibernate "Lite"




11

## Hibernate Architecture – Detailed View 2

- ◆ In this scenario, the application interacts only with the Hibernate API
  - Hibernate takes care of the other details
  - This is the more common way that Hibernate is used



12



**Using Hibernate**

- Hibernate Overview
- Using Hibernate**
- Mapping a Simple Class
- Inserting, Updating, and Deleting
- Querying and HQL

### Acquiring Hibernate

- ◆ Hibernate is an open source project
  - It can be freely downloaded from <http://www.hibernate.org>
  - Can get binary or source distributions
  - Can also view and download documentation (included in binary)
- ◆ **Hibernate Tools**, a plugin for Eclipse (Hibernate 3.x) is available at <http://tools.hibernate.org> and includes:
  - Mapping editor to create/manage Hibernate mapping files
  - Console that allows you to configure DB connections, visualize classes and their relations, and execute queries interactively
  - Reverse engineering tools that can generate domain model classes and Hibernate mapping files
  - Wizards for creating various Hibernate artifacts
  - Ant task allowing you to run Hibernate tasks as part of your build

14

## Using Hibernate

- ◆ The Hibernate binary is a set of jar files
  - The main Hibernate jar, (*hibernate3.jar* for Hibernate 3 releases), is included in the root of the Hibernate distribution
  - A set of supporting jar files, is included in the *lib* directory of the distribution
    - Not all the supporting jar files are always needed to run hibernate
    - There is a *readme.txt* file that tells what each jar is for and when it's needed

15

## Configuring Hibernate

- ◆ Hibernate needs to be configured to connect to the specific database you are using
- ◆ It also needs to be configured for the type of database
  - Hibernate may do things differently for different databases
- ◆ The configuration can be done in multiple ways
  - Via an XML configuration file (named **hibernate.cfg.xml** by default)
  - Via a properties file (named **hibernate.properties** by default)
    - The properties file was used in earlier versions of Hibernate
    - It is still supported, but most users are now using the XML file
  - Programmatically via a Java API
- We'll look mainly at the *hibernate.cfg.xml* file

16

## Basic Structure of hibernate.cfg.xml

- ◆ The configuration file has the structure shown below
  - A <session-factory> element contains the configuration for the connection to the database
  - The file needs to be on the classpath

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
  <!-- configuration for a SessionFactory instance -->
  <session-factory>
  </session-factory>
</hibernate-configuration>
```

17

## Session Factory Configuration

- ◆ Needs to have the usual database information
  - Database JDBC URL
  - Database driver
  - Username, password
  - You may also specify a datasource name if you are using one
- ◆ You can also specify a database dialect
  - This is the fully qualified classname of a class that customizes Hibernate's behavior for a specific database
- ◆ Other properties
  - Many other things can be specified in the configuration
  - For example, caching related properties

18

## Session Factory Configuration Properties

- ◆ There are many properties available
  - We show some of the important ones here – see the docs!
  - There are many properties to tune the connection, caching, transactions
- ◆ JDBC connection properties
  - **hibernate.connection.driver\_class**: JDBC driver class
  - **hibernate.connection.url**: JDBC URL
  - **hibernate.connection.username**: database user
  - **hibernate.connection.password**: database user password
  - **hibernate.connection.pool\_size**: max pooled connections
- ◆ DataSource connection properties
  - **hibernate.connection.datasource**: datasource JNDI name
  - **hibernate.jndi.url**, **hibernate.jndi.class**: JNDI properties
- ◆ Optional configuration properties
  - **hibernate.dialect**: Classname of a Hibernate dialect
  - **hibernate.show\_sql**: Write all SQL statements to console
  - **hibernate.cache.provider\_class**: Classname of custom cache provider

19

## Example Properties for PostgreSQL

- ◆ The example below shows properties that would be suitable if you were using the open-source PostgreSQL database

```
<!-- DOCTYPE and other detail omitted -->
<session-factory>
  <!-- database connection settings -->
  <property name="hibernate.connection.username">guest</property>
  <property name="hibernate.connection.password">password</property>
  <property name="hibernate.connection.url">
    jdbc:postgres://localhost/db</property>
  <property name="hibernate.connection.driver_class">
    org.postgresql.Driver</property>

  <!-- SQL dialect -->
  <property name="hibernate.dialect">
    org.hibernate.dialect.PostgreSQLDialect</property>

  <!-- connection pool settings (uses built-in connection pool) -->
  <property name="hibernate.connection.pool_size">1</property>
</session-factory>
```

20

## The Configuration Class

- ◆ A **Configuration** object is used to specify the configuration files that are read to configure Hibernate
- ◆ You usually create one instance of **Configuration** and one instance of **SessionFactory** per application, usually via the following **Configuration** methods:
  - A **constructor** configures Hibernate via ***hibernate.properties***
  - **configure()** configures Hibernate via ***hibernate.cfg.xml***
    - The constructor and **configure()** methods are overloaded, allowing you to specify the configuration file name and location
  - **buildSessionFactory()** creates a **SessionFactory**
- You can also configure Hibernate programmatically with a **Configuration** instance; see the Javadoc for more info

21

## The SessionFactory Interface

- ◆ A **SessionFactory** creates sessions
  - Its behavior is controlled by configuration properties
  - Usually an application has a single **SessionFactory**
  - **A SessionFactory is immutable**
  - It is an expensive-to-create, threadsafe object intended to be shared by all application threads
  - It is created once, usually on application startup, from a **Configuration** object
    - This is often done in a utility class and then retrieved via that same class
- ◆ It is a global factory responsible for a particular database
  - Its configuration defines the database and the entities that can be managed by a given **Session** instance

22

## SessionFactory API

- ◆ Here are some common methods of `SessionFactory`
  - **`openSession()`** - Create a database connection and open a session on it
  - **`openSession(Connection connection)`**: Open a session on the given connection
    - For applications that manage JDBC connections themselves
  - **`getCurrentSession()`** – Get the current session
    - "Current session" is used in container-managed sessions
  - **`void close()`**: Destroy this `SessionFactory` and release all its resources (caches, connection pools, etc).
  
- ◆ Many more methods – see the documentation

23

## The Session Interface

- ◆ **`Session`** is the main runtime interface used with Hibernate
  - It abstracts the notion of a persistence service
  - An inexpensive, non-threadsafe object that should be used once, for a single request, or single unit of work, and then discarded
  - Will not obtain a JDBC Connection (or a Datasource) unless it is needed, hence consumes no resources until used
  - Generally, you keep the session around for the duration of a logical transaction
  
- ◆ `Session` has a number of capabilities
  - **Loading**: retrieving an instance from persistent storage
  - **Saving**: persisting objects to persistent storage
  - **Deleting**: removing objects from persistent storage
  - **Updating**: updating the persistent data of an entity

24

## Sessions and Transactions

- ◆ By default, work done with a session is not persisted to the database until the associated transaction commits
  - Objects that are saved or updated are written to a transaction cache for that session
  - When the transaction commits, the objects are written to the DB
- ◆ You can work with transactions through the Hibernate API
  - You start a transaction and obtain an **org.hibernate.Transaction** object from a session  
**Transaction tx = s.beginTransaction();**
  - You can commit or rollback via the transaction object  
**tx.commit();**

25


## A Basic Hibernate Program Example

- ◆ We start simply, and we'll cover more of the details later
  - We basically read the configuration and connect to the DB here

```
import org.hibernate.*;
import org.hibernate.cfg.Configuration;

public class TestHibernate {
    public static void main(String[] args) {
        SessionFactory sf;
        Session s;
        try {
            sf = new Configuration().configure().buildSessionFactory();
            s = sf.openSession();
            s.beginTransaction();
            System.out.println("Connected: " + s.isConnected());
            s.getTransaction().commit();
            s.close();
            sf.close();
        } catch (HibernateException e) { e.printStackTrace(); }
    }
}
```

26



**Mapping a Simple Class**

- Hibernate Overview
- Using Hibernate
- Mapping a Simple Class**
- Inserting, Updating, and Deleting
- Querying and HQL

### Persistent Entity Classes

- ◆ A persistent entity class is a **lightweight persistent domain object**
  - They are fine-grained objects representing state stored in a DB
  - Entity classes are one of the primary programming artifacts of Hibernate
- ◆ Entity classes must be persistable, i.e., have a database representation
  - Have a persistent identity – basically the primary key in the DB
  - Are normally created/updated/deleted within a transaction
- ◆ Metadata describes the mapping to the data store
  - Can be done with XML or annotations (new in Hibernate 3)
  - The metadata required is minimized through the use of intelligent defaults

28

## Persistent Classes

- ◆ Persistent classes work best if they follow a few simple rules
  - Implement a **no argument constructor** for Hibernate to instantiate instances through reflection (required)
  - Declare **getter/setter methods** for all persistent fields (optional)
  - Provide an **identifier property** (usually called id) (optional)
    - Maps to primary key column of database
    - Theoretically optional, but required for use of the full feature set of Hibernate
  - Implement **equals()** and **hashCode()** (optional)
    - Necessary if you mix instances created in different sessions
  - If instances will be passed as a detached object through a remote interface, the class must implement **Serializable**

29

## An Example Persistent Class

```
package com.javatunes;

import java.util.Date;

public class Event {
    private Long id; // property for database id
    private String title;
    private Date date;

    public Event() { }

    public Long getId() { return id; }
    private void setId(Long id) { this.id = id; }

    public Date getDate() { return date; }
    public void setDate(Date date) { this.date = date; }

    public String getTitle() { return title; }
    public void setTitle(String title) { this.title = title; }
}
```

30

## The Event Class

- ◆ The Event class has three properties
  - **id** (Long), **date** (java.util.Date), **title** (String)
  - The id property holds a unique identifier for each event
  - The class has JavaBean style get/set methods for all the properties
  - The class also has no-argument constructor
  
- ◆ **By default, no properties are persistent**
  - That is, they are **not** stored in the database
  - A mapping file is used to specify persistent fields

31

## The id Property

- ◆ The id property will hold the primary key of the entity
  - A class must have a primary key, and it is highly recommended that you define a property to hold it
  
- ◆ A simple primary key must be one of the following:
  - Java primitive or wrapper class (integral types most common)
  - java.lang.String, java.util.Date, java.sql.Date
  
- ◆ Generated primary keys are supported
  - Only integral types will be portable
  
- ◆ Composite primary keys are also possible
  - These use a primary key class

32

## The Hibernate Mapping File

- ◆ Specifies the O-R mapping for your entity classes
  - Here is the top-level structure of the mapping file
  - This declares a mapping for the Event class, with a mapping to the EVENTS table
  - The name of the mapping file is generally named after the class (e.g., **Event.hbm.xml** for the **Event** class)

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
  <class name="com.javatunes.Event" table="EVENTS">
    <!-- property mappings ... -->
  </class>
</hibernate-mapping>
```

33

## The EVENTS Table

- ◆ Let's assume that the EVENTS table is declared as below
- ◆ Assume you are using a generated primary key
  - A very common situation
  - The SQL shown is for the open source Derby database using an Identity column
- ◆ Let's look at how to map our Event class

```
CREATE TABLE EVENTS
(
  EVENT_ID      BIGINT NOT NULL GENERATED ALWAYS AS IDENTITY
                (START WITH 1, INCREMENT BY 1),
  EVENT_DATE    DATE,
  TITLE         VARCHAR(80),
  CONSTRAINT    PK_EVENTS PRIMARY KEY(EVENT_ID)
);
```

34

## The O-R Mapping – Illustrated

- ◆ Mapping of the **EVENTS** table to the **Event** class
  - An **EVENTS** table *row* becomes an **Event** *object*

Event class

```
Long    id;
String  title;
Date    date;
```

Event object

```
id      10
title   Wedding
date    2002-02-27
```

EVENTS

<u>BIGINT</u> EVENT_ID (PK)	<u>DATE</u> EVENT_DATE	<u>VARCHAR</u> TITLE
1	2008-07-15	Hibernate Training Session
...	...	...
<b>10</b>	<b>2002-02-27</b>	<b>Wedding</b>

35

## Mapping the id Property with <id>

- ◆ The **<id>** element declares the primary key in the database
  - The **name** attribute specifies the property it is mapped to (id)
  - The **column** attribute maps this to the **EVENTS** table's primary key column (EVENT\_ID)
  - We also configure Hibernate to automatically generate the primary key value for us

```
<hibernate-mapping>
  <class name="com.javatunes.Event" table="EVENTS">
    <id name="id" column="EVENT_ID">
      <generator class="identity"/>
    </id>
  </class>
</hibernate-mapping>
```

36

## Generating the id Value

- ◆ The **<generator>** element specifies a generated key
  - There are many options for generating this primary key, including letting the database or Hibernate generate it
  - **increment**: Identities unique only when no other process is inserting into the table (not for cluster)
  - **identity**: Uses identity columns for databases supporting it
  - **sequence**: Uses a sequence for databases supporting it
  - **assigned**: lets application assign value
    - Often used for natural keys
  - More are available; see the documentation

37

## Mapping Properties with **<property>**

- ◆ The **<property>** element declares persistent properties
  - By default, no properties are considered persistent
  - The **name** attribute specifies the property name in the class
  - The **column** attribute specifies the column name in the database
    - Here we map the **date** property to the **EVENT\_DATE** column
  - The **type** attribute specifies the Hibernate type of the property
  - The **title** property will map to the **TITLE** column by default; the type will be determined by Hibernate

```
<hibernate-mapping>
  <class name="com.javatunes.Event" table="EVENTS">
    <id name="id" column="EVENT_ID">
      <generator class="increment"/>
    </id>
    <property name="date" type="date" column="EVENT_DATE"/>
    <property name="title"/>
  </class>
</hibernate-mapping>
```

38

## Hibernate Mapping Types

- ◆ The type of the date property is specified as a Hibernate mapping type
  - These are converters which can translate from Java to SQL data and vice-versa
  - If this attribute is not specified, Hibernate will try to determine the correct type mapping
    - As long as the types can be converted, Hibernate can handle this
  - In the case of a `java.util.Date`, Hibernate can't tell if it should map to a SQL DATE, TIMESTAMP, or TIME column, so we specified it in the mapping
  - The title property (`String`) maps to the TITLE column (VARCHAR), and the defaults work for this

39

## Common Hibernate Type Mappings

- ◆ **integer, long, short, float, double, character, byte, boolean, yes\_no, true\_false**
  - Type mappings from Java primitives or wrapper classes to appropriate (vendor-specific) SQL column types
  - `boolean`, `yes_no` and `true_false` are all alternative encodings for a Java `boolean` or `java.lang.Boolean`
- ◆ **string**
  - `java.lang.String` to VARCHAR (or Oracle VARCHAR2)
- ◆ **date, time, timestamp**
  - `java.util.Date` to SQL types DATE, TIME and TIMESTAMP
- ◆ **calendar, calendar\_date**
  - Type mappings from `java.util.Calendar` to SQL types TIMESTAMP and DATE (or equivalent)
- ◆ **big\_decimal**
  - `java.math.BigDecimal` to NUMERIC (or Oracle NUMBER)

40

## The Mapping File

- ◆ The file would normally be called *Event.hbm.xml*
  - It can be placed in the same directory as the *Event.class* file
  - This, of course, should be on the classpath
- ◆ You specify that the mapping exists by including a reference to it in the **hibernate.cfg.xml** file, as shown below

```
<!-- DOCTYPE and other detail omitted -->
<session-factory>
  <!-- session factory properties, detail omitted -->

  <!-- mapping files -->
  <mapping resource="com/javatunes/Event.hbm.xml"/>
</session-factory>
```

41

## Hibernate Sessions

- ◆ A **session** is required to actually interact with the database, to create, read, or write a mapped entity
  - Without a session, your types are just POJOs
- ◆ The session also **manages** entity instances
  - When the session obtains an entity reference, the referenced object becomes a **persistent** (or **managed**) entity
- ◆ The **Session** interface defines the methods used for working with entities
  - It is used to create and remove persistent entity instances, to find entities by their primary key, and to query over entities

42

## Retrieving Persistent Objects

- ◆ Use the `Session.get()` method to retrieve persistent objects
  - `get()` requires that you pass in the id of the entity you want
  - It returns the persistent instance of the given entity class with the given identifier, or null if no instance with the id is found

```
// HibernateUtil is an application class for getting the factory
SessionFactory factory = HibernateUtil.getSessionFactory();
Session s = factory.openSession();
Long id = new Long(1);

Event e = (Event) s.get(Event.class, id);

System.out.println("Retrieved event: " + e.getTitle());
s.close();
```

43



## Inserting, Updating, and Deleting

Hibernate Overview  
Using Hibernate  
Mapping a Simple Class  
**Inserting, Updating, and Deleting**  
Querying and HQL

## Inserting Instances

- ◆ It's very easy to insert new instances (rows) into the DB
  - Simply create a new instance using *new*
  - This will be a **transient** instance (one that has no database representation)
  - Set values for the properties (except for the ID property)
  - Save the instance to the DB using a Hibernate session
  
- ◆ Use the **Session.save()** method to persist the instance
  
- ◆ The instance will be inserted into the DB
  - The id value is returned
  - When the transaction completes successfully, the insert will be made permanent

45

## Inserting Instance Example

- ◆ Here's an example of persisting a new Event instance
  - When the instance is first created, it is unknown to the session
    - It is called a **transient** instance
  - Once it is saved, it is a **managed** instance in the session's scope
  - Once the transaction commits, the insert is permanent

```
// code related to Session, SessionFactory, etc. omitted
Session s = ...

Event e = new Event();
e.setDate(new Date());
e.setTitle("Hibernate Training Session");

Transaction tx = s.beginTransaction();
Long newID = (Long) s.save(e);
tx.commit();

System.out.println("Saved Event with ID: " + newID);
s.close();
```

46

## Modifying a Persistent Instance

- ◆ If you have a persistent instance (one associated with an active session) you can just update that instance
  - Hibernate monitors changes to persistent instances
    - This is called **automatic dirty checking**
  - The changes will be persisted when the tx commits

```
// code related to Session, SessionFactory, etc. omitted
Session s = ...
Long id = new Long(1);

Transaction tx = s.beginTransaction();
Event e = (Event) s.get(Event.class, id);
// let's say you need to change the title
e.setTitle("Hibernate Really Rocks");
// no need to call a session method to persist this change
tx.commit();

s.close();
```

47

## Deleting an Instance


- ◆ It's also very easy to delete an instance from the database
  - The instance has to be in the session scope
  - You can then call **Session.delete()** on the instance
  - When the transaction commits, the row will be deleted

```
// code related to Session, SessionFactory, etc. omitted
Session s = ...
Long id = new Long(1);

Transaction tx = s.beginTransaction();
Event e = (Event) s.get(Event.class, id);
s.delete(e);
tx.commit();

s.close();
```

48



**LearningPatterns**

## Querying and Hibernate Query Language (HQL)

- Hibernate Overview
- Using Hibernate
- Mapping a Simple Class
- Inserting, Updating, and Deleting
- Querying and HQL**

### Hibernate Query Language

- ◆ Hibernate has an OO query language called **Hibernate Query Language (HQL)**
  - Similar to SQL in syntax and structure
  - Leverages knowledge about SQL
  - If you don't know the identifiers of the objects you are looking for, you need a query since `Session.get()` requires an id
- ◆ Designed to query **object graphs**, rather than relational tables
  - Fully object-oriented
  - Supports inheritance and polymorphism
- ◆ Structure is similar to SQL
  - SELECT, FROM, and WHERE clauses
  - Can use lower or upper case for keywords

50

## HQL Basics

- ◆ Here's an example of the most basic query you can make

### FROM Event

- This query will return all the Event instances in the database
  - This query looks similar to SQL
  - The key difference is that you are making object based queries
  - **You are selecting from an *entity*, not a *table* (Event)**
  - Note that the SELECT clause is not always required
    - Since we are returning complete objects, and only have one potential object type in the query, the SELECT is not needed
- ◆ Here are equivalent queries
    - **FROM Event e**: Defines alias for Event entity
    - **SELECT e FROM Event e**: Explicitly states the SELECT clause

51

## Creating a Query

- ◆ Hibernate provides a **Query** interface for executing queries
  - You obtain a Query instance from an active session

```
Query q = s.createQuery("FROM Event");
```

- This interface is an OO representation of a query
- It has methods to:
  - Set query parameters
  - Execute a query
  - Set paging parameters on a query
  - Get metadata about a query
  - And more
- A query instance may be executed multiple times
  - Its lifespan is bounded by the lifespan of the session that created it

52

## Executing a Query

- ◆ Once you have a Query object you can use it to retrieve persistent instances with the following methods
  - **List list()**: Return the results as a list
    - Result objects with their properties loaded into memory
    - Most common way of using a query
  - **Object uniqueResult()**: Convenience method to return a single instance that matches the query (null if no match)

53

## Executing a Query – Example

- ◆ Here we use the `list()` method and then iterate through the returned objects – this is the most common way to query

```
Query q = s.createQuery("FROM Event");
List l = q.list();
Iterator i = l.iterator();
while (i.hasNext()) {
    Event e = (Event) i.next();
    System.out.println("Event: " + e.getId());
}
```

- ◆ The `list()` method executes the query, retrieving all the entities into memory at once, and returns the result as a `java.util.List`
  - We can then iterate through the list as we normally do
  - Control fetching via `setFetchSize()` and `setMaxResults()`

54

## WHERE Clause/Restriction

- ◆ Of course, we can also provide constraints for a query in a **WHERE** clause

```
FROM Event WHERE id > 10
```

- This query does what you expect it to do
  - It returns all Event instances with an id greater than 10
- ◆ HQL supports the same basic operators as SQL

55

## Query Parameters

- ◆ Hibernate allows you to specify and populate **query parameters** in a way similar to JDBC prepared statements

- **Named parameters**, which are not available in JDBC, are preferred because they:

- Are insensitive to the order they occur in the query string
- May occur multiple times in the same query
- Are self-documenting

```
FROM Event WHERE id > :id
```

- **Positional parameters** also available:

- Cumbersome, but familiar to JDBC programmers
- **NOTE: numbering starts at 0, not at 1 like JDBC**

```
FROM Event WHERE id > ?
```

56

## Using Query Parameters

- ◆ The Query interface has an extensive set of methods to set the query parameters for all supported parameter types
  - setFloat, setInteger, setString, setDate, ...
- ◆ You can also use the generic setParameter method
  - Here's an example using named parameters (preferred)

```
Query q = s.createQuery("FROM Event WHERE id > :id");
q.setLong("id", 10); // or q.setParameter("id", new Long(10))
List l = q.list();
```

- Here's an example using positional parameters

```
Query q = s.createQuery("FROM Event WHERE id > ?");
q.setLong(0, 10); // or q.setParameter(0, new Long(10))
List l = q.list();
```

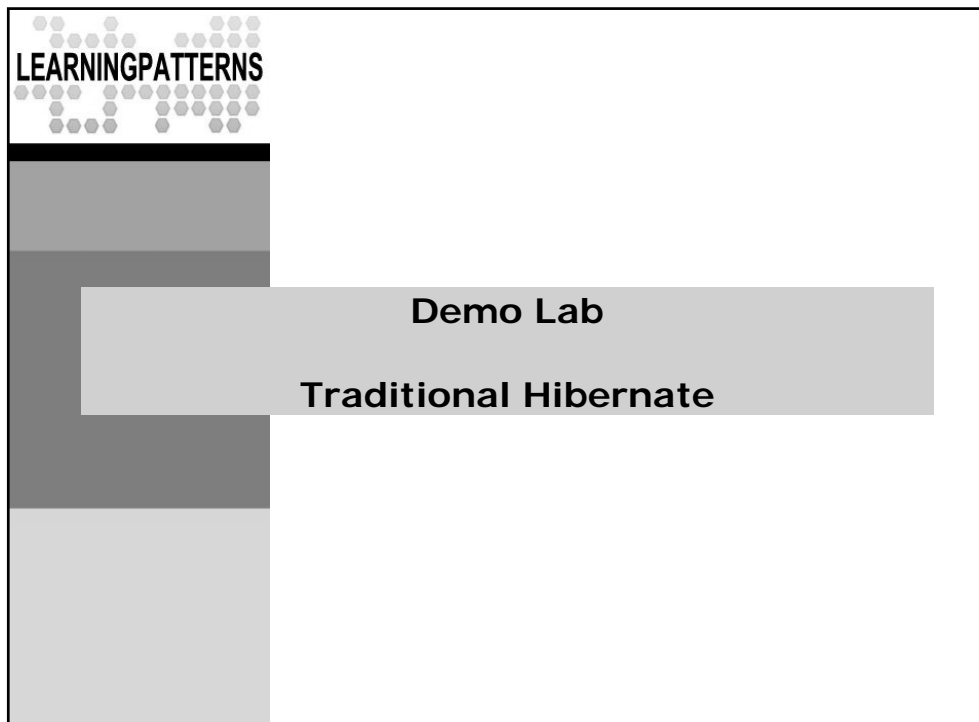
57


## Named Queries

- ◆ Instead of scattering HQL throughout your code, you can use named queries
  - These are predefined in Hibernate mapping files and then retrieved via the session object
    - Query q = s.getNamedQuery("Event.beforeDate");**
- ◆ Named queries can also yield efficiency gains
  - They can be parsed, prepared, and cached when the mapping file is processed

```
<hibernate-mapping>
  <class>...</class>
  <query name="Event.beforeDate">
    <![CDATA[FROM Event e WHERE e.date <= :date]]>
  </query>
</hibernate-mapping>
```

58



**Overview** 

- ◆ In this lab, we will explore how Hibernate is used to implement Object-Relational mapping in Java
- ◆ Some of the points to notice will be
  - How entity classes are mapped to the database
  - How the Session is used to work with the persistent objects
  - How queries are done using HQL
- ◆ **Lab Directory: workspace\Hibernate**

60

## Start the Database



- ◆ We are using the open source Derby database
  - We will set up and run this database using a number of scripts supplied in the setup in **C:\StudentWork\PJUG\Software\Derby**

### Tasks to Perform

- ◆ Open the **StudentWork\Hibernate\Derby\dbSetenv.cmd** file
  - Look at how DB\_NAME is set (you don't need to change this):  
**set DB\_NAME=jdbc:derby://localhost:1527/JavaTunesDB**
  - This sets the server that our tools will talk to
    - The server on localhost listening on the standard Derby server port
    - Using the database with the name JavaTunesDB
- ◆ Start the Derby database server
  - Execute **dbStart.cmd** (you can double-click on it)
  - This starts a standalone Derby server that can accept network connections to the database

```

C:\Derby Server
Security manager installed using the Basic server security policy.
Apache Derby Network Server - 10.3.1.4 - (561794) started and ready to accept co
nnections on port 1527 at 2008-01-29 16:25:40.890 GMT
    
```

61

## Creating and Browsing the Database



### Tasks to Perform

- ◆ Create the database
  - Execute **dbCreate.cmd** (you can double-click on it)
    - This creates the JavaTunesDB database
    - You can ignore a DROP TABLE error, if you see it
- ◆ Execute **dbSQL.cmd** to run ij (you can double-click on it)
  - This command file is set up to connect to the correct database
  - Using **ij**, browse the DB to check the setup (sample SQL below)

```

ij> select * from item;
... Lots of output ...
ij> exit;
    
```

- ◆ **ij** is Derby's SQL command line tool
  - It allows us to create database objects and view and manipulate database data, without having to write code to do so
  - Most database packages provide such a tool

62

## The MusicItem Class

- ◆ The labs are centered around persisting the **MusicItem** class, defined below (only get methods shown)
  - It has an id property and a number of value properties

```
public class MusicItem
implements java.io.Serializable {
    public Long getId() ...
    public String getNum() ...
    public String getTitle() ...
    public String getArtist() ...
    public Date getReleaseDate() ...
    public BigDecimal getListPrice() ...
    public BigDecimal getPrice() ...
}
```

63

## The Database Table for MusicItem



- ◆ The database table for **MusicItem** is defined as shown below
  - Notice that the table name is **ITEM**
  - Notice that the ID (in the **ITEM\_ID** column) is an Identity column
  - Note the names and types of the other columns

```
CREATE TABLE ITEM
(
    ITEM_ID          BIGINT NOT NULL GENERATED ALWAYS AS IDENTITY
                    (START WITH 1, INCREMENT BY 1),
    ITEM_NUM         VARCHAR(10) NOT NULL,
    TITLE            VARCHAR(40),
    ARTIST           VARCHAR(40),
    RELEASE_DATE     DATE,
    LIST_PRICE       DECIMAL(5,2),
    PRICE            DECIMAL(5,2),
    CONSTRAINT PK_ITEM PRIMARY KEY (ITEM_ID),
    CONSTRAINT UNQ_ITEM_NUM UNIQUE (NUM)
);
```

64

**Lab**

## The O-R Mapping

- ◆ Mapping of the **ITEM** table to the **MusicItem** class
  - An **ITEM** table row becomes a **MusicItem** object

MusicItem class

```

Long      id;
String    num;
String    title;
String    artist;
Date      releaseDate;
BigDecimal listPrice;
BigDecimal price;
                    
```

MusicItem object

```

id        21
num       CD521
title     Music
artist    Madonna
releaseDate 2002-02-27
listPrice 14.99
price     11.97
                    
```

ITEM

BIGINT ITEM_ID (PK)	VARCHAR ITEM_NUM	VARCHAR TITLE	VARCHAR ARTIST	DATE RELEASE_DATE	DECIMAL LIST_PRICE	DECIMAL PRICE
1	CD501	Diva	Annie Lennox	1992-01-04	17.97	13.99
...	...	...	...	...	...	...
21	CD521	Music	Madonna	2002-02-27	14.99	11.97

65

**Lab**

## The Configuration and Mapping Files

**Tasks to Perform**

- ◆ Open the hibernate.cfg.xml file and review it
- ◆ Open the MusicItem.java file and its associated mapping file, MusicItem.hbm.xml
  - Spend some time looking at the mapping
- ◆ Also take a look at HibernateUtil.java, which manages the creation of the session factory

66

## The Test Client



### Tasks to Perform

- ◆ Open the MusicItemTest.java file
- ◆ Review the code there, which does the following things:
  - Retrieves a MusicItem via its id
  - Updates an existing MusicItem
  - Inserts a new MusicItem
  - Deletes an existing MusicItem
  - Queries for the MusicItem with a given item number
  - Queries for all MusicItems with a list price less than a given price
- ◆ There is no SQL in any of this - it is all Java, and simple Java
- ◆ This is the power of Hibernate



67




## Part 2 Hibernate with the Java Persistence API (JPA)

JPA Overview  
Mapping a Simple Class  
Entity Manager and Persistence Context  
Inserting, Updating, and Deleting  
Querying and JPQL

## Objectives

- ◆ Describe the goals of the **Java Persistence API (JPA)**, and its architecture
- ◆ Be familiar with how to **map entity classes to a database** with JPA
- ◆ Learn how to retrieve, insert, update, and delete JPA persistent objects
- ◆ Be familiar with querying using JPQL

69



**JPA Overview**

- JPA Overview**
- Mapping a Simple Class
- Entity Manager and Persistence Context
- Inserting, Updating, and Deleting
- Querying and JPQL

## Java Persistence API Overview

- ◆ Completely new persistence framework for Java
  - Totally different from previous frameworks
  - Draws on technology from Hibernate, Toplink, JDO, and others
  - Replaces EJB Entity Beans
  
- ◆ Overall goals (same as Hibernate)
  - Provide **ORM capability** for using a **Java OO domain model** to manage a relational database
  - Provide a **light-weight POJO based framework** for Java persistence, including **inheritance** and **polymorphism**
  - Provide a **query language** (an extension of EJB QL) that helps **remove or encapsulate vendor specific SQL** code
  - **Relieve the developer from 95%** of common data persistence related programming tasks

71

## JPA Benefits

- ◆ Simplicity and flexibility
  - POJO based!
  - ORM is completely metadata driven (**annotations** or XML)
  - No need to write JDBC code
  - Common defaults that allow metadata to be minimized
  - Persistence API is totally separate from entity classes
  - Can be used in Java SE environments (without app server)
  
- ◆ Completeness
  - Supports full range of OO features
    - Inheritance, custom types, collections, associations
  - Powerful query language
  
- ◆ Performance
  - Minimizes number of database interactions
  - Object caching

72

## Java Persistence Environments


- ◆ Java Persistence is a required part of EJB 3.0
  - It is also a required part of Java EE 5
  - Its types are in the **javax.persistence** package
  
- ◆ Java Persistence can also be used in **Java SE programs**
  - It is packaged as a set of jar files
  - There are some minor configuration and programming differences from using it in a Java EE environment
  
- ◆ Application server support is growing
  - **JBoss**    **Hibernate based**
  - Oracle    Toplink based
  - Sun        Toplink based
  - BEA        Kodo based

73

## Hibernate and JPA

- ◆ Hibernate and JPA basically do the same thing
  - The main difference is the use of standardized annotations and APIs in JPA vs. Hibernate-specific XML mapping files and APIs
  
- ◆ Hibernate supports JPA via two additional Hibernate projects
  - Hibernate Annotations + Hibernate EntityManager
  - These are built on top of Hibernate Core and add a few libraries
  
- ◆ Hibernate Annotations can also be used independent of JPA
  - `org.hibernate.annotations` package
  - You're probably better off using JPA instead (standardized)
  
- ◆ In this section, we will show you how to use Hibernate via JPA
  - We'll point out the similarities between the two (many)
  - As well as the differences (few)

74



**Mapping a Simple Class**

JPA Overview  
**Mapping a Simple Class**  
Entity Manager and Persistence Context  
Inserting, Updating, and Deleting  
Querying and JPQL

## Entity Classes

- ◆ An entity is a **lightweight persistent domain object**
  - Entities are fine-grained objects representing state stored in a DB
  - An entity is one of the primary programming artifacts of JPA
- ◆ Entities
  - Must be persistable, i.e., have a database representation
  - Have a persistent identity – basically the primary key in the DB
  - Are normally created/updated/deleted within a transaction
- ◆ Entity metadata describes the mapping to the data store
  - Can be done with **annotations** (preferred) or XML
  - The metadata required is minimized through the use of intelligent defaults

76

## Entity Class Requirements

- ◆ Implement a no-argument ctor, with other ctors allowed
- ◆ Contain instance variables to hold the persistent state
  - These must **NOT** be declared public
  - Clients of the entity must use accessor (getter/setter) methods for all persistent fields
- ◆ Provide an identifier property (usually a property called id)
  - Maps to primary key in database
  - Primitive type, or primitive wrapper, String, Date, composite key
- ◆ If instances will be passed as a detached object (e.g., through a remote interface), it must implement `Serializable`

77

## An Example Entity Class with JPA

```
package com.javatunes;

import java.sql.Date;
import javax.persistence.Entity;
import javax.persistence.Id;

@Entity // indicates this is an entity class
public class Event implements java.io.Serializable {
    @Id // designates ID field
    private Long id;
    private String title;
    private Date date;

    public Event() { }

    public Long getId() { return id; }
    private void setId(Long id) { this.id = id; }

    public Date getDate() { return date; }
    public void setDate(Date date) { this.date = date; }

    public String getTitle() { return title; }
    public void setTitle(String title) { this.title = title; }
}
```

78

## The Entity Declaration

- ◆ **@Entity** declares the class to be a persistent entity
- ◆ By default, it will be mapped to a table called **Event**
  - You can use **@Table** to declare a different table name
  - For example, if the table was called EVENTS you could use:

```
@Entity
@Table(name="EVENTS")
public class Event { /* ... */ }
```

79

## The Event Class

- ◆ The Event class has three properties
  - **id** (Long), **date** (java.sql.Date), **title** (String)
  - The id property holds a unique identifier for each event
  - Has JavaBean style get/set methods for all the properties
  - Also has no-argument constructor
- ◆ By default, **all non-transient properties are persistent**
  - **This is different from Hibernate**
  - By default, they are stored in a column with the same name as the property (same as Hibernate)
  - The column type in the database also uses reasonable defaults
- ◆ Non-persistent properties can be annotated with **@Transient**
  - Fields that use the Java transient modifier are also not persisted

80

## The Id Property

- ◆ **@Id** denotes that this property holds the primary key
  - A class must have a primary key
- ◆ A simple primary key must be one of the following:
  - Java primitive or wrapper class (integral types most common)
  - `java.lang.String`, `java.util.Date`, `java.sql.Date`
- ◆ Generated primary keys are supported
  - Only integral types will be portable
- ◆ Composite primary keys are also possible
  - These use a primary key class

81

## Generated Id Property

- ◆ You use **@GeneratedValue** to specify a generation strategy for primary keys
  - Possible strategies are:
    - **AUTO**: Persistence provider picks best strategy for DB
    - **IDENTITY**: Uses DB identity column
    - **SEQUENCE**: Uses DB sequence column
    - **TABLE**: Uses underlying database table
- ◆ You use **@Column** to specify the column name (if necessary)

```
@Id
@GeneratedValue(strategy= GenerationType.IDENTITY)
@Column(name="EVENT_ID")
private Long id;
```

82

## The EVENTS Table

- ◆ Let's assume that the EVENTS table is declared as below
- ◆ Assume you are using a generated primary key
  - A very common situation
  - The SQL shown is for the open source Derby database using an Identity column
- ◆ We'll look at how to map our Event class based on this table

```
CREATE TABLE EVENTS
(
  EVENT_ID      BIGINT NOT NULL GENERATED ALWAYS AS IDENTITY
                (START WITH 1, INCREMENT BY 1),
  EVENT_DATE    DATE,
  TITLE         VARCHAR(80),
  CONSTRAINT    PK_EVENTS PRIMARY KEY(EVENT_ID)
);
```

83

## Mapping Properties

- ◆ By default, all properties are considered persistent
  - However, all the defaults may not be appropriate for us
  - In our example, we must map the date property to the EVENT\_DATE column
  - The title property will map to the TITLE column by default, so we don't need to do anything

```
@Entity
@Table(name="EVENTS")
public class Event implements java.io.Serializable {
  @Id
  @GeneratedValue(strategy=GenerationType.IDENTITY)
  @Column(name="EVENT_ID")
  private Long id;
  private String title;
  @Column(name="EVENT_DATE")
  private Date date;
  // other code omitted ...
}
```

84

## Field Access or Property Access

- ◆ In the Event class, the persistence runtime will access the persistent fields directly in our event class
  - This is because we have annotated the persistent fields
  - It is also possible to annotate the accessor (getter) methods
    - In that case, the runtime will access the properties via the accessors
  - This may be desirable if you have business logic in the accessor methods (e.g., validation logic)
  
- ◆ You should not mix field and accessor styles within a class
  - This is not portable

```
@Id // property access is used
public Long getId() { return id; }
```

85

## Basic Mapping Types

- ◆ JPA can map a large number of persistable types
  - It automatically maps them to a JDBC type in the DB
  - If the type in the DB is different, it will do its best to convert it
  - The persistent fields or properties of an entity may be of the following types:
    - Java primitive types and wrappers of the primitive types
    - `java.lang.String`
    - `java.math.BigInteger`, `java.math.BigDecimal`
    - `java.util.Date`, `java.util.Calendar`  
`java.sql.Date`, `java.sql.Time`, `java.sql.Timestamp`
    - `byte[]`, `Byte[]`, `char[]`, and `Character[]`
    - enums
    - User-defined serializable types

86

## Persisting to the Database

- ◆ An **entity manager** is used to persist to the database
  - It is represented by the **javax.persistence.EntityManager** interface
  - This interface encapsulates the API for persisting to the database
  - **Analogous to the Hibernate Session**
- ◆ The entity manager API is completely separate from the mapping definition of an entity class
  - This allows for a much cleaner definition of an entity class

87



## Entity Manager and Persistence Context

JPA Overview  
Mapping a Simple Class  
**Entity Manager and Persistence Context**  
Inserting, Updating, and Deleting  
Querying and JPQL

## The Entity Manager and Persistence Context

- ◆ An **entity manager** is required to interact with a database, to create, read, or write an entity
  - An entity manager is configured to work with a specific data store
- ◆ The entity manager also **manages** entity instances
  - When the entity manager obtains an entity reference, the referenced object becomes a **managed** entity
- ◆ An **EntityManager** is associated with a **persistent context**
  - A persistence context is a set of managed entity instances
  - **Only one instance** with a given persistent identity can exist within a persistence context (just like in Hibernate)
    - If you get an event with id=25 twice from a persistent context, the 2<sup>nd</sup> retrieval returns the same instance as the 1<sup>st</sup>

89

## The EntityManager Interface

- ◆ An **EntityManager** is used to interact with the persistent context
  - The **EntityManager** interface is used to create and remove persistent entity instances, to find entities by their primary key, and to query over entities
  - It defines the methods that are used to interact with the persistence context
  - We use the JPA **EntityTransaction** interface this time instead of the Hibernate **Transaction**
- ◆ Some of the important **EntityManager** methods are:
  - `<T> T find(Class<T> entityClass, Object pk)`
  - `void persist(Object entity)`
  - `void remove(Object entity)`

90

## Persistence Unit

- ◆ A **persistence unit** defines the configuration of an **EntityManagerFactory**
  - Persistence unit is configured in *META-INF/persistence.xml*
  - **Persistence unit is analogous to the Hibernate Configuration (configured in *hibernate.cfg.xml*)**
  - **EntityManagerFactory is analogous to the Hibernate SessionFactory**

```
<persistence version='1.0' ... XMLSchema declarations omitted ...>
  <persistence-unit name="events">
    <properties>
      <property name="hibernate.connection.username" value="sa"/>
      <property name="hibernate.connection.password" value="pswd"/>
      <!-- other properties not shown -->
    </properties>
  </persistence-unit>
</persistence>
```

91

## Injecting an EntityManager in Java EE

- ◆ An **EntityManager** is usually **injected** when used in an EJB
  - Using **@PersistenceContext** and the name of the persistence unit, as shown in the example below ("events")
  - The persistence unit is configured in *META-INF/persistence.xml*
  - The example below uses a stateless session bean façade around the JPA code (very common)

```
@Stateless
public class EventServiceBean {
  @PersistenceContext(unitName="events")
  private EntityManager em;

  public void addEvent(Event event) {
    em.persist(event);
  }
  ...
}
```

92

## Getting an EntityManager in Java SE

- ◆ An `EntityManager` is obtained from the `EntityManagerFactory`, which itself is obtained from the **Persistence** bootstrap class

```
EntityManagerFactory factory =  
    Persistence.createEntityManagerFactory("events");  
EntityManager em = factory.createEntityManager();
```

- ◆ As in Hibernate, the factory is expensive to create, and there is usually only one instance per application
  - A utility class is generally used to instantiate and provide it


93

## Retrieving Persistent Objects

- ◆ You can use the `EntityManager.find()` method to retrieve persistent objects
  - This works just like Hibernate's `Session.get()`
  - Returns the persistent instance of the given entity class with the given identifier, or null if there is no such instance

```
// HibernateUtil is an application class for getting the factory  
EntityManagerFactory factory =  
    HibernateUtil.getEntityManagerFactory();  
EntityManager em = factory.createEntityManager();  
Long id = new Long(1);  
  
Event e = (Event) em.find(Event.class, id);  
  
System.out.println("Retrieved event: " + e.getTitle());  
em.close();
```

94



**Inserting, Updating, and Deleting**

- JPA Overview
- Mapping a Simple Class
- Entity Manager and Persistence Context
- Inserting, Updating, and Deleting**
- Querying and JPQL

### Inserting Instances

- ◆ Inserting into the DB is very similar to Hibernate
  - Simply create a new instance using *new*
  - This will be a **transient** instance (one that has no database representation)
  - Set values for the properties (except for the ID property)
  - Save the instance to the DB using the entity manager
- ◆ Use the **EntityManager.persist()** method to persist the instance
- ◆ The instance will be inserted into the DB
  - When the transaction completes successfully, the insert will be made permanent
  - To get the id value, simply call `getId()` on the instance

96

## Inserting Instance Example

- ◆ Here's an example of persisting a new Event instance
  - When the instance is created, it is unknown to the entity manager
    - It is called a **transient** instance (as it is in Hibernate)
  - Once saved, it is a **managed** instance in the persistence context
  - Once the transaction commits, the insert is permanent

```
// code related to EntityManager, EntityManagerFactory, omitted
EntityManager em = ...

Event e = new Event();
e.setDate(new Date());
e.setTitle("Hibernate Training Session");

EntityTransaction tx = em.getTransaction();
tx.begin();
em.persist(e);
tx.commit();

System.out.println("Saved Event with ID: " + e.getId());
em.close();
```

97

## Modifying a Persistent Instance

- ◆ As with Hibernate, you can just update an existing instance
  - The entity manager monitors changes to persistent instances, just like the Hibernate session does
    - This is called **automatic dirty checking**
  - The changes will be persisted when the tx commits

```
// code related to EntityManager, EntityManagerFactory, omitted
EntityManager em = ...
Long id = new Long(1);

EntityTransaction tx = em.getTransaction();
tx.begin();
Event e = (Event) em.find(Event.class, id);
// let's say you need to change the title
e.setTitle("Hibernate Really Rocks with JPA");
// no need to call an entity manager method to persist this change
tx.commit();

em.close();
```

98

## Deleting an Instance

- ◆ Deleting from the database very similar to Hibernate
  - The instance has to be in the persistence context first
  - You can then call **EntityManager.remove()** on the instance
  - When the transaction commits, the row will be deleted

```
// code related to EntityManager, EntityManagerFactory, omitted
EntityManager em = ...
Long id = new Long(1);

EntityManagerTransaction tx = em.getTransaction();
tx.begin();
Event e = (Event) em.find(Event.class, id);
em.remove(e);
tx.commit();

em.close();
```

99



## Querying and Java Persistence Query Language (JPQL)

JPA Overview  
Mapping a Simple Class  
Entity Manager and Persistence Context  
Inserting, Updating, and Deleting  
**Querying and JPQL**

## Java Persistence Query Language

- ◆ JPA has an OO query language similar to Hibernate's HQL
  - If you know HQL, you basically know JPQL
- ◆ Here's a basic query selecting all Event instances

```
SELECT e from Event e
```

- NOTE that in JPQL, the SELECT clause is required

101

## Creating and Executing Queries

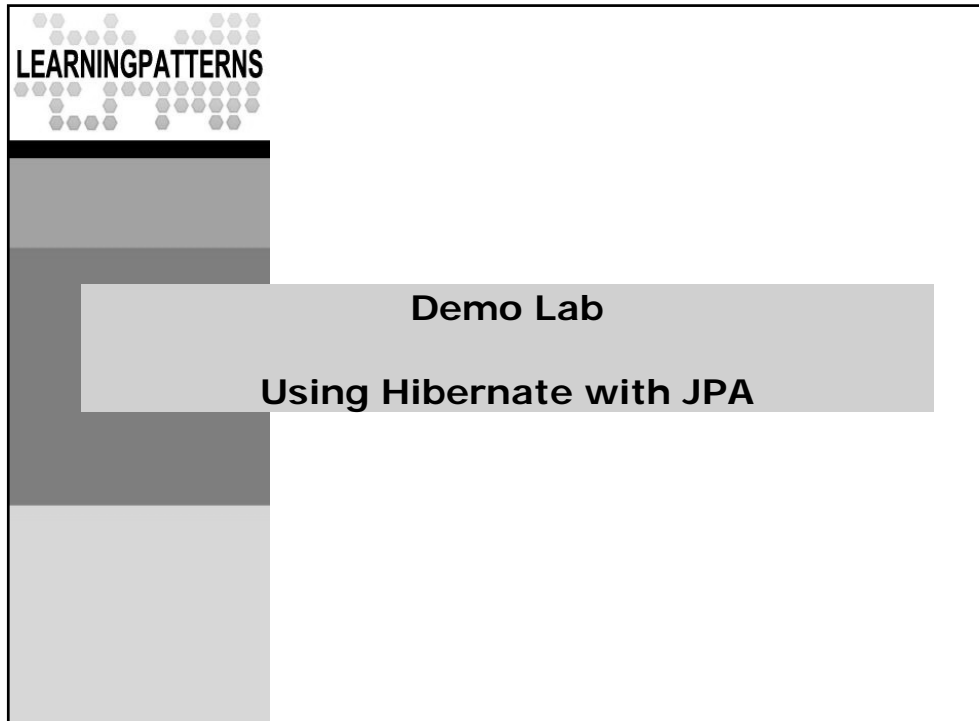
- ◆ JPA provides a **Query** interface analogous to Hibernate's
  - You can create queries or retrieve named queries via the entity manager


```
Query createQuery(String jpql)  
Query createNamedQuery(String name)
```

- ◆ Named queries are defined in entity classes via the `@NamedQuery` annotation
  - Or they can be defined in *META-INF/orm.xml*
- ◆ Once you have the Query object, you can retrieve instances very similarly to Hibernate

```
List getResultList()  
Object getSingleResult()
```

102



**Overview** 

- ◆ In this lab, we will explore how JPA is used to implement Object-Relational mapping
  - With Hibernate "underneath" as the ***JPA provider***
- ◆ Some of the points to notice will be
  - How entity classes are mapped to the database via annotations
  - How the EntityManager is used to work with persistent objects
  - How queries are done using JPQL
- ◆ **Lab Directory: workspace\HibernateJPA**

104

## The Configuration and Mapping



### Tasks to Perform

- ◆ Open the META-INF/persistence.xml file and review it
  - Notice how it contains the same information as hibernate.cfg.xml
- ◆ Open the MusicItem.java file
  - Spend some time looking at how the annotations are used to define the mapping
- ◆ Also take a look at HibernateUtil.java, which manages the creation of the entity manager
  - Notice how similar this is to the utility class we used with straight Hibernate

105

## The Test Client



### Tasks to Perform

- ◆ Open the MusicItemTest.java file
- ◆ Review the code there, which does the same things as in the previous lab
  - Again, there is no SQL in any of this - it is all Java, and simple Java
- ◆ But this time, there are no references to any Hibernate-specific APIs and no separate XML mapping file
  - Our application is portable across different JPA providers
  - The only thing we would need to change to use a different provider would be META-INF/persistence.xml



106