Fast Track to JPA 2
(Java Persistence API V 2)

using Hibernate / Eclipse

LearningPatterns™, Inc. Courseware

LearningPatterns is a trademark of LearningPatterns Inc.

Student Guide

This material is copyrighted by LearningPatterns Inc. This content and shall not be reproduced, edited, or distributed, in hard copy or soft copy format, without express written consent of LearningPatterns Inc. Copyright © 2004-10 LearningPatterns Inc.

For more information about Java Enterprise Java, or related courseware, please contact us. Our courses are available globally for license, customization and/or purchase.

LearningPatterns. Inc. Services@learningpatterns.com | www.learningpatterns.com

Global Courseware Services 55 Wanaque Ave. #188 | Pompton Lakes, NJ 07442 USA
212.487.9064 voice | 201.336.9118 fax

Java, and all Java-based trademarks and logo trademarks are registered trademarks of Sun Microsystems, Inc., in the United States and other countries. LearningPatterns and its logos are trademarks of LearningPatterns Inc. All other products referenced herein are trademarks of their respective holders.

Table of Contents – Fast Track to JPA 2

 Copyright © 2004-10 LearningPatterns Inc. All rights reserved

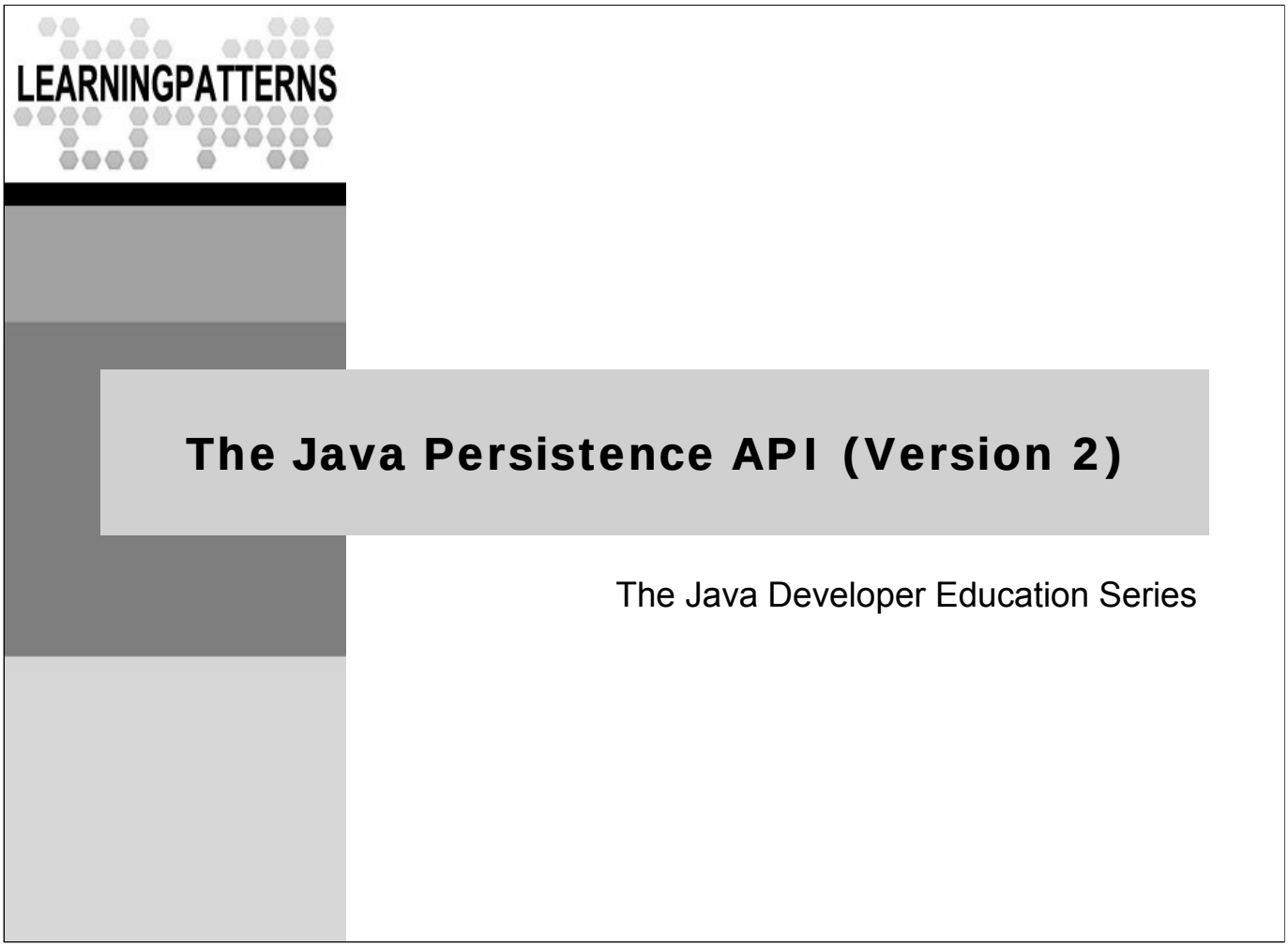
 Copyright © 2004-10 LearningPatterns Inc. All rights reserved

 Copyright © 2004-10 LearningPatterns Inc. All rights reserved

 Copyright © 2004-10 LearningPatterns Inc. All rights reserved

 Copyright © 2004-10 LearningPatterns Inc. All rights reserved

 Copyright © 2004-10 LearningPatterns Inc. All rights reserved



Notes:

- Java and all Java-based trademarks are registered trademarks of Sun Microsystems, Inc

Copyright © 2006-10 LearningPatterns Inc. All rights reserved.

Workshop Overview

- This course provides a thorough introduction to the Java Persistence API Version 2 (JPA) including:
  - The needs that JPA is designed to address
  - The basic concepts and architecture
  - Thorough coverage of the API and details on its use
  - Design principles for correct usage
- The workshop consists of 50% discussion, 50% hands-on lab exercises, including a series of labs designed to exercise all important concepts
  - Most of the labs follow a common fictional case study - JavaTunes, an online music store
    - CDs (Item table), Inventory (Inventory table) and others

Notes:

Copyright © 2006-10 LearningPatterns Inc. All rights reserved.

Workshop Objectives

- At completion you should be able to
  - Understand how JPA relates to the rest of Java
  - Understand JPA concepts and architecture
  - Be familiar with the JPA API
  - Be able to write and use persistent entities
  - Understand important design principles for JPA (and how they relate to other technologies such as Java EE)
  - Be aware of the new capabilities in JPA 2

Notes:

Copyright © 2006-10 LearningPatterns Inc. All rights reserved.

Workshop Agenda

- Session 1: **Java Persistence API Intro**
- Session 2: **Inserts and Queries**
- Session 3: **Lifecycle**
- Session 4: **Associations**
- Session 5: **Criteria Queries**
- Session 6: **JPA Additional Capabilities**
- Session 7: **Integration**

Notes:

Copyright © 2006-10 LearningPatterns Inc. All rights reserved.

Course Prerequisites

- Proficiency in Java and Object-Oriented programming
- Knowledge of relational databases
- Some knowledge of Spring is needed for the integration sessions

Notes:

Copyright © 2006-10 LearningPatterns Inc. All rights reserved.

Release Level

- This course contains instructions for running the labs using the following platforms:
  - **Java 5** or later (tested on Java 5 and 6)
    - Java 5+ is required since JPA depends on Java 5 annotations
  - **Hibernate 3.5.x** (The JPA persistence provider for the labs)
  - **Eclipse Java EE Edition** (any fairly recent version)
  - **JBoss 5.x** (For the optional EJB integration lab)
- All labs have been tested on Microsoft Windows XP
  - Most relatively recent similar versions of most of the software will likely work - For example, a slightly later version of Hibernate, Eclipse or Tomcat
  - There might be minor configuration changes

Notes:

Copyright © 2006-10 LearningPatterns Inc. All rights reserved.

Typographic Conventions

- Code that is inline in the text will appear in a fixed-width code font, such at this:
  `JavaTeacher teacher = new JavaTeacher()`
  - Any class names, such as *JavaTeacher*, method names, or other code fragments will also appear in the same font
  - If we want to emphasize a particular piece of code, we'll also bold it (and in the slide, change it's color) such as ***BeanFactory***
  - Filenames will be in italics, such as *JavaInstructor.java*
  - We sometimes denote more info in the notes with a **star ***
  - Lastly, longer code examples will appear in a separate code box as shown below

```
package com.javatunes.teach;
public class JavaInstructor implements Teacher {
   public void teach() {
      System.out.println("BeanFactories are way cool");
   }
}
```

Notes:

- * If we had additional information about a starred item in the slide, it would appear here in the notes

- We might also put other related information that generally pertains to the material in the slide

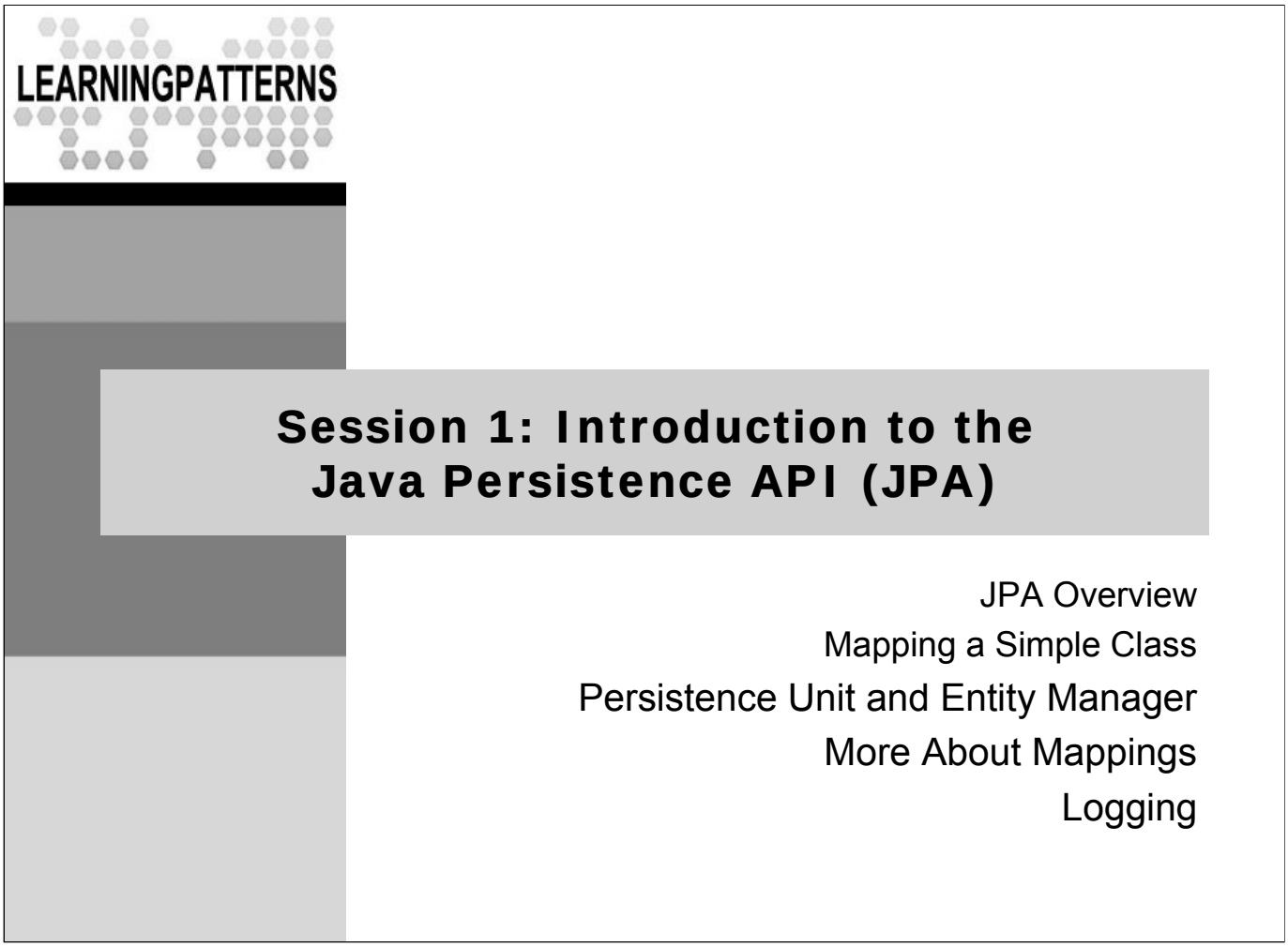
Copyright © 2006-10 LearningPatterns Inc. All rights reserved.

Labs

- The workshop consists of a minimum of 50% hands-on lab exercises
  - Many of the labs follow a common fictional case study - JavaTunes, an online music store
  - The labs are contained directly in the course book, and have detailed instructions on what needs to be done
- The course includes setup zip files that contain skeleton code for the labs
  - Students just need to add code for the particular capabilities that they are working with
  - There is also a solution zip file that contains completed lab code
- Lab slides contain an icon like the one in the upper right corner of this slide
  - The end of each lab is clearly marked with a stop like this one to the right

Notes:

Copyright © 2006-10 LearningPatterns Inc. All rights reserved.



Notes:

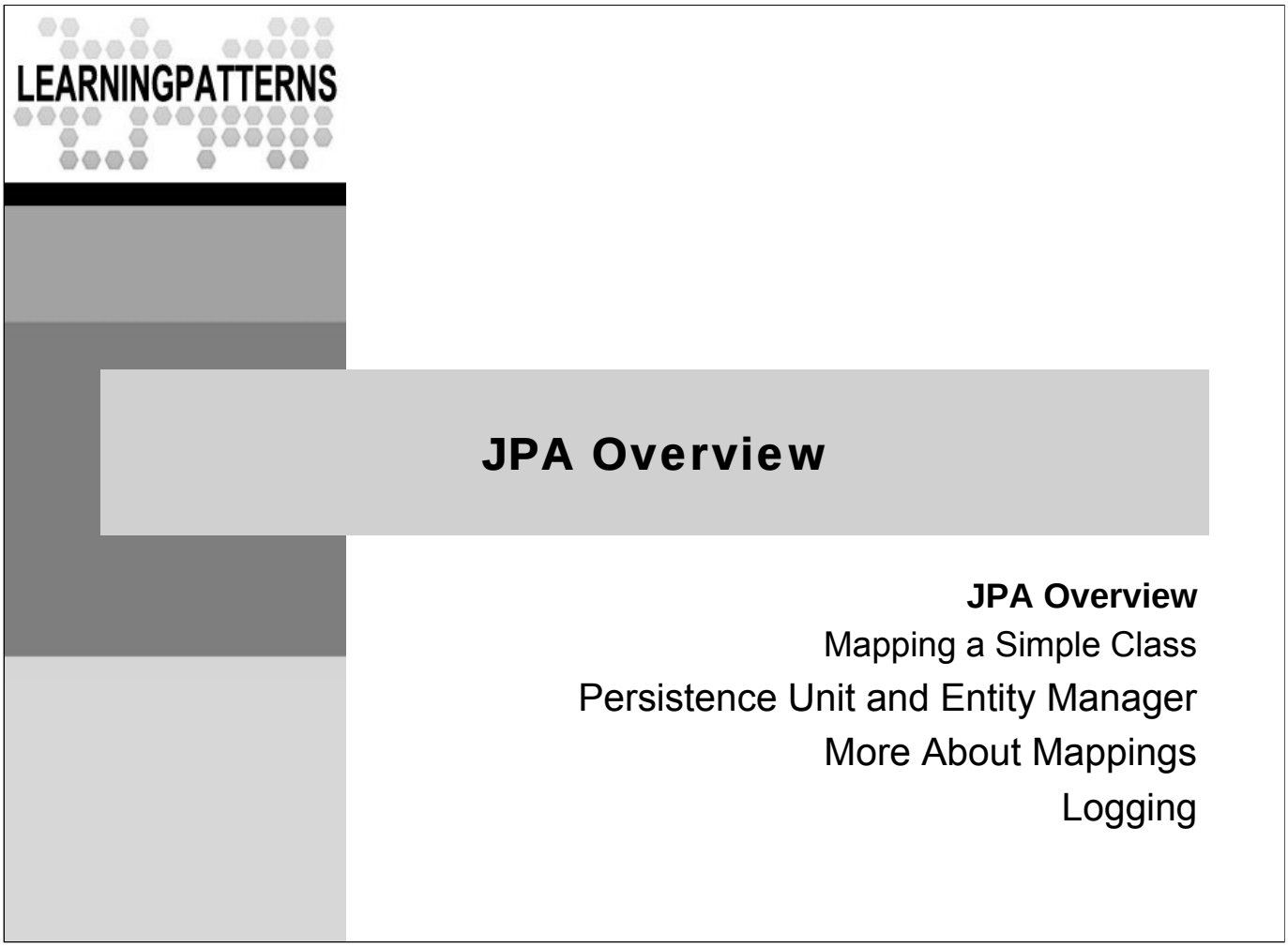
Copyright © 2006-10 LearningPatterns Inc. All rights reserved.

Lesson Objectives

- Describe the issues of object-relational mapping
- Describe the overall goals of the Java Persistence API (JPA)
- Describe the JPA architecture
- Create a simple JPA application
- Map a simple class to a DB using JPA

Notes:

Copyright © 2006-10 LearningPatterns Inc. All rights reserved.



Notes:

Copyright © 2006-10 LearningPatterns Inc. All rights reserved.

The Issues with Persistence Layers

- Data is a core element of many applications, and often stored in relational databases
- Much effort is expended writing persistence layers to store /retrieve data from the DB
  - These layers are complicated to write, and "homegrown" solutions are often buggy and incomplete
  - Changes to the database schema are often expensive to propagate to the persistence layer
- Even more complicated with an OO language, like Java
  - Java has JDBC for DB access, but many issues with that, e.g.
    - You don't store/load objects to the database – it's low level SQL
    - For example, to persist an object of some class, you write the SQL
    - To load an object from the database, you get result rows from the database, which must be converted into objects

Notes:

Copyright © 2006-10 LearningPatterns Inc. All rights reserved.

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 work
  - 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

Notes:

Copyright © 2006-10 LearningPatterns Inc. All rights reserved.

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
  - 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

Notes:

- JPA replaces the Entity bean technology in previous releases of EJB
  - Entity beans had significant problems in their architecture, and they never caught on
- There have been many ORM frameworks for Java available before JPA
  - For example, Toplink, Hibernate, Kodo, and JDO
- JPA provides a standard set of capabilities and a standard API for working with ORM

Copyright © 2006-10 LearningPatterns Inc. All rights reserved.

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 object types, collections, associations
  - Powerful query language
- Performance
  - Minimizes number of database updates
  - Lazy load on collections, disabling retrieval of associated objects
  - Object caching

Notes:

- JPA makes heavy use of defaults
  - The amount of metadata is minimized as much as possible
  - In some cases, almost no metadata will be needed

Copyright © 2006-10 LearningPatterns Inc. All rights reserved.

Java Persistence Environments

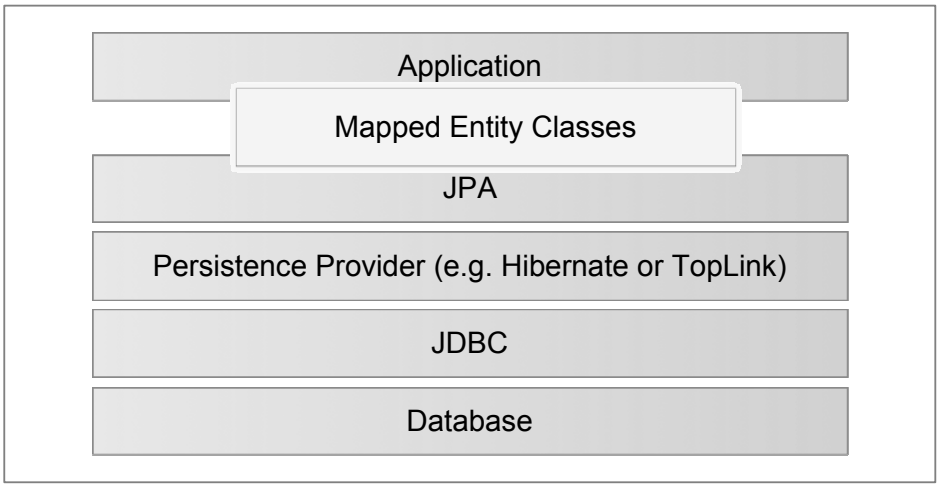
- Java Persistence is a standard part of **Java 5 EE**
- Java Persistence can also be used in **Java SE programs**
  - It is packaged as a set of jar files
  - There are some minor configuration and programming differences from using it in a Java EE environment
  - We'll show how to do it both ways
- Application server support is widespread
  - Almost all application servers are supporting JPA
  - Many support the full Java 5 EE range of technologies
  - Most JPA support is built on top of a few existing technologies
    - e.g. Oracle/Sun – Toplink based, Weblogic – Kodo based, JBoss – Hibernate based

Notes:

Copyright © 2006-10 LearningPatterns Inc. All rights reserved.

JPA Architecture – High Level View

- JPA uses database configuration data and entity metadata to provide persistence services and objects to the application
  - Generally, the JPA implementation is built over an existing persistence provider, which in turn is built over JDBC

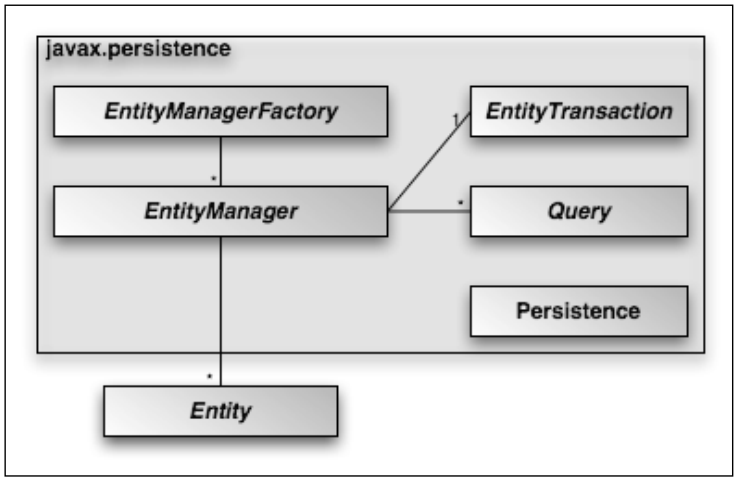


Notes:

Copyright © 2006-10 LearningPatterns Inc. All rights reserved.

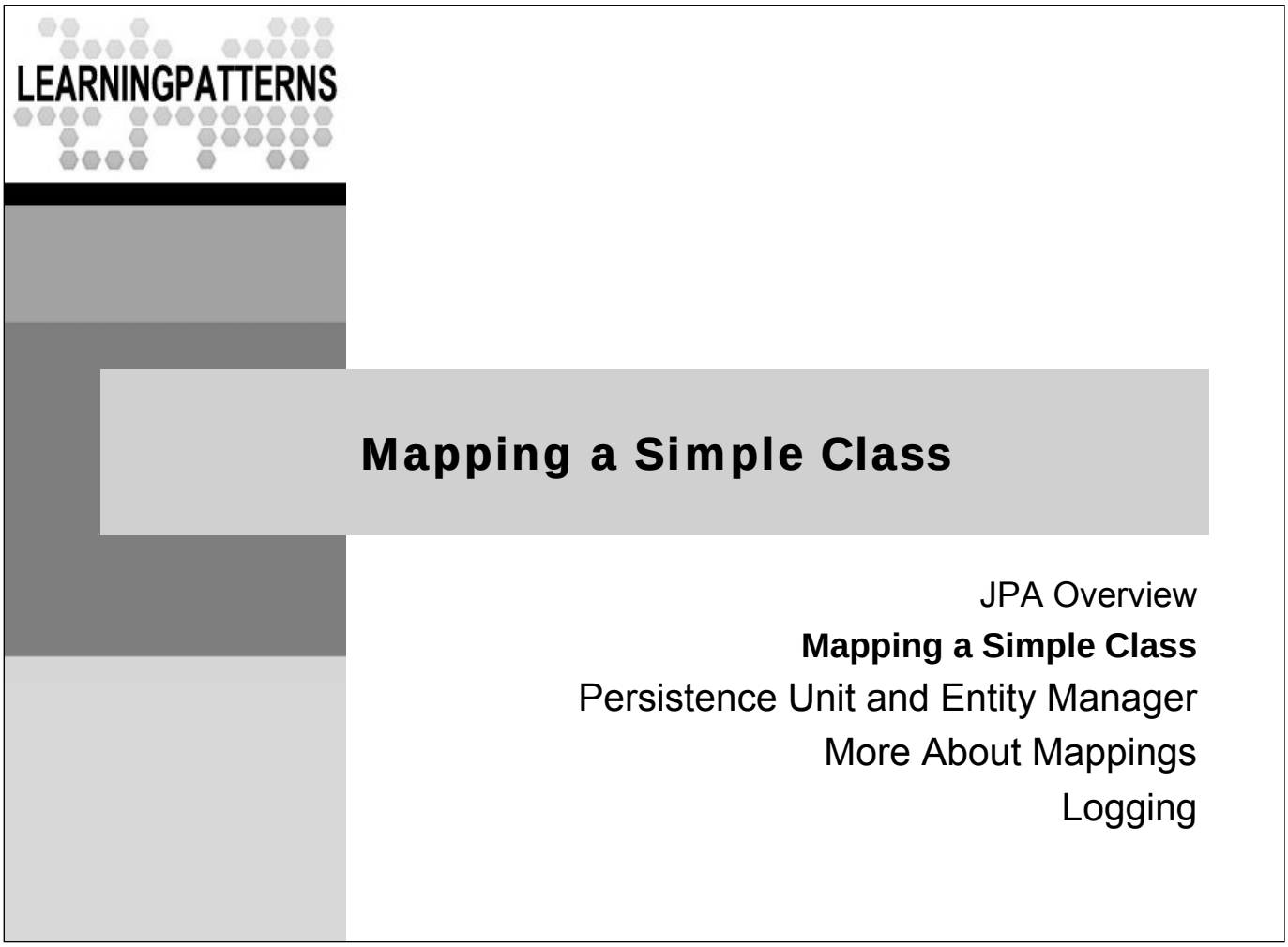
JPA Architecture – Programming View

- There are a number of key types in using JPA
  - ***Persistence***: For configuration of the system
  - ***EntityManager***: Manages a set of persistent entities
  - ***EntityManagerFactory***: Factory for EntityManagers
  - ***Query***: For finding entities
  - ***Entity***: POJO class mapped using JPA



Notes:

Copyright © 2006-10 LearningPatterns Inc. All rights reserved.



Notes:

Copyright © 2006-10 LearningPatterns Inc. All rights reserved.

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 (usually preferred) or XML
  - The metadata required is minimized through the use of intelligent defaults
  - The core annotations are in the package ***`javax.persistence`***

Notes:

- The concept of an entity has been present in database architecture for a long time
  - An entity is basically a set of data that is grouped together
  - It may participate in relationships to other entities
- In JPA any application defined object can be an entity

Copyright © 2006-10 LearningPatterns Inc. All rights reserved.

Entity Class Requirements

- Must be configured as an entity (via annotation or XML)
- Implement a no argument constructor for JPA to instantiate instances (may have other constructors)
- Contain instance variables to hold persistent state
  - Must **NOT** be declared public
  - Clients of the entity use accessor methods (get/set) for all persistent fields
- Provide an identifier property (usually 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*

Notes:

- From the JPA 2 specification:
  - The entity class must be annotated with the Entity annotation or denoted in the XML descriptor
  - The no-arg constructor must be public or protected.
  - The entity class must be a top-level class. An enum or interface must not be designated as an entity.
  - The entity class must not be final. No methods or persistent instance variables may be final.
  - If an entity instance is to be passed by value as a detached object (e.g., through a remote interface), the entity class must implement the Serializable interface.
  - Entities support inheritance, polymorphic associations, and polymorphic queries.
  - Both abstract and concrete classes can be entities. Entities may extend non-entity classes as well as entity classes, and non-entity classes may extend entity classes.
  - The persistent state of an entity is represented by instance variables, which may correspond to Java-Beans properties. An instance variable must be directly accessed only from within the methods of the entity by the entity instance itself. Instance variables must not be accessed by clients of the entity. The state of the entity is available to clients only through the entity's methods—i.e., accessor methods (getter/setter methods) or other business methods.

Copyright © 2006-10 LearningPatterns Inc. All rights reserved.

An Example Entity Class

```
package com.javatunes.schedule;

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

@Entity
public class Event implements java.io.Serializable {

    @Id
    private Long id;
    private String title;
    private Date date;

    Event() {}

    public Event(Long id) { setId(id); }

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

    // get/setDate and get/setTitle not shown but are as you expect
}
```

Notes:

- Regular classes can be transformed into entities by simply adding appropriate annotations
  - The class needs a no-arg constructor
- You can also use an XML configuration file to declare that a class is an entity
  - In general, annotations are much more widely used, and will be used in this class
- 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*
  - user-defined serializable types,
  - *byte[], Byte[], char[], and Character[]);*
  - enums;
  - entity types and/or collections of entity types;
  - embeddable classes

Copyright © 2006-10 LearningPatterns Inc. All rights reserved.

javax.persistence.Entity Annotation

- **@Entity** declares the class to be a persistent entity
- The name of the entity defaults to the unqualified name of the class (Event)
  - This name is used in queries, which we cover later
  - The name can also be set with the name element of *@Entity*
  *@Entity(**name="OurEventEntity"**)*
- 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:

```
import javax.persistence.*;

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

Notes:

- *@Entity* has only one element
  - name: The name of the entity for use in things like queries
- The default table name is the Entity name
  - Which in our case is the default value of the unqualified name of the class
  - It can be changed using *@Table(name="TableName")*
- *@Table* also has the following elements:
  - *catalog*: The catalog of the table
  - *schema*: The schema of the table
  - *uniqueConstraints*: Unique constraints to be placed on the table (used only if the table is generated from the Entity class)
  - These are all only useful if you are generating the DDL (table definitions) from the JPA entities
  - It is in the *javax.persistence* package

Copyright © 2006-10 LearningPatterns Inc. All rights reserved.

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**
  - That is, they are stored in the database
  - By default, they are stored in a column with the same name as the property
  - 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

Notes:

- JPA tries to use defaults that minimize the amount of metadata (annotation) information that is required in your entity classes
  - If a property is present in your class, it is persistent according to the standard, basic defaults
  - An optional ***@Basic*** annotation may be placed on the property to document this, but it's not necessary
- Note that the setter for id is private in our class definition
  - Programs will not be allowed to change this value
  - This is a common way of defining the accessor methods for the id

Copyright © 2006-10 LearningPatterns Inc. All rights reserved.

javax.persistence.Id and ID property

- *@Id* denotes that this property holds the primary key
  - A class must have a primary key
  - A non-composite primary key must correspond to a single field in an entity (e.g. the id field in the Event class)
- 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

Notes:

- A composite primary key must correspond to either a single persistent field or property or to a set of such fields or properties
  - A primary key class must be defined to represent a composite primary key
  - Composite primary keys typically arise when mapping from legacy databases when the database key is comprised of several columns
  - *@EmbeddedId* and *@IdClass* are used to denote composite primary keys

Copyright © 2006-10 LearningPatterns Inc. All rights reserved.

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/setter) methods (see below)
  - In that case, the runtime will access the properties via the accessor methods
  - This may be useful 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; }
```

Notes:

- *Caution should be exercised in adding business logic to the accessor methods when property-based access is used. The order in which the persistence provider runtime calls these methods when loading or storing persistent state is not defined. Logic contained in such methods therefore cannot rely upon a specific invocation order.* [JPA 2 Specification, Final, sec. 2.2]

Copyright © 2006-10 LearningPatterns Inc. All rights reserved.

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)
);
```

Notes:

- JPA has support for other types of generated values
  - For example, sequences and table generated keys

Copyright © 2006-10 LearningPatterns Inc. All rights reserved.

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
- Both annotations are in *javax.persistence*

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

Notes:

- *@GeneratedValue* also allows you to define a generator for the strategy

  ```
  @GeneratedValue(strategy=SEQUENCE, generator="EVENT_SEQ")
  ```

- *@Column* has a large number of optional elements
  - For example: nullable, unique, …

    ```
    @Column(name="SOME_COLUMN", nullable="false", unique=true)
    ```
  - Some of these (e.g. nullable) are only used if you are generating the DDL from the entity declarations
  - We'll cover some of them in the course
  - See the documentation for complete coverage
- All these types are in the *javax.persistence* package
- The AUTO strategy only makes sense if the runtime is generating the table definitions
  - Otherwise, of course, you'll need to use what's actually in the DB definition

Copyright © 2006-10 LearningPatterns Inc. All rights reserved.

Mapping Properties

- By default, all properties are considered persistent and mapped to columns with the same name as the property
  - However, all the defaults may not be appropriate for us
  - In our example, we 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 to change the defaults

```
// package / imports not shown in most examples ...

@Entity
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 ...
}
```

Notes:

Copyright © 2006-10 LearningPatterns Inc. All rights reserved.

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,
    - Entity types and/or collections of entity types;
    - Embeddable classes

Notes:

- The simple types above are mapped as part of the immediate state of the entity
  - They include almost all information that you want to persist
- Sometimes the type in the database is not exactly the Java type
  - In almost all cases, the provider runtime can convert between the two types
  - If the provider can't do the conversion, then generally an exception will be thrown

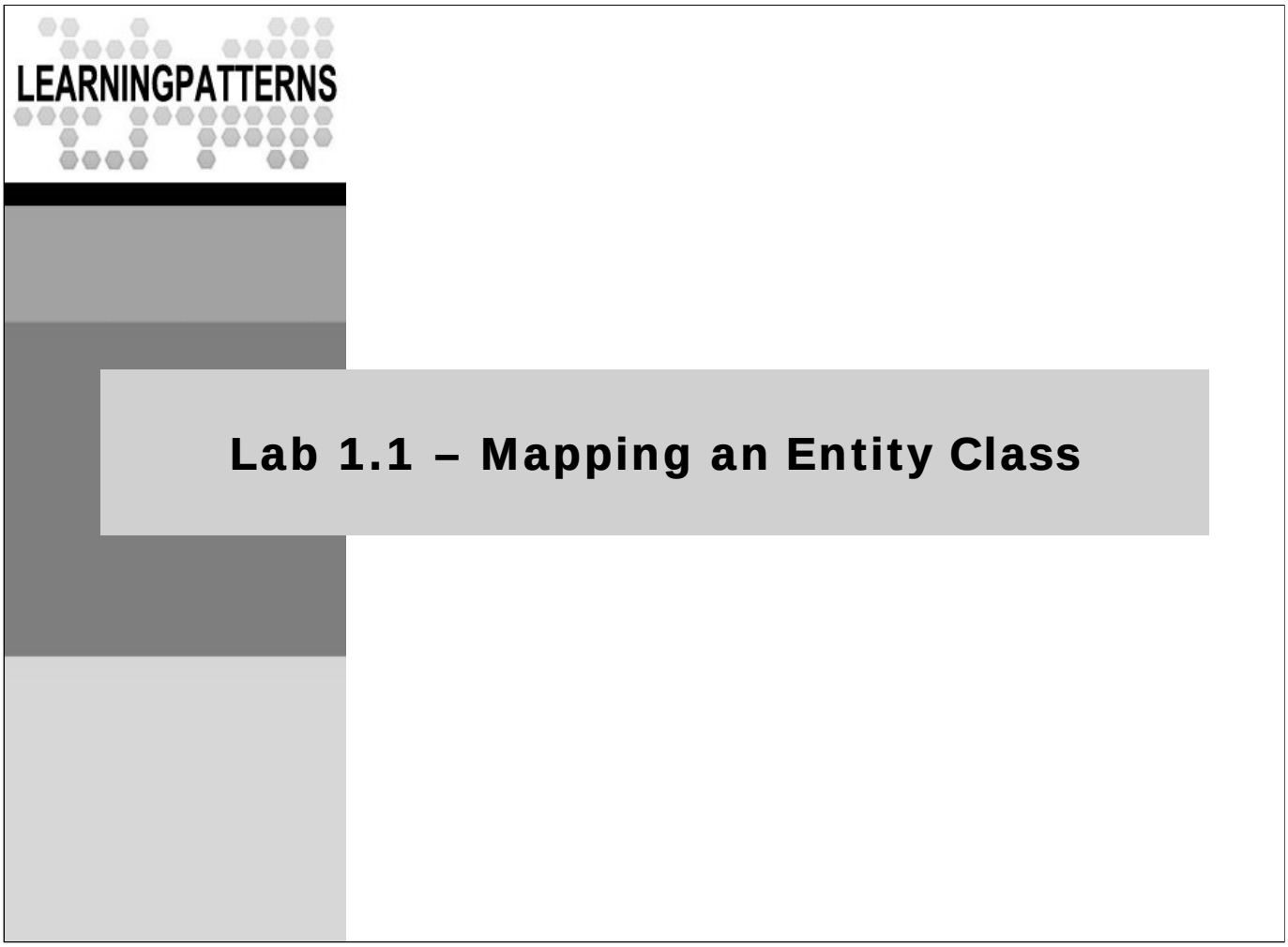
Copyright © 2006-10 LearningPatterns Inc. All rights reserved.

Persisting to the Database

- An **entity manager** (abbreviated EM) 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
  - We'll look at this shortly
- The entity manager API is completely separate from the mapping definition of an entity class
  - These responsibilities are not included in the mapping definition
  - This allows for a much cleaner definition of an entity class

Notes:

Copyright © 2006-10 LearningPatterns Inc. All rights reserved.



Notes:

Copyright © 2006-10 LearningPatterns Inc. All rights reserved.

Lab 1.1 – Mapping an Entity Class

- **Overview**: In this lab we will set up our development environment, and map a class - *MusicItem* - to the database:
  - We will add JPA annotations to the *MusicItem* class
  - These will include annotations for the entity class itself, the primary key and for the properties
  - We will NOT run a program until the next lab
- **Objectives**: Become familiar with the JPA API for mapping
  - Map an entity class using Java Persistence
  - Set up the Eclipse development environment
- **Builds on previous labs**: None
- **Approximate Time**: 30-40 minutes

Notes:

Copyright © 2006-10 LearningPatterns Inc. All rights reserved.

Information Content and Task Content

- Within a lab, information only content is presented in the normal way – the same as in the student manual pages
  - Like these bullets at the top of the page
- Tasks that the student needs to perform are in a box with a slightly different look – to help you identify them
- An example appears below

Tasks to Perform

- Look at these instructions, and notice the different look of the box as compared to that above
  - Make a note of how it looks, as future labs will use this format
- OK – Now **get out your setup CD**; we're ready to start working

Notes:

Copyright © 2006-10 LearningPatterns Inc. All rights reserved.

Setup Environment

Lab

Tasks to Perform

- Make sure **Java 5** (or later) is installed
  - Usually in: *C:\Program Files\Java\jdk1.5.x*
  - If not, download at: **http://java.sun.com/javase/downloads**
  - Java 5 is needed for the annotations in JPA
- Make sure **Eclipse** is installed (usually in *C:\eclipse*)
  - If not, download at **http://www.eclipse.org/downloads**
  - You want the Java EE version
- Make sure Hibernate 3.5 is installed
  - Usually in ***C:\hibernate-distribution-3.5.0-XXX***
  - If not, download at **http://www.hibernate.org/downloads.html**
  - We use Hibernate 3.5 as our JPA 2 provider
- Modify the lab instructions to match your environment

Notes:

- Hibernate 3.5 provides a complete JPA 2 implementation
  - We'll use that as our JPA provider
  - There are other open source implementations available – such as EclipseLink which is a TopLink based implementation
- The exact name of the Hibernate directory will depend on the release version that you use – e.g. C:\hibernate-distribution-3.5.0-Final
  - This was the current release as of the writing of this course (3.5.0 Final)
- If your environment is different from the one that the lab instructions are based on, then modify the instructions to conform to your environment
  - For example, you may have software installed in a different location, or may be using a slightly different version
  - That's OK, generally this will only result in minor differences from what the lab instructions say
  - Just conform to what is needed for your environment

Copyright © 2006-10 LearningPatterns Inc. All rights reserved.

Extract the Lab Setup Zip File

- To do the labs, you'll need the setup zip file for the course
  - This will either be on a CD for the course, or given by the instructor
  - The file name should be something like:

    JPA2_Hibernate-Tomcat_LabSetup_20100405.zip
- Our base working directory for this course (created when the setup zip is extracted) will be : **C:\StudentWork\JPA**
  - It includes a directory structure and files (e.g., Java files, XML files, other files) that will be needed in the labs
  - All instructions assume that this zip file is extracted to C:\. **If you choose a different directory, please adjust accordingly**

Tasks to Perform

- Unzip the lab setup file to ***C:***
  - This will create the directory structure, described in the next slide, containing files that you will need for doing the labs

Notes:

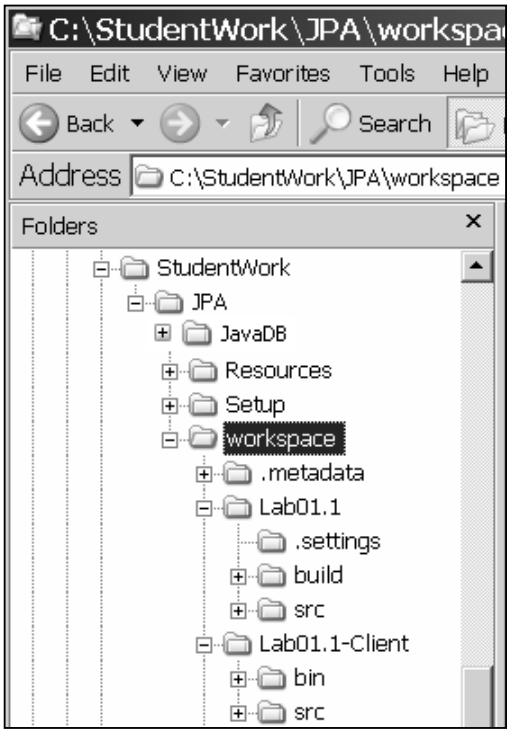
- The setup files may also contain additional folders, depending on how they are distributed
  - **Resources**: Documentation, specifications, etc.

Copyright © 2006-10 LearningPatterns Inc. All rights reserved.

Lab Directory Structure

Lab

- **StudentWork\JPA** contains
  - **JavaDB**: Database files
  - **Resources** : Extra files (e.g. docs)
  - **Setup**: Files needed for lab work
  - **workspace**: Lab working directories
- **StudentWork\JPA\workspace** contains the following folders:
  - **common**: shared files
  - **LabNN** : Lab directories
    - **LabNN/bin/** : compiled code *
    - **LabNN/src/** : Java source
    - **LabNN/src/META-INF/**: persistence.xml …



Notes:

- Eclipse will put the compiled Java code into the bin directory for Java projects
  - These will be generated for you, so you don't need to worry about them

Copyright © 2006-10 LearningPatterns Inc. All rights reserved.

General Instructions

- The root lab directory where you will do your work for this lab is: **C:\StudentWork\JPA\workspace\Lab01.1**
  - This directory already exists in your workspace – you'll need to edit files under this directory, and build the application from within this directory
  - In general, all the files you will work on for a lab will be under the root directory (and instructions are given relative to this directory)
- Detailed instructions are included in this lab
  - They include complete instructions for working in the **Eclipse** environment, as well as details about the lab requirements
- Subsequent labs require you to do the same thing as this lab to build/run your application, and they include fewer detailed instructions on how to do so

Notes:

Copyright © 2006-10 LearningPatterns Inc. All rights reserved.

The Eclipse Development Environment

- **Eclipse** is an open source platform for building integrated development environments (IDEs)
  - Used mainly for Java development
  - Can be extended via plugins to create applications useful in many areas (e.g. C# programming)
  - **http://www.eclipse.org** is the main website
- The remainder of this lab gives detailed instructions on using Eclipse to run the labs
  - Starting it, creating and configuring projects, etc.
- The other labs in the course include fewer specific details regarding Eclipse – they may just say build/run as previously
  - For these labs, you should use the same procedures to build/run as in this lab
  - Refer back to these lab instructions as needed

Notes:

- The Eclipse source base was originally developed by IBM
  - It was released by IBM into open source
  - IBM's RAD (and previously its WSAD) environment is built on top of Eclipse

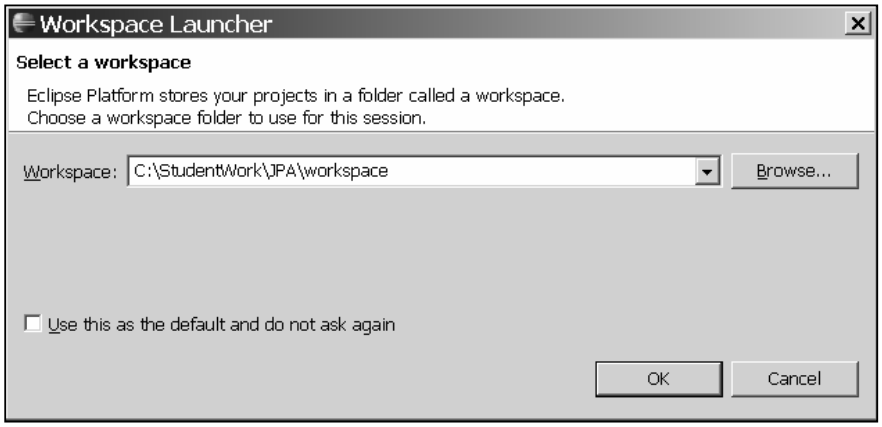
Copyright © 2006-10 LearningPatterns Inc. All rights reserved.

Launch Eclipse

Lab

Tasks to Perform

- To launch eclipse, go to **c:\eclipse** and run **eclipse.exe**
  - A dialog box should appear prompting for workbench location
  - Set the workbench location to **C:\StudentWork\JPA\workspace**
  - If a different default Workbench location is set, change it
  - Click **OK**
  - In the window that opens, close the **Welcome** screen (see notes)



Notes:

- If Eclipse was installed elsewhere, adjust the paths to the Eclipse executable accordingly
  - You can put a shortcut to this executable on your desktop
- Eclipse opens with a Welcome screen which allows you to navigate to different things (such as tutorials)
  - We will not be using this, so we will close it and go directly to the Workbench
  - To close it, just click the X on the Welcome tab

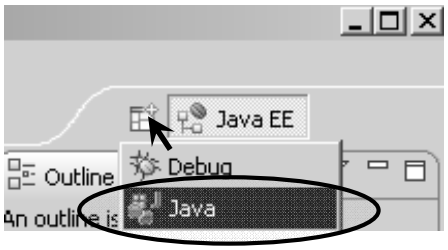
Copyright © 2006-10 LearningPatterns Inc. All rights reserved.

Workbench and Java Perspective

Lab

Tasks to Perform

- You should be in a Java EE perspective *
- If in a **Java EE** perspective, **open a Java one** by clicking the Perspective icon at the top right of the Workbench, and select Java (as shown below left)
  - Close the Java EE perspective by right clicking its icon, and selecting close (as shown below right)
  - If you were in a Java perspective, then just remain in it



Notes:

- The Eclipse Java EE version opens in the Java EE perspective by default

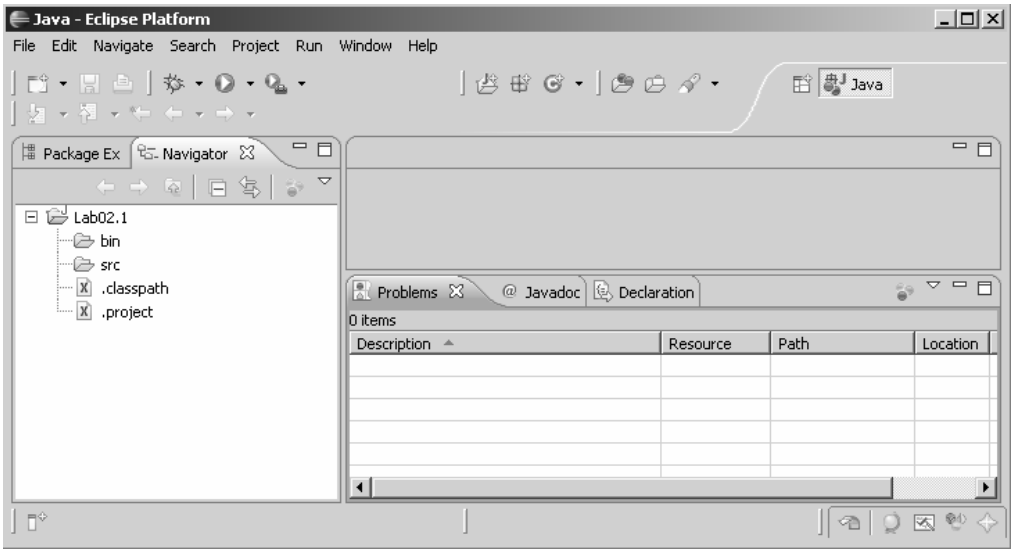
Copyright © 2006-10 LearningPatterns Inc. All rights reserved.

Unclutter the Workbench

Lab

Tasks to Perform

- Let's unclutter the Java Perspective by closing some views
  - Close the Task List, Outline and Hierarchy views (click the X for each)
  - Open the Navigator View (**Window | Show View | Navigator**)
  - You can save this as the default if you want (see note)



Notes:

- To save the perspective as the default Java perspective go to
  - Window → Save Perspective As → Java

- You can reset the perspective to it's defaults via
  - Window → Reset Perspective

Copyright © 2006-10 LearningPatterns Inc. All rights reserved.

Create User Libraries

- We'll need to include a number of Hibernate, JPA, and support jars in our classpath
  - We'll set these up as a user library for ease of use

Tasks to Perform

- Go to **Window | Preferences | Java | Build Path | User Libraries**
  - Click **New…**
  - In the next dialog, call the library **JPA2**, and press **OK**
  - Click the **Add JARs…** button, browse to <*hibernate*> *, select *hibernate3.jar* and select OK*
- You will need to add in all the jars listed on the following page
  - Each time you want to add jars, you'll need to select the JPA2 library in the **User Libraries** dialog, click the **Add JARs…** button, browse to the appropriate directory, select the jars needed and click OK

Notes:

- <hibernate> stands for the directory that Hibernateis installed in
  - For example, *C:\hibernate-distribution-3.5.0-Final*
- *hibernate3.jar* contains the complete Hibernate distribution, and we add that in as this is the easiest way to include Hibernate on your classpath
  - Starting in Hibernate 3.5, it includes the Hibernate Annotations and Hibernate EntityManager projects, which provide the JPA support
- You can select multiple files in the dialog by using holding down the `Control` key as you select each file

Copyright © 2006-10 LearningPatterns Inc. All rights reserved.

Create User Libraries

Lab

Tasks to Perform

- Add in the following jars to the **JPA** library *
  - From <*hibernate*>*lib**required*, add in all the jars in that directory
    - These are jars needed by Hibernate
  - From <*hibernate*>*lib**jpa*, add in all the jars in that directory
    - These are jars needed for JPA (e.g. the JPA 2 API)
  - From C:*StudentWork**JPA**JavaDB**lib* add in *derbyclient.jar* *
  - From C:*StudentWork**JPA**workspace**common**lib* add in all the jars in that directory
    - These are additional jars needed for the labs, such as logging jars
- **If using Java 5**, add the following jars
  - From C:*StudentWork**JPA**workspace**common****lib5*** add in all the jars in that directory (a JAXB implementation needed by Hibernate's JPA)
    - Not needed for Java 6, which includes JAXB in the JDK
- That's all the jars that you'll need, so click OK and finish

Notes:

- *derbyclient.jar* is needed to run the database access labs that use the JavaDB database – which is just a Derby database

- The workspace\common\lib directory contains any other dependencies that are needed in the labs
  - There may not be anything in the directory, in which case you don't need to do anything about it

Copyright © 2006-10 LearningPatterns Inc. All rights reserved.

The JavaTunes Online Music Store

- The labs are based on functionality from the JavaTunes music store
  - This online store allows you to browse for music, put music in a shopping cart, and purchase music
- We'll first use the ***MusicItem*** type - which is a simple value class representing an item of music in the JavaTunes store
  - It's in the package ***com.javatunes.persist***
  - It has the following properties, including get/set methods for each
    - *Long* ***id***, *String* ***num***, *String* ***title***, *String* ***artist***, *java.util.Date* ***releaseDate***, *BigDecimal* ***listPrice***, *BigDecimal* ***price***
- We provide the *MusicItem* class, but you'll need to annotate it to make it an entity class
  - In this lab, we'll annotate it, but won't use it until the next lab

Notes:

- MusicItem is a simple JavaBean type class with properties represented by get/set methods
  - For instance, the get/set methods for id are

```
public Long getId()
public void setId(Long id)
```

  - You can look at the class in the lab files if you want to see more about it

Copyright © 2006-10 LearningPatterns Inc. All rights reserved.

The Database Table for MusicItem

- The database table for *MusicItem* is defined as shown below
  - We'll create the database in the next lab
  - Note that the **table name is *Item***
  - Note the primary key (in *ITEM_ID*) **is an Identity column**
  - Note the **names and types of the other properties**

```
CREATE TABLE Item
(
   ITEM_ID     BIGINT NOT NULL GENERATED ALWAYS AS IDENTITY (START
                                          WITH 1, INCREMENT BY 1),
   NUM         VARCHAR(10) NOT NULL,
   Title       VARCHAR(40),
   Artist      VARCHAR(40),
   ReleaseDate DATE,
   ListPrice   DECIMAL(5,2),
   Price       DECIMAL(5,2),
   Version     INTEGER,
   CONSTRAINT  PK_Item  PRIMARY KEY(ITEM_ID)
);
```

Notes:

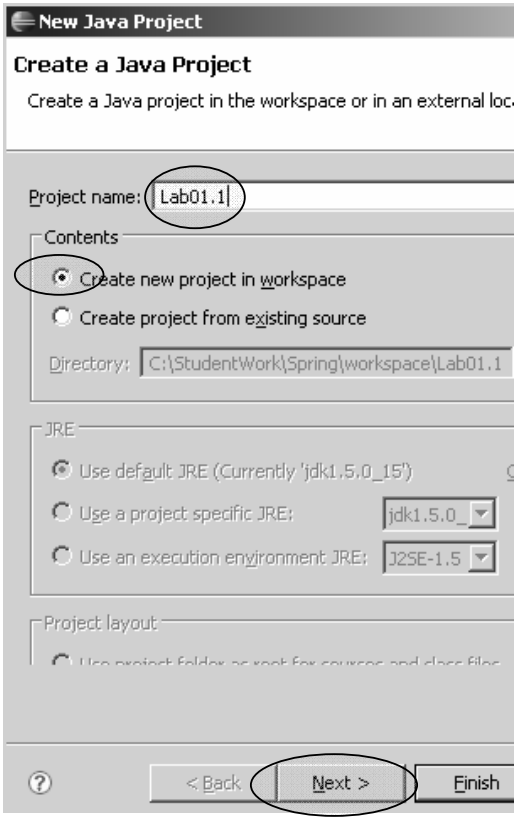
Copyright © 2006-10 LearningPatterns Inc. All rights reserved.

Create a Project for our Application

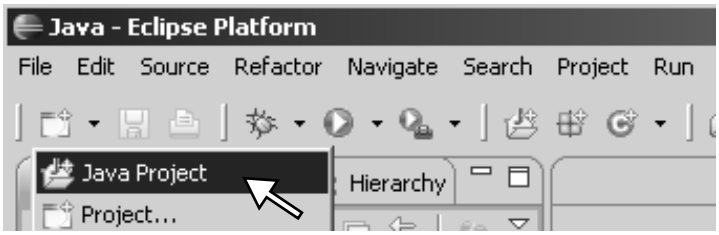
Lab

Tasks to Perform

- Close any open welcome screens
- Create a **Java Project** *
  - Call the project **Lab01.1**
  - Make sure **Create new project in workspace** is checked
  - Eclipse will then automatically set the project directory to *Lab01.1*
  - This directory already exists, and contains skeleton code for the lab
  - Click **Next**
  - This will bring you to a dialog where you can set the Java Settings for the project



Notes:



- To create a new Java Project, use the menu item
  - **File| New | Project | Java | Java Project**
- There are many other ways to create a project
  - You can use the new icon on the left hand side of the toolbar, as shown above

- You will need to create a **new Java project** in Eclipse each time the lab instructions tell you to use a new Lab directory
  - This happens several times in the course, and you'll need to create a new project each time

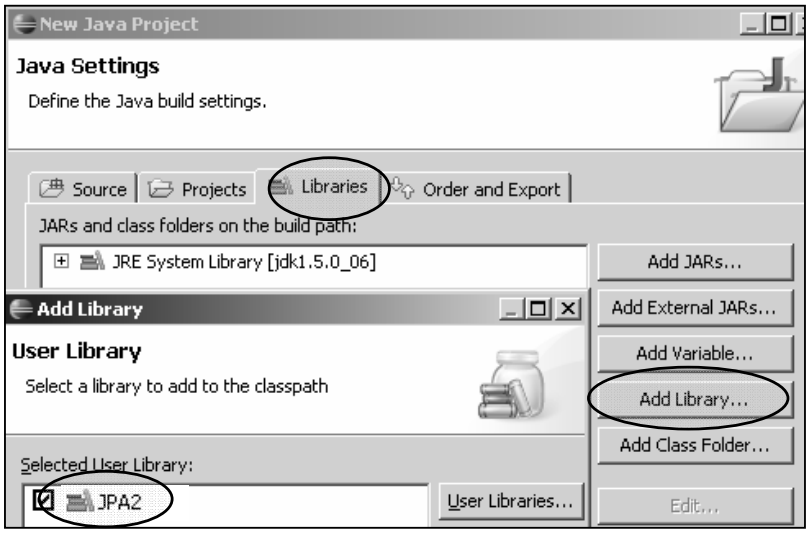
Copyright © 2006-10 LearningPatterns Inc. All rights reserved.

Add the JPA2 Library to the ClassPath

Lab

Tasks to Perform

- In the **Java Settings** dialog, click the **Libraries** tab *
  - Click the **Add Library** button, and in the dialog that comes up, select **User Library** then click **Next**
  - Check off **JPA2**, then click **Finish** in all open dialogs *



Notes:

- If you forget to add the library in this step, you can always add it later as follows
  - Right click on the project, select Properties, then select Java Build Path
  - Go to the Libraries tab, and add in the JPA2 library as described above

- When the project is created, Eclipse will attempt to compile all the Java source
  - This will lead to warnings about unused imports, variables, etc.
  - You can just ignore these – most of them are because the files we give you are just skeletons, and not complete.
  - If you want, you can shut these warnings off as follows:
    Window | Preferences | Java | Compiler | Errors/Warnings
  - Go to the "Unnecessary Code" section, and change the settings for "Unused import", "Unused local or private member", and "Local variable is never read" to "Ignore"

Copyright © 2006-10 LearningPatterns Inc. All rights reserved.

Finish MusicItem

Tasks to Perform

- Open ***MusicItem.java*** and do the following:
  - Add annotations to declare *MusicItem* an entity class
  - You'll need to import them – see the next slide for Eclipse help with importing
  - Add annotations to declare the primary key
  - Make sure you add in whatever imports are needed (see notes)
  - Look over the other properties and see what (if anything) is needed to persist them
  - Are the default values OK for them?
- Eclipse should **compile** the code for you when you save it
- Once the file compiles cleanly, **you're finished with this lab**
  - We'll use this class in the next lab

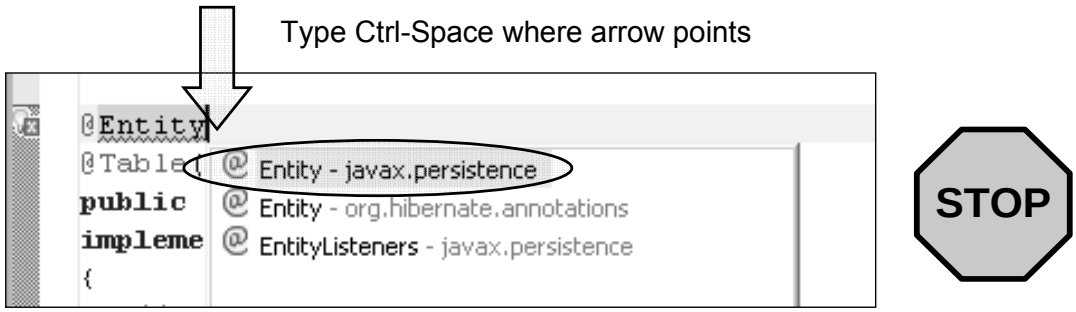
Notes:

- When looking at the properties, you'll want to make sure that the default column name is appropriate for the database table definition
  - If not, you'll need to use an @Column annotation to specify the column name
- Note that Eclipse makes it very easy to import classes into your code
  - Type the name of the class you're using at the point in the source you need it
  - At any time while you're typing it, you can type Ctrl-space (that means press the Ctrl key and space bar at the same time) - this invokes Eclipse's Content Assistance
  - This will give you a list of potential classes that are on the build path of the code you're writing - you can choose the correct class, and Eclipse will add the import statement for you - and complete the class name if you've only typed part of it

Copyright © 2006-10 LearningPatterns Inc. All rights reserved.

Eclipse Hint - Importing

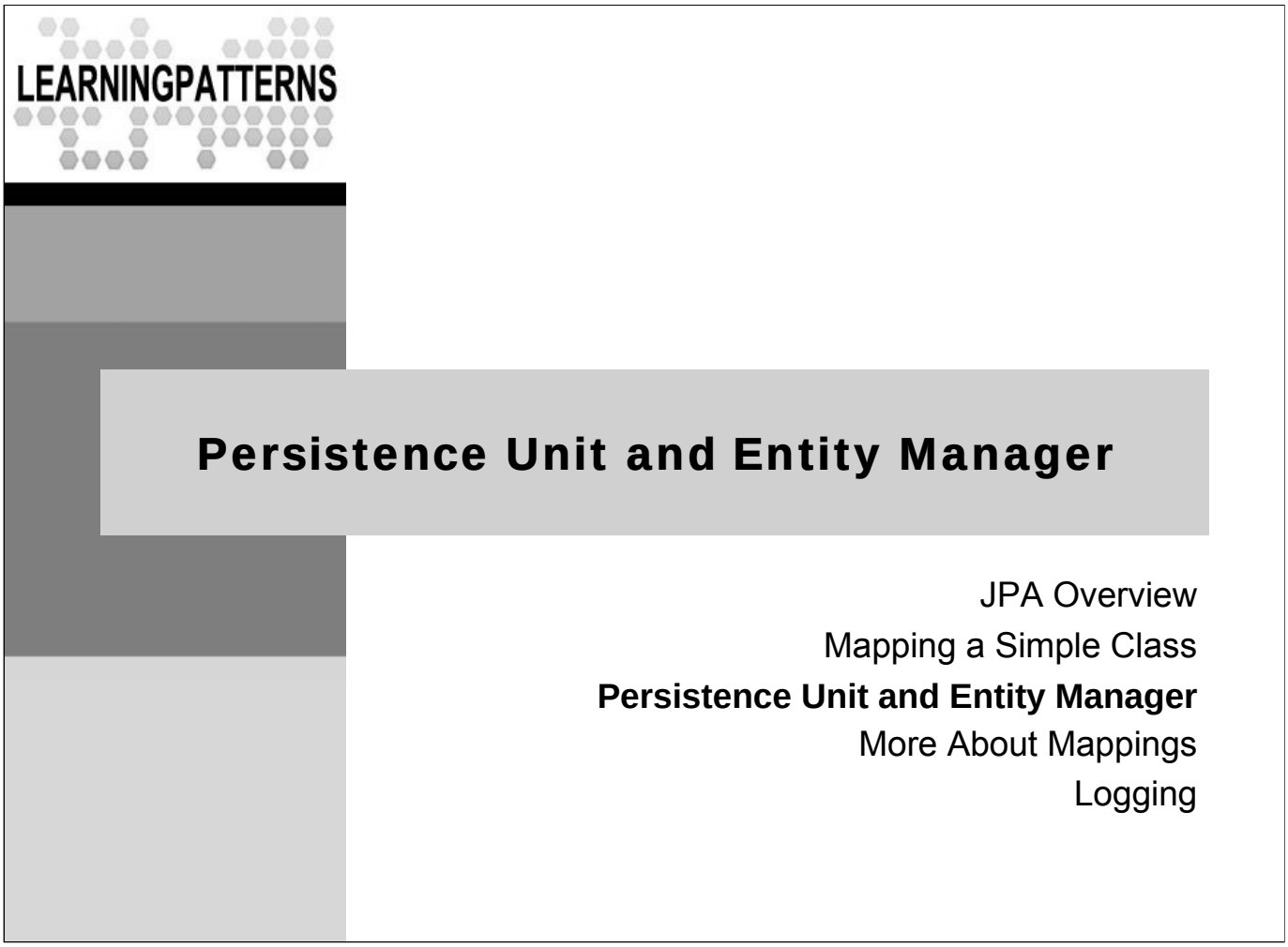
- One of the nicest features of Eclipse is its auto-import feature
  - To try it out, in your *MusicItem.java* file, remove the import for *`Entity`* if you've already put it in (if you haven't imported it, then just continue below)
  - At the location where you're going to use the annotation, type *`@Entity`*, then type **Ctrl-Space**
  - Select the *`javax.persistence`* entry and hit return
  - An ***`import javax.persistence.Entity`*** statement will very conveniently be added near the top of your source file



Notes:

- Eclipse can help you with importing classes that you need
  - When you use a class, and it's not properly imported, Eclipse notices, and marks if for you
  - If you move your cursor into or at the end of the class name, and type Control-space, it will bring up a window with possible classes to import
  - You can select the class you want, and Eclipse will automatically add in the import for you

Copyright © 2006-10 LearningPatterns Inc. All rights reserved.



Notes:

Copyright © 2006-10 LearningPatterns Inc. All rights reserved.

The Persistence Unit

- A **persistence unit** defines the set of entities that can be managed by an entity manager (covered next)
  - A persistence unit defines the types that can be managed, the database they are persisted to, and the underlying mechanism used to persist entities (the persistence provider)
- A persistence unit is named and configured in a configuration file (usually called ***persistence.xml***)
  - It must be included in the META-INF directory of one of the jar files in the application (called the root of the persistence unit)
  - It provides for standard JPA configuration
    - Also allows provider specific configuration properties which are passed through to the underlying provider

Notes:

- The persistence unit will be packaged up in a persistence archive
  - Basically just a jar that contains persistence.xml in the META-INF directory
  - The root jar file can be a regular jar, an ejb-jar, and others
- The persistence unit is basically a configuration artifact that lets you specify things like how the DB is accessed, global configuration information such as transaction manager type, etc.

Copyright © 2006-10 LearningPatterns Inc. All rights reserved.

persistence.xml

- *persistence.xml*, as in the example below, defines a persistence unit
  - **<persistence>** specifies the namespaces and versions (for JPA 2)
  - **<persistence-unit>** defines the persistence unit
  - The required **name** attribute specifies the persistent unit name
  - The optional **transaction-type** attribute specifies transaction type
    - Resource local (i.e. JDBC transactions) used here – suitable for Java SE *
  - **<properties>** allows you to pass configuration properties to the provider, as with the **hibernate.dialect** property below

```
<?xml version="1.0" encoding="UTF-8"?>

<persistence xmlns="…" version"2.0"> <!-- namespaces not shown -->
   <persistence-unit name="events"
                     transaction-type="RESOURCE_LOCAL">
      <properties>
        <property name="hibernate.dialect"
                  value="org.hibernate.dialect.DerbyDialect"/>
      </properties>
   </persistence-unit>
</persistence>
```

Notes:

- The root *<persistence>* element specifies the namespaces and version according to the JPA spec
  - The labs and the example above are compatible with JPA 2
  - The labs include the complete namespace declarations which are not shown to save space
- Different environments will use different transaction types
  - For example, in a Java EE environment, we would use JTA transactions, and have transaction-type="JTA"
- Different underlying providers will require different configuration mechanism
  - The property capability shown allows you to pass whatever properties the provider needs
  - These are dependent on the provider, and will vary for different providers
- The *hibernate.dialect* property is a vendor specific property used to configure an underlying Hibernate provider
  - It specifies the type of database that is being used
  - This allows Hibernate to optimize the SQL for that database
- There are a number of other elements available in persistence.xml
  - See the documentation for more detail

Copyright © 2006-10 LearningPatterns Inc. All rights reserved.

Classes included in a persistence unit

- The classes included in a persistence unit are defined by:
  - Annotated managed persistence classes contained in the root of the persistence unit (unless exclude-unlisted-classes is specified)
  - One or more object/relational mapping XML files
  - One or more jar files that will be searched for classes
  - An explicit list of classes
- Generally, in a Java SE environment, you will use auto-scanning, where annotated persistent classes are automatically detected
  - The code sample below shows how to list them explicitly, if desired

```
<!-- Disable scanning -->
<exclude-unlisted-classes>true</exclude-unlisted-classes>
<!-- Include the Event class -->
<class>com.javatunes.Event</class>
```

Notes:

- From the JPA spec [JPA 2 Specification, Final, Sec. 8.2.1.6]
  - All classes contained in the root of the persistence unit are searched for annotated managed persistence classes—classes with the Entity, Embeddable, or MappedSuperclass annotation—and any mapping metadata annotations found on these classes will be processed, or they will be mapped using the mapping annotation defaults. If it is not intended that the annotated persistence classes contained in the root of the persistence unit be included in the persistence unit, the exclude-unlisted-classes element must be specified as true. The exclude-unlisted-classes element is not intended for use in Java SE environments.
  - A object/relational mapping XML file named orm.xml [with mapping information for the classes listed in it] may be specified in the META-INF directory in the root of the persistence unit or in the META-INF directory of any jar file referenced by the persistence.xml. Alternatively, or in addition, one or more mapping files may be referenced by the mapping-file elements of the persistence-unit element. These mapping files may be present anywhere on the class path.
  - An orm.xml mapping file or other mapping file is loaded as a resource by the persistence provider. If a mapping file is specified, the classes and mapping information specified in the mapping file will be used
  - One or more JAR files may be specified using the jar-file elements instead of, or in addition to the mapping files specified in the mapping-file elements. If specified, these JAR files will be searched for managed persistence classes, and any mapping metadata annotations found on them will be processed, or they will be mapped using the mapping annotation defaults defined by this specification. Such JAR files are specified relative to the directory or jar file that contains[82] the root of the persistence unit.

Copyright © 2006-10 LearningPatterns Inc. All rights reserved.

The EntityManager & Persistence Context

- An **entity manager** (abbreviated EM) object is used to interact with a database, to create, read, or write an entity
  - An entity manager is configured to work with a specific data store
  - It also contains the API to interact with the data store
- The EM also **manages** entity instances
  - When the EM obtains an entity reference, the referenced object becomes a **managed** entity
- A **persistence context** is associated with an EM, and is comprised of the set of all managed entities within it
  - **Only one instance** with a given persistent identity can exist within a persistence context
  - e.g., if you get an event with id=25 twice from a persistent context, the 2nd retrieval returns the same instance as the 1st

Notes:

- Entities don't interact with the database on their own, or contain the API to do so
  - Applications must use an entity manager to interact with the DB
- Note that the idea of a persistence context is an important one
  - A persistence context is a set of entity instances in which for any persistent entity identity there is a unique entity instance
  - Within a persistence context, database identity and Java identity are the same
  - Within a persistence context, the entity instances and their lifecycle are managed

Copyright © 2006-10 LearningPatterns Inc. All rights reserved.

EntityManager Interface

- ***EntityManager*** contains the entity manger API
  - Defines methods to create and remove persistent entities, to find entities by their primary key, and to query over entities
- Some important *EntityManger* methods include:
  - ***<T> T find(Class<T> entityClass, Object primaryKey)***: Find an entity by its primary key
  - ***void persist(Object entity)***: Make a new entity instance managed and persistent
  - ***void refresh(Object entity)***: Refresh the state of the instance from the database, overwriting changes made to the entity, if any
  - ***void remove(Object entity)***: Remove the entity instance
- Many more methods – We'll cover many of them in the course

Notes:

- Note that an entity manager is not thread safe
  - It is meant to be used by a single thread at a time
- Entities are also intended to be accessed by a single thread at a time while they are being managed
  - Really they also shouldn't be accessed by multiple threads when they are detached (which we talk about later)
  - However, if you want to manage the thread safety in the entity instance for when it's detached, it's possible to do so
  - It's not desirable
  - It's better to make copies and merge the state back into a persistence context
- Notice that the find method is declared using Java generics
  - This allows you to use it without casting the return type
  - It's much more convenient than writing it in terms of Object and then casting the return value

Copyright © 2006-10 LearningPatterns Inc. All rights reserved.

Obtaining an Entity Manager

- An ***EntityMangerFactory*** provides *EntityManger*s
  - An *EntityManagerFactory* is bound to a persistence unit
  - It is thread-safe, and immutable
- In Java SE, you usually **programmatically create** the entity manager with an entity manager factory
  - You also programmatically close it, and control the transactions (either directly, or with another container like Spring)
  - We'll start with using JPA in a Java SE environment
- In a Java EE or Spring environment, the entity manager is generally **injected** using dependency injection
  - Java EE also provides container managed transactions, and TX scoped entity managers making JPA very easy to use
  - We'll look at this later

Notes:

- You can obtain an *EntityManager* from an *EntityManagerFactory*
  - Its behavior is controlled by the configuration properties
  - Any *EntityManager* you get from a given factory will only manage instances as defined by that persistence unit
- When you create and close an entity manager, you are also creating and destroying the associated persistence context
  - This just means that when the entity manager is closed, all entities in the persistence context become unmanaged
  - We'll look at this more when we talk about detached entities
- When using EJB, you can also inject an entity manager by providing the name of the persistence unit
  - This is the preferred way to obtain an entity manager when working with EJB

Copyright © 2006-10 LearningPatterns Inc. All rights reserved.

Java SE APIs

- In a Java SE environment, you generally use a bootstrap class called ***Persistence*** to read the configuration (*persistence.xml*)
  - The *Persistence* type then produces the *EntityManagerFactory*, from which you can get *EntityManager*s
  - You pass in the name of the persistence unit when creating the factory
- You will likely use a **resource-local** entity manager
  - Where the application directly controls the transaction boundaries using the entity manager API
  - This is configured in the *persistence.xml* file:

```
<persistence-unit name="javatunes"
                  transaction-type="RESOURCE_LOCAL">
```

- Every new entity manager has its own persistence context

Notes:

Copyright © 2006-10 LearningPatterns Inc. All rights reserved.

Entity Manger and Transactions

- By default, work done with the EM 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 EM
  - When the transaction commits, the objects are written to the DB
- You can work with transactions through the JPA API *
  - You start a transaction by using the entity manager to obtain a **javax.persistence.EntityTransaction** object

```
EntityTransaction tx = em.getTransaction();
tx.begin();
```

  - You can commit (or rollback) via the transaction object

```
tx.commit();  // Commit the transaction
```

Notes:

- There are a number of different ways to work with transactions
  - We'll cover these in more detail later
  - Fro now, we'll show the basics of starting and committing transactions

Copyright © 2006-10 LearningPatterns Inc. All rights reserved.

Using JPA in Java SE

- Below is some common boilerplate code for Java SE
  - We'll talk about more of the details later

```
// Get an entity manager factory
EntityManagerFactory emf =
            Persistence.createEntityManagerFactory("events");
// Get an entity manager
EntityManager em = enf.createEntityManager();
try {
  em.getTransaction().begin();  // Begin a transaction
  // Application code using the entity manager
  em.getTransaction().commit();  // Commit a transaction
}
finally {
  if (em.getTransaction().isActive()) {  // TX did not commit
    em.getTransaction().rollback();      // Roll the TX back
  }
  em.close();  // Close the entity manager
  emf.close(); // Close the entity manager factory
}
```

Notes:

- It's important to close an application-managed entity manager when you're done with it
  - This releases all resources held by the entity manager, and the persistence context becomes unmanaged
  - In a Java SE environment, we need to do this explicitly
  - We use a finally block to make sure the transaction is finalized, and that all the resources are closed
  - After calling close, most method calls on the entity manager or on Query objects obtained from it will throw an *IllegalStateException*

- In a Java EE environment, most of this is not needed
  - You'll use a container managed entity manager which will be scoped to the transaction, injected, and automatically closed when the transaction is finalized

Copyright © 2006-10 LearningPatterns Inc. All rights reserved.

Retrieving Persistent Objects

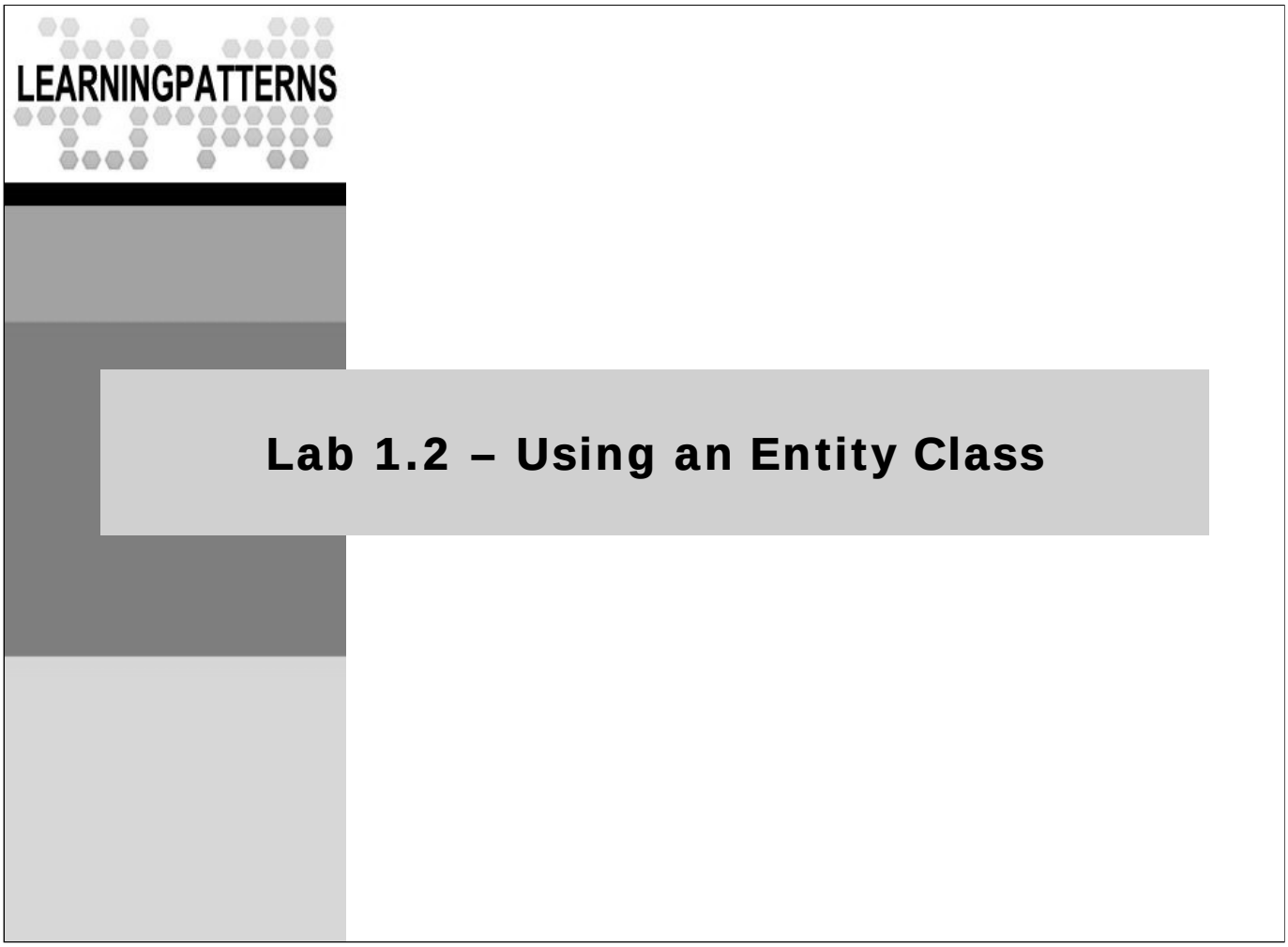
- You can use the *EntityManger.find()* method to retrieve persistent objects from the database via a primary key
  - **<T> T find(Class<T> entityClass, Object primaryKey)**
  - Returns the persistent instance of the given entity class with the given identifier, or null if there is no such persistent instance
  - If an instance with the given primary key is already managed (i.e. it was already retrieved into this persistence context) return that instance

```
// EntityManger initialization omitted
Long id = new Long (1);
Event retreivedEvent = em.find(Event.class,id);
```

Notes:

- The find method takes the class of the entity being sought, and the id or primary key that identifies the entity
  - It returns a managed entity that is initialized with the data from the database
- There is no cast required on the return value of the find method
  - It is defined using a parameterized return type, to return the same type that was passed in
- If there is no object found with the given id, then find returns a null
  - No exception is thrown
  - Application code needs to check for nulls if this is a concern
- There should only be one managed instance with a given primary key in a persistence context
  - The methods of *EntityManager* that retrieve instances enforce this rule

Copyright © 2006-10 LearningPatterns Inc. All rights reserved.



Notes:

Copyright © 2006-10 LearningPatterns Inc. All rights reserved.

Lab 1.2 – Using an Entity Class

- **Overview**: In this lab we will use the *EntityManager* to lookup *MusicItem* entities from data stored in a database
  - We provide a simple test program to do this
  - We will need to add JPA code into the main method
- **Objectives**: Use a JPA entity class to work with stored data
- **Builds on previous labs**: Lab 1.1
  - Continue working in your **Lab01.1** directory for this lab
- **Approximate Time**: 25-35 minutes

Notes:

- It's a stylistic choice as to where you put your JPA code
  - Generally, the entity classes will contain the annotations (unless you decide to do it with XML mapping files), but you still need to decide where entity manager code should go
- For now, we'll just put it in a test program – a simple program with a main method
  - In the future we'll show some other choices – Data Access Objects, EJB, etc

Copyright © 2006-10 LearningPatterns Inc. All rights reserved.

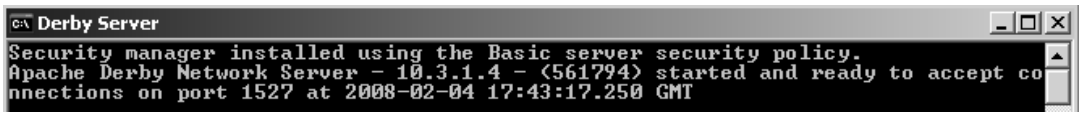
Setup/Create the Database

Lab

- We're using JavaDB – which is actually the open source Derby DB

Tasks to Perform

- Look at the scripts we provide to work with the Derby DB, in ***C:\StudentWork\JPA\Derby***
- 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



- Create the database
  - Execute ***dbCreate.cmd*** (you can double-click on it)
  - This creates the JavaTunesDB database
  - (you can ignore any DROP TABLE errors, if you see them)

Notes:

- JavaDB is included with Java 6
  - We include it in the setup for those not using Java 6
  - It is just the open source Derby database

- *dbCreate.cmd* generates DROP TABLE errors the first time it is run, because it drops the Item and other tables, which don't exist the first time, then creates them
  - The database for this course consists only of the Item table.
  - If you ever want a "fresh" database, simply rerun *dbCreate.cmd*.

- *dbSQL.cmd* launches the ij program , a command line SQL tool.
  - All commands in ij are terminated with a semicolon (`;`). For help, type `help;`. To exit, type `exit;`.

- To stop the database server, run the *dbStop.cmd* program
  - But leave it running for now

Copyright © 2006-10 LearningPatterns Inc. All rights reserved.

Using an SQL Command Tool

- **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

Tasks to Perform

- Execute ***dbSQL.cmd*** to run ij (you can double-click on it)
  - This command file is set up to connect to the correct DB
  - Using ij, you can browse the DB
  - Execute the query shown below to see the items in the DB

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

Notes:

- All ij commands are terminated with a semicolon (`;`).
- Many ij commands require single quotes (`'`).
- For a list of commands, type `help;`.
- ij can connect to any database server if you know have the driver and know URL. You just need to specify the driver and hostname, as shown below for Derby and the JavaTunes DB
  - `DRIVER 'com.ibm.db2.jcc.DB2Driver';`
  - `connect 'jdbc:derby:net://localhost:1527/JavaTunesDB' USER 'guest' PASSWORD 'password' ;`

Copyright © 2006-10 LearningPatterns Inc. All rights reserved.

persistence.xml

Lab

Tasks to Perform

- Open *persistence.xml* - located in the ***META-INF*** directory
  - Look for the TODOs
  - Set the name of the persistence unit to ***javatunes***
  - Set the transaction type to ***RESOURCE_LOCAL***
- Note the Hibernate specific properties in *persistence.xml*, e.g.

```
<property name="hibernate.dialect"
    value="org.hibernate.dialect.DerbyDialect"/>
```

  - This particular property tells Hibernate that we are using the Derby database

- There are many other Hibernate Specific properties
  - The next slides cover what is available, and how to configure the database connection using them

Notes:

- We also have another property that turns on basic logging of the SQL that Hibernate uses

```
<property name="hibernate.show_sql" value="true"/>
```

  - We'll use this in a later lab to see how to optimize querying using fetch joins
  - For now, it's useful to see what SQL Hibernate is generating

Copyright © 2006-10 LearningPatterns Inc. All rights reserved.

Hibernate Configuration Properties

- There are many properties available
  - We show some of the important ones here – see the docs if you're interested
- 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

Notes:

- Some of the other Hibernate properties are:
- JDBC Tuning
  - hibernate.jdbc.fetch_size, hibernate.jdbc.batch_size, hibernate.jdbc.use_scrollable_resultset, hibernate.connection.provider_class, hibernate.connection.isolation, hibernate.connection.autocommit, hibernate.connection.<*propertyName*>, hibernate.jndi.<*propertyName*>
- Hibernate Cache Properties
  - hibernate.cache.provider_class, hibernate.cache.use_query_cache, hibernate.cache.use_second_level_cache, and more
- Hibernate Transaction Properties
  - hibernate.transaction.factory_class, jta.UserTransaction, hibernate.transaction.manager_lookup_class, hibernate.transaction.flush_before_completion, hibernate.transaction.auto_close_session

Copyright © 2006-10 LearningPatterns Inc. All rights reserved.

Finish persistence.xml

Tasks to Perform

- Complete the other hibernate properties for a DB connection as shown below (look for the TODOs in the file)
  - You need to fill in the username, password, and the name of the database in the connection url (JavaTunesDB)

```
<!-- Database Connection Settings -->
<property name="hibernate.connection.username">guest</property>
<property name="hibernate.connection.password">password</property>
<property name="hibernate.connection.url">
   jdbc:derby://localhost:1527/JavaTunesDB</property>
<property name="hibernate.connection.driver_class">
   org.apache.derby.jdbc.ClientDriver</property>

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

<!-- Other Hibernate specific configuration not shown -->
```

Notes:

- The first four property elements contain the necessary configuration for the JDBC connection
  - The dialect property element specifies the particular SQL variant Hibernate generates.
  - If you're working in a J2EE environment, you can specify a datasource with standard JPA elements, and don't need to use the Hibernate elements

- Since Hibernate is the persistence provider, it is the layer in your application which connects to this database, so it needs connection information
  - The connections are made through a JDBC connection pool, which we also have to configure
  - The Hibernate distribution contains several open source JDBC connection pooling tools, but this example uses the Hibernate built-in connection pool.
  - Note that you have to copy the required library into your classpath and use different connection pooling settings if you want to use a production-quality third party JDBC pooling software
  - [Hibernate 3.2.2 Reference Documentation, Section 1.2.3]

Copyright © 2006-10 LearningPatterns Inc. All rights reserved.

Finish the Test Class

Tasks to Perform

- Open *JPATest.java* (located in the package *com.javatunes.persist.client*)
  - We'll use this to write some fairly straightforward code to work with JPA, and we'll add complexity later
- Do the following in the main method
  - Create an *EntityManagerFactory* and *EntityManager*
    - You'll need to use the name of the persistence unit from *persistence.xml* (javatunes)
  - Use the *EntityManager.find()* method to get an instance of *MusicItem* using the id of 1, and print it out (you can just use *System.out.println()* with an item to output it)
  - See the earlier manual slides for examples
  - Don't forget to add in any imports

Notes:

- In the last lab, you put annotations on *MusicItem* to make it an entity class
  - In this lab, you'll write code to use the *EntityManager* and the *MusicItem* class go do to the database

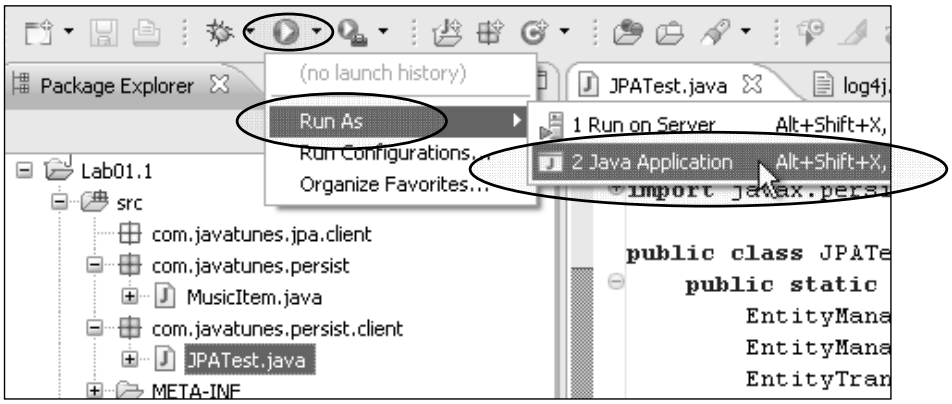
Copyright © 2006-10 LearningPatterns Inc. All rights reserved.

Running Your Application

Lab

Tasks to Perform

- After a clean build (one that is error-free, but not necessarily warning-free), test the application as follows
  - Select *JPATest.java* in the Navigator or Package Explorer view
  - Click the run button arrow on the task bar *
  - Lathe menu that appears



Notes:

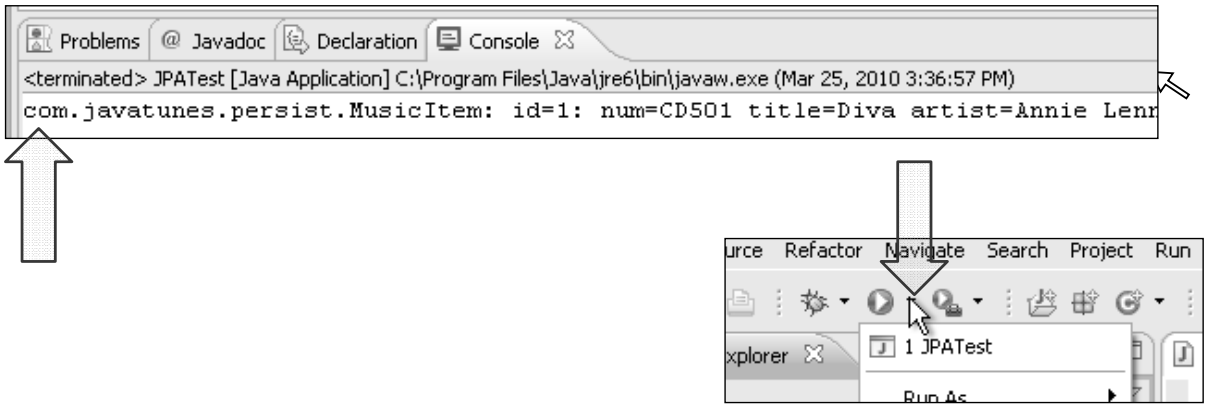
Copyright © 2006-10 LearningPatterns Inc. All rights reserved.

Viewing the Console

Lab

Tasks to Perform

- You will see the results in the console view as shown below
  - If necessary, open the Console (**Window | Show View | Console**)
- To run again, you can press the **Run Icon** as shown at bottom
  - This brings up a list of previously run programs that you can pick from
  - Just select the ***JPATest*** program
  - Errors often show up as exceptions / stack traces in the console



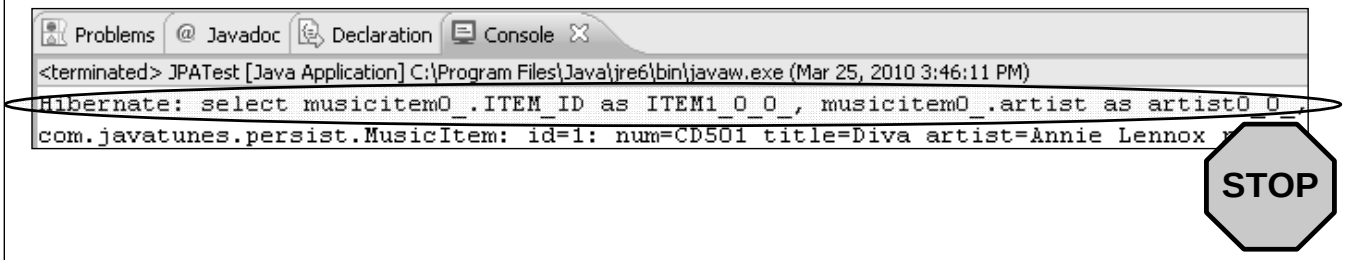
Notes:

Copyright © 2006-10 LearningPatterns Inc. All rights reserved.

Hibernate Logging

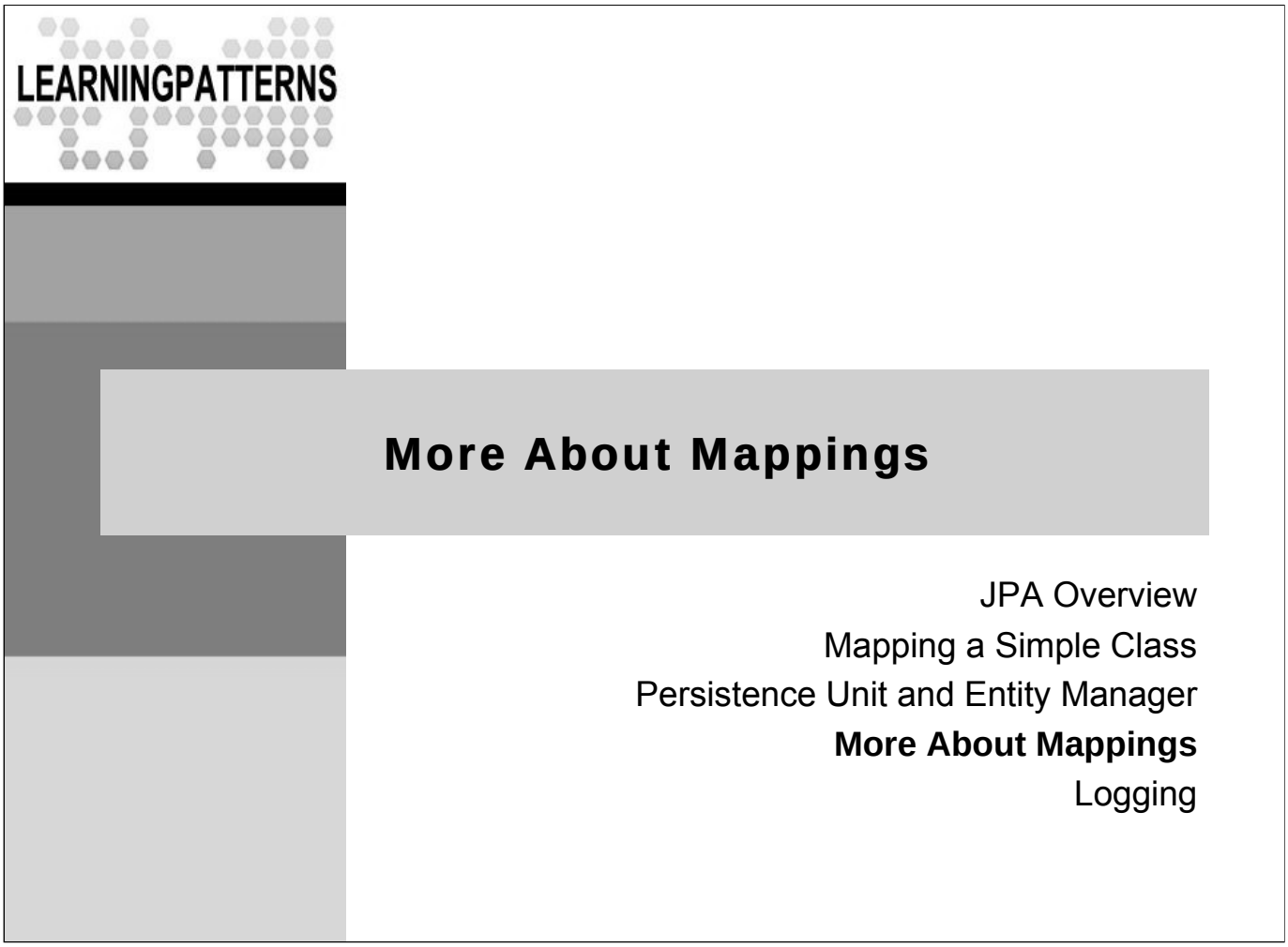
Tasks to Perform

- Hibernate provides a simple way to view SQL queries using the property ***hibernate.show_sql***
  - When set to true will output all SQL queries to the console
  - There is more extensive logging available that we'll look at later
- Open *persistence.xml* and set *hibernate.show_sql* to *true*
  - It is already present in the configuration file
- Run the application again, and you'll see this logging
  - The line that includes "select musicitem0_.ITEM_ID …" is generated by hibernate's logging



Notes:

Copyright © 2006-10 LearningPatterns Inc. All rights reserved.



Notes:

Copyright © 2006-10 LearningPatterns Inc. All rights reserved.

Default Mappings

- JPA has defaults for many basic types, as summarized in the table
  - These types will by default be mapped to a database column with the attribute name, and and can map to the database types shown below

Java type	Database type
String (char, char[])	VARCHAR (CHAR, VARCHAR2, CLOB, TEXT)
Number (BigDecimal, BigInteger, Integer, Double, Long, Float, Short, Byte)	NUMERIC (NUMBER, INT, LONG, FLOAT, DOUBLE)
int, long, float, double, short, byte	NUMERIC (NUMBER, INT, LONG, FLOAT, DOUBLE)
byte[]	VARBINARY (BINARY, BLOB)
boolean (Boolean)	BOOLEAN (BIT, SMALLINT, INT, NUMBER)
java.util.Date	TIMESTAMP (DATE, DATETIME)
java.sql.Date	DATE (TIMESTAMP, DATETIME)
java.sql.Time	TIME (TIMESTAMP, DATETIME)
java.sql.Timestamp	TIMESTAMP (DATETIME, DATE)
java.util.Calendar	TIMESTAMP (DATETIME, DATE)
java.lang.Enum	NUMERIC (VARCHAR, CHAR)
java.util.Serializable	VARBINARY (BINARY, BLOB)

Notes:

- The persistent fields and properties of an entity class are generically referred to in the JPA documentation as the “attributes” of the class

- Note that a particular Java type may be mapped to a number of different database types
  - These are all supported by JPA, though there could conceivably be data loss depending on the Java type and database type
  - For example, a string may be stored in a VARCHAR column that is not large enough for the length of the string, and is truncated
  - If you use JPA to generate your DB schema, you can do things like specifying the length in the column declaration

Copyright © 2006-10 LearningPatterns Inc. All rights reserved.

@Basic and @Column

- The default mapping is the equivalent of annotating an attribute with ***@Basic***
  - It is applicable to any of the types in the previous slide
  - *@Basic* can also be used to change the fetch type (eager/lazy)
  - Generally it is not used when the defaults are fine
- ***@Column*** can be used to modify the mapping to the DB, and supports the following elements
  - ***name***: Name of the column (default – attribute name)
  - ***table***: Name of the table (if a secondary table – covered later)
  - ***insertable***: Is column included in SQL inserts (default true)
  - ***updatable***: Is column included in SQL updates (default true)
  - ***nullable***: Can column hold null values (default true)
  - In addition, there are a number of attributes that are mainly useful if JPA is used for table creation

Notes:

- *@Column* supports a number of elements that are generally only useful if JPA generates the database schema
  - Providers have various tools to support this
  - The elements include the following:
  - ***length***: The column length for strings (default 255)
    - Typically only used during table creation, but some providers may actually validate the data during db operations
  - ***precision***: Precision of a numeric column – typically used in conjunction with scale
  - ***scale***: Number of decimal digits a numeric column can hold

Copyright © 2006-10 LearningPatterns Inc. All rights reserved.

Field and Property Access

- The JPA runtime may access attributes directly, or through their property accessor (get) methods
  - Determined by where you put the access annotations
  - The example below annotated the get method – causing JPA to access them using the accessor methods
  - Previously we annotated fields – resulting in direct field access
  - You must be consistent within a class – either annotating fields or property methods
  - Alternatively, you can use ***@Access*** to explicitly declare the access type (see notes)

```
@Entity // Most detail omitted
public class Event implements java.io.Serializable {

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

Notes:

- @Access can be used to explicitly declare FIELD or PROPERTY (get/set) access for a type
  - You map the type with a particular access type
  - Individual attributes or properties can be designated to use the other access type by annotating the instance variable or get method respectively using @Access
  - In the example below, all the attributes are accessed via field access, except for the date, which uses property access
  - Generally this is not needed

```
@Entity // Most detail omitted
@Access(AccessType.FIELD)
public class Event implements java.io.Serializable {

  @Access(AccessType.PROPERTY)
  public Date getDate() { /*… */ }
  public void setDate(Date dateIn) { /* … */ }
}
```


Copyright © 2006-10 LearningPatterns Inc. All rights reserved.

Temporal (Date/Time) Mappings

- Temporal mappings can get somewhat complicated
  - There are multiple Java and database types, and the default mappings between them are:
  - ***java.sql.Date***: *DATE*
  - ***java.sql.TIME***: *TIME*
  - ***java.sql.Timestame***: *TIMESTAMP*
- ***@Temporal*** is used to map *java.util.Date* and *Calendar* to the appropriate SQL type, as shown at bottom
  - These Java types are not specific enough for JPA to determine the DB type, so *@Temporal* must be used
  - Can specify either *DATE*, *TIME*, or *TIMESTAMP*

```
@Temporal(TemporalType.DATE)
private Date date;
```

Notes:

- *java.util.Dat*e and *java.util.Calendar* do not supply enough information for JPA to map to the database

- *java.sql.Date* will map to a *DATE* column by default
  - Our date property in *MusicItem* is of type *java.util.Date*, so we will need to use *@Temporal*

Copyright © 2006-10 LearningPatterns Inc. All rights reserved.

Mapping Enums

- Enum mapping is supported in two ways
  - By default, they are mapped using the enum integer values
- ***@Enumerated*** can be used to modify the mapping via specifying an EnumType of:
  - ***ORDINAL***: Map as integer value of enum (the default)
  - ***STRING***: Map as string value of enum
  - String values can be useful if people read the raw DB data

```
public enum EventType { PUBLIC, PRIVATE}
```

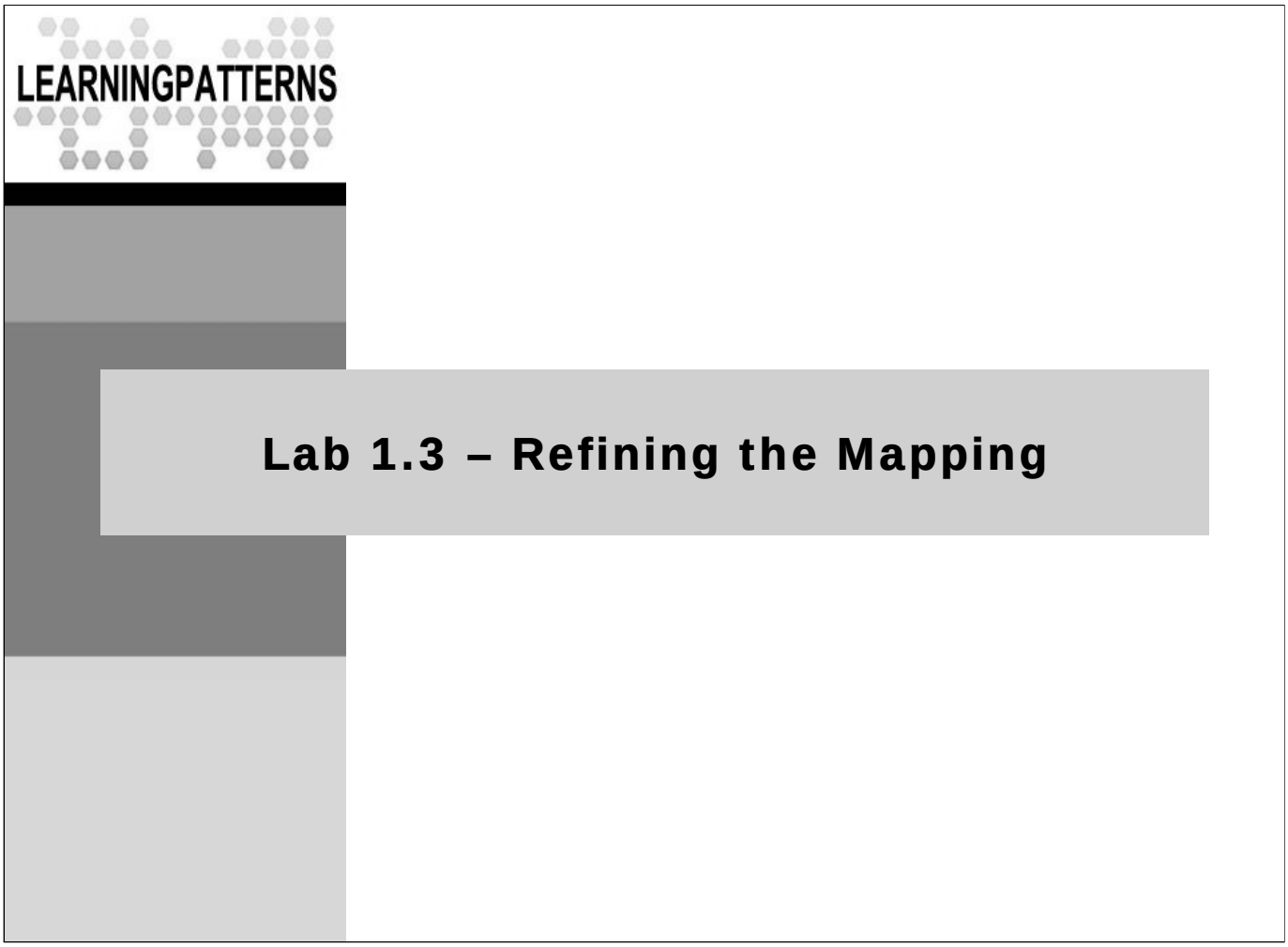
```
@Entity // Most detail omitted
public class Event implements java.io.Serializable {

  @ENUMERATED(EnumType.STRING) // Store in DB as string value
  private EventType typeOfEvent;
}
```

Notes:

- Enums are typically used as constants

Copyright © 2006-10 LearningPatterns Inc. All rights reserved.



Notes:

Copyright © 2006-10 LearningPatterns Inc. All rights reserved.

Lab 1.3 – Refining the Mapping

- **Overview**: In this lab we will refine the *MusicItem* mapping
  - There's not much that we can do, since it's a simple class with the table already defined, but we'll refine how the release date is mapped
- **Objectives**: Use more mapping capabilities
- **Builds on previous labs**: Lab 1.2
  - Continue working in your **Lab01.1** directory for this lab
- **Approximate Time**: 15-20 minutes

Notes:

Copyright © 2006-10 LearningPatterns Inc. All rights reserved.

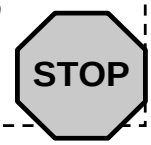
Refine the ReleaseDate Mapping

- The *releaseDate* property is of Java type *java.util.Date*
  - It's mapping is actually undefined in the spec, though typically implementations will choose a type to map to (e.g. timestamp or date)

Tasks to Perform

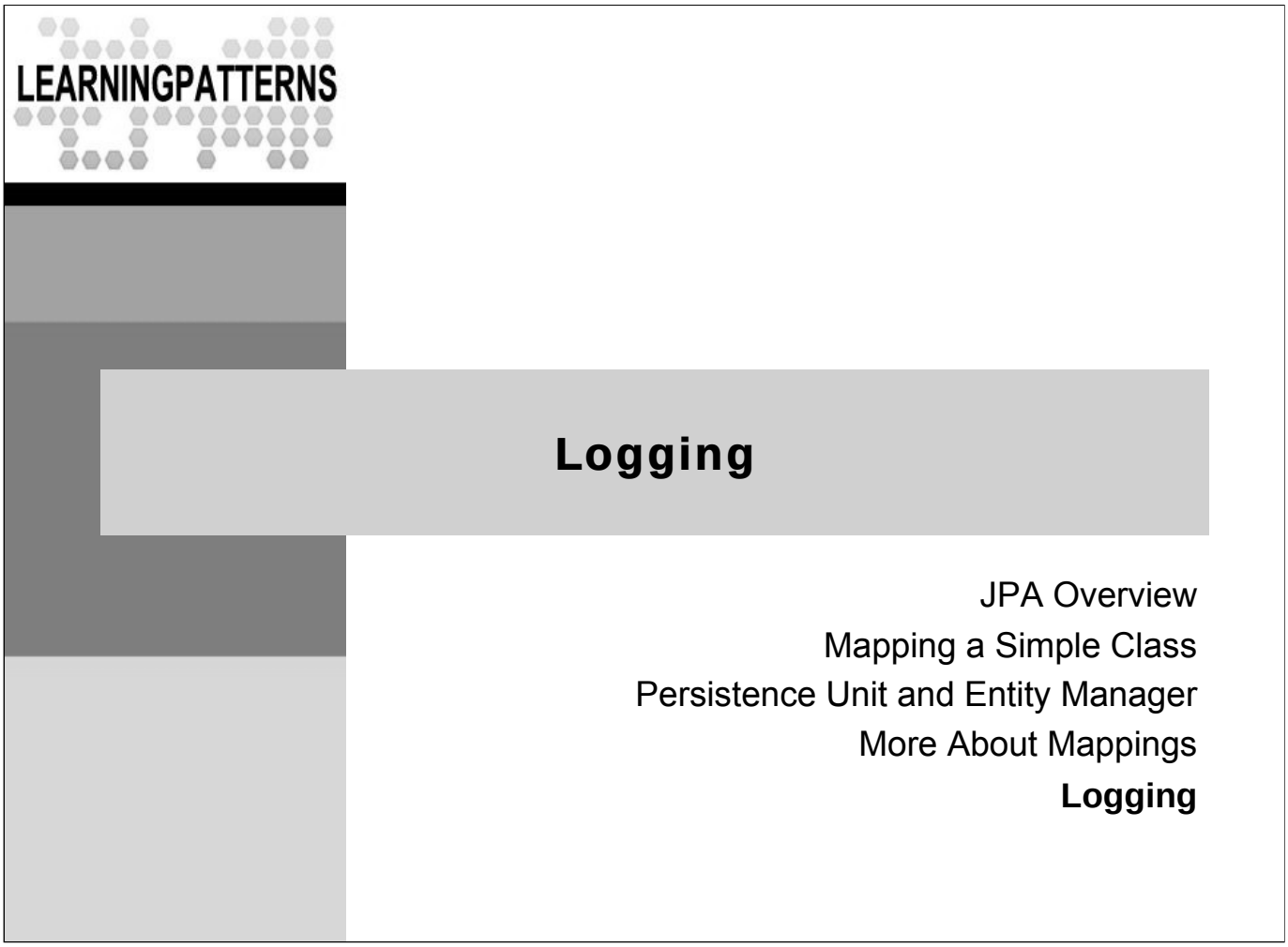
- In your test client, print out the type of the *releaseDate* property of the item retrieved with *find()*
  - Easy to get this with the code below (if m is your *MusicItem*)

    m.getReleaseDate().getClass().getName()
  - The Hibernate provider defaults this to a timestamp
- Next, in your MusicItem class, modify the date mapping by annotating the field with ***@Temporal*** to specify that it is a date
  - See the earlier slides for details if you need to
  - Run the program again – you should see that the *releaseDate* property is now a *java.sql.Date*
  - You've modified the mapping !



Notes:

Copyright © 2006-10 LearningPatterns Inc. All rights reserved.

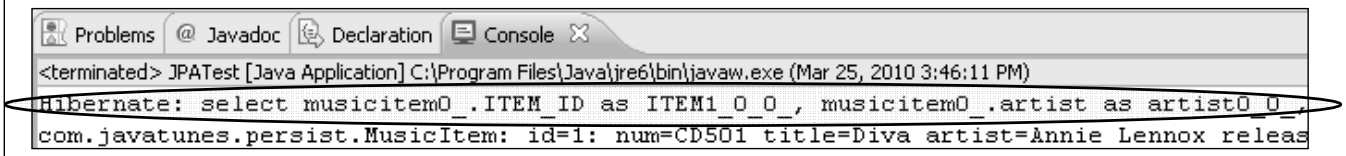


Notes:

Copyright © 2006-10 LearningPatterns Inc. All rights reserved.

hibernate.show_sql

- Since our JPA 2 provider is Hibernate based, we will be looking at Hibernate logging in this section
- Your console output for the labs will have shown you some debugging output
  - The line that includes "select musicitem0_.ITEM_ID …" is output that is produced because we set ***hibernate.show_sql*** to true in *persistence.xml* in an earlier lab
  - Hibernate also includes the capability to view a much larger amount of logging output



Notes:

Copyright © 2006-10 LearningPatterns Inc. All rights reserved.

Simple Logging Facade for Java - SLF4J

- Hibernate uses **SLF4J** to do its logging
- SLF4J can direct your logging output to several logging frameworks depending on your chosen binding
  - NOP, Simple, log4j version 1.2, JDK 1.4 logging, JCL or logback
- To setup logging properly you need *slf4j-api.jar* in your classpath together with the jar file for your preferred binding
  - *slf4j-api.jar* comes with the Hibernate distribution
  - You will need a jar for your binding, and the actual logging implementation
- For the labs, we have chosen to use log4j, and supplied the SLF4J binding file (*slf4j-log4j12.jar*) as well as the log4j jar
  - These are included in the lab's *workspace\common\lib* directory
  - By including them in the classpath, you selected log4j as your logging implementation for the labs

Notes:

- We will work with log4j, since it is a widely used, very powerful logging framework
  - To use log4j you will need to place a *log4j.properties* file in your classpath
- If you wanted to use a different logging implementation, you'd need to get the jar for that binding from the slf4j website (http://www.slf4j.org), and include it in the classpath (as well as any other needed files for the logging implementation)

Copyright © 2006-10 LearningPatterns Inc. All rights reserved.

Apache Log4J

- Log4j is an open source framework that allows developers to control how log statements are output with arbitrary granularity
  - Hibernate is already set up to produce log4j output – i.e. it includes logging statements in its implementation
- It is possible to configure the logging output from Hibernate via the log4j configuration file
  - To do this, we'll need to include the log4j library (log4j-nnn.jar) and configuration file (log4j.properties) in our labs
  - This will allow us to explicitly configure and then view the log4j output from Hibernate
- We don't go deeply into the details of Log4j here

Notes:

- The logging statements in the Hibernate distribution may or may not produce output depending on how you configure log4j

Copyright © 2006-10 LearningPatterns Inc. All rights reserved.

Hibernate log4j.properties file

- The Hibernate distribution ships with a sample *log4j.properties* file which includes entries for the major loggers that Hibernate writes to in it's implementation
  - If you want to configure the debugging information that your Hibernate programs output, you can start with this file
  - It is usually in the *[hibernate]\doc\tutorial\src directory*
  - **We supply it for you** in the lab setup – so for the labs it is already available
- The Hibernate reference manual says the following:
  - "**We strongly recommend** that you familiarize yourself with Hibernate's log messages. A lot of work has been put into making the Hibernate log as detailed as possible, without making it unreadable. It is an essential troubleshooting device."

Notes:

Copyright © 2006-10 LearningPatterns Inc. All rights reserved.

The log4j.properties file

```
### 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

log4j.rootLogger=warn, stdout

log4j.logger.org.hibernate=info
## More detail omitted ...
```

- This sample from the Hibernate distribution shows:
  - An **appender**, stdout, which directs output to the console, with a logging level of warn
  - The **root logger**, which sets up defaults for all loggers
  - A **hibernate logger**, org.hibernate, with a logging level of info
    - With output to the stdout appender inherited from the root logger

Notes:

- Any line starting with "#" in the log4j.properties file is treated as a comment

Copyright © 2006-10 LearningPatterns Inc. All rights reserved.

Modifying log4j.properties for Hibernate

- It's easy to change the debugging messages that Hibernate outputs
  - Output from log4j is controlled and organized by categories
  - For example, the line below states that all messages from the category org.hibernate, of info level or above, should be output

```
log4j.logger.org.hibernate=info
```

  - If you want even more error messages, then you could change that level to be all messages of debug level (the lowest level) or above by changing that line to read

```
log4j.logger.org.hibernate=debug
```

  - You could eliminate that category entirely, by commenting it out

```
#log4j.logger.org.hibernate=info
```

    - The file supplied in the labs has all categories commented out, so there has been no Hibernate logging
  - The main Hibernate logger categories are shown in the next slide

Notes:

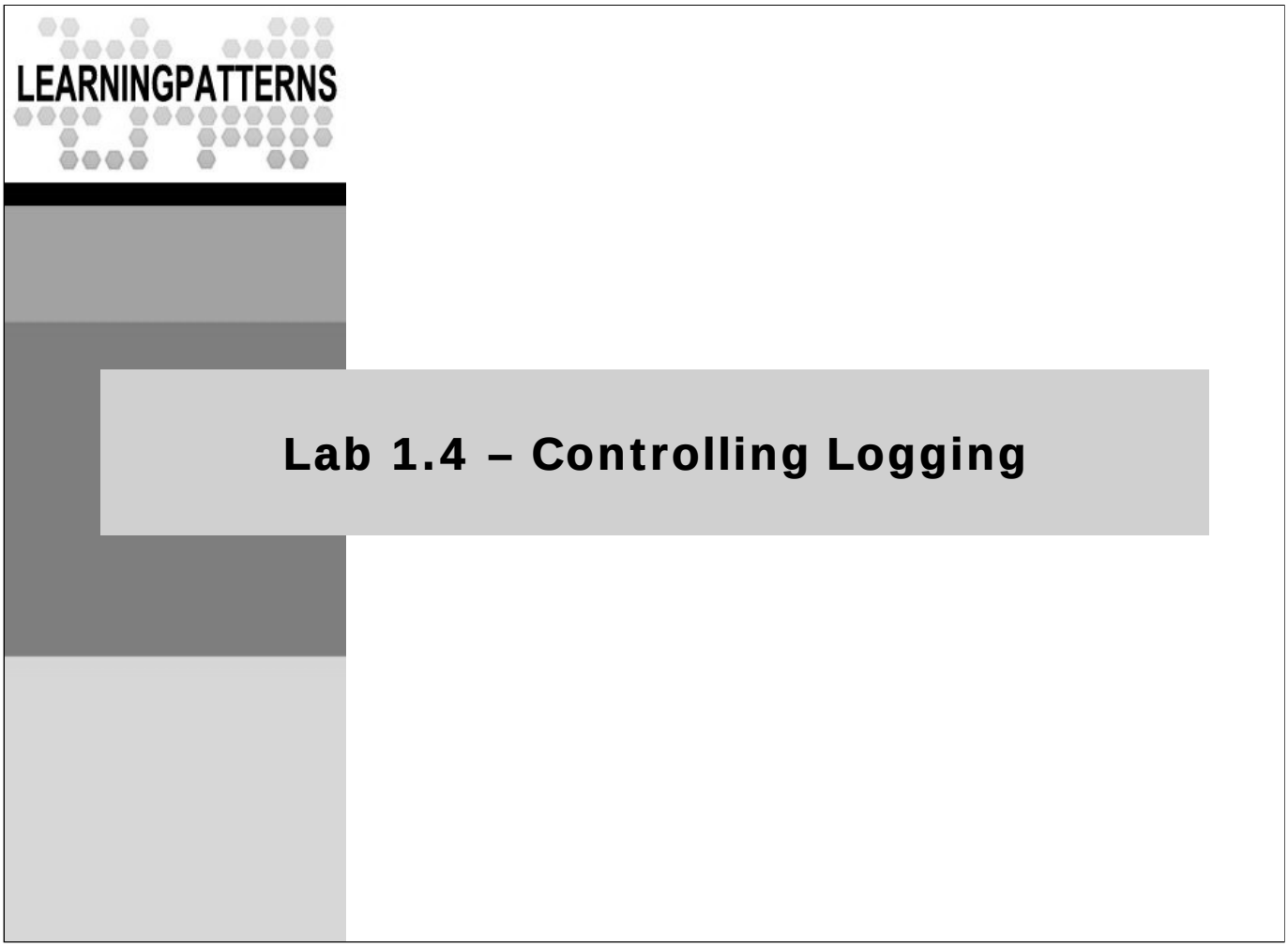
Copyright © 2006-10 LearningPatterns Inc. All rights reserved.

Hibernate Logging Categories

- **org.hibernate.SQL**: Log all SQL DML statements as they are executed
- **org.hibernate.type**: Log all JDBC parameters
- **org.hibernate.tool.hbm2ddl**: Log all SQL DDL statements as they are executed
- **org.hibernate.pretty**: Log the state of all entities (max 20 entities) associated with the session at flush time
- **org.hibernate.cache**: Log all second-level cache activity
- **org.hibernate.transaction**: Log transaction related activity
- **org.hibernate.jdbc**: Log all JDBC resource acquisition
- **org.hibernate.hql.ast.AST**: Log HQL and SQL ASTs during query parsing
- **org.hibernate.secure**: Log all JAAS authorization requests
- **org.hibernate**: Log everything (a lot of information, but very useful for troubleshooting)

Notes:

Copyright © 2006-10 LearningPatterns Inc. All rights reserved.



Notes:

Copyright © 2006-10 LearningPatterns Inc. All rights reserved.

Lab – Controlling Logging

- **Overview**: In this lab you will modify the logging output that the provider (Hibernate) is producing
  - We already provide all the needed jars, and the log4j.properties file
  - We'll modify the log4j.properties file to produce different levels of logging output
- **Objectives**:
  - Learn about log4j configuration, and modify the log4j logging output
- **Builds on previous labs**: Lab 1.3
- **Approximate Time**: 20-25 minutes

Notes:

Copyright © 2006-10 LearningPatterns Inc. All rights reserved.

Modify the log4j.properties file

Tasks to Perform

- In your Eclipse project, open src*log4j.properties* for editing
  - Run the program after each change to logging to see the results
  - Uncomment the following category to see general logging

    log4j.logger.org.hibernate=info
  - To see much more extensive logging, change the level to ***debug***
  - Uncomment the following category, so that log4j will print the SQL that Hibernate is using

    log4j.logger.org.hibernate.SQL=debug
  - Modify the JDBC bind parameter output, to get more detail, so we can see exactly what values are being used in queries
  - Comment out the info level line, and uncomment the trace line

    #log4j.logger.org.hibernate.type=info

    log4j.logger.org.hibernate.type=trace

Notes:

- Of course, you want to open the log4j.properties file that is in your Eclipse project

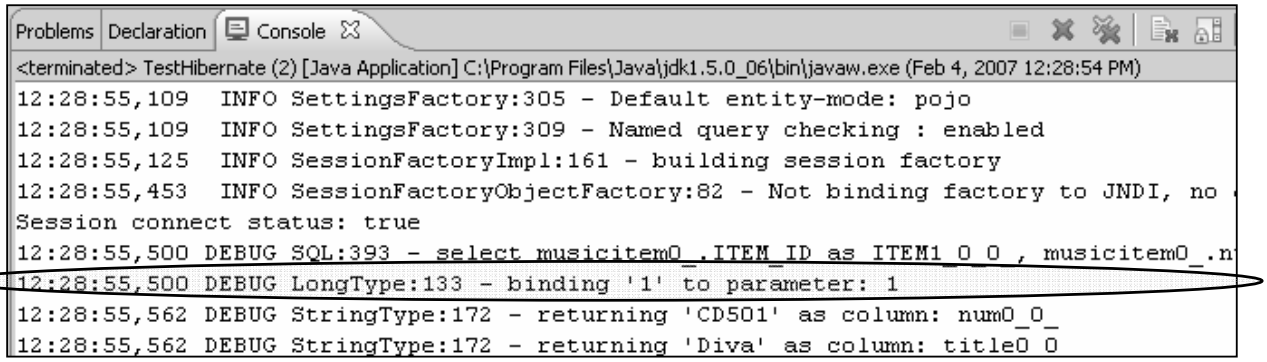
Copyright © 2006-10 LearningPatterns Inc. All rights reserved.

Run Your Program

Lab

Tasks to Perform

- You should see more detail on the JDBC bind parameters after the last change (as shown below)
  - This is useful to see the actual values used in the SQL
- Try some of the other logging loggers for a few minutes



Notes:

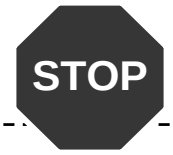
Copyright © 2006-10 LearningPatterns Inc. All rights reserved.

Look at the Documentation !

Tasks to Perform

- Once done, comment out the **log4j.logger.org.hibernate** logger
  - This produces a lot of output that you don't normally need
- Decide what level of Hibernate logging you want to use
  - If the original show_sql logging is enough for you, then comment out the loggers in log4j.properties
  - If you want to use the Hibernate logging, then you can comment out the *show_sql* property in *persistence.xml*
  - The Hibernate SQL category gives us more control over this

```
<!-- <property name="hibernate.show_sql">true</property>  -->
```



Notes:

Copyright © 2006-10 LearningPatterns Inc. All rights reserved.

Review Questions

- Why do we need something like the Java Persistence API?
- What does Java Persistence do?
- What is an entity?
- How do you map an entity to the database?
- What is the id property of an entity?
- What is a persistent context
- What does the entity manager do?

Notes:

Copyright © 2006-10 LearningPatterns Inc. All rights reserved.

Lesson Summary

- Object-relational mapping (ORM) is a difficult task
  - It takes a great deal of effort to write a persistence layers to store and retrieve data from the database
  - The object and relational models have differences that add to this difficulty
- ORM tools automate this task for us
  - It allows you to define a mapping in some way, rather than write code to implement that mapping
- The Java Persistence API provides **ORM capability** for using a **Java OO domain model** to manage a relational database
  - POJO based, supporting inheritance and associations
  - Provides an object based query language

Notes:

Copyright © 2006-10 LearningPatterns Inc. All rights reserved.

Lesson Summary

- An entity is a **lightweight persistent domain object**
  - Metadata is used to describe the mapping to the database
  - Annotations are usually used, but XML files may also be used
- Entities must had an id that uniquely identifies them
  - The ids may be auto-generated in the database
  - Sequences and identity rows are both supported
- Entities are managed by an entity manager
  - A **persistence context** is comprised of the set of managed entities within an entity manager
  - **Only one instance** with a given persistent identity can exist within a persistence context
  - All interactions with the database are actually done using the entity manager

Notes:

Copyright © 2006-10 LearningPatterns Inc. All rights reserved.