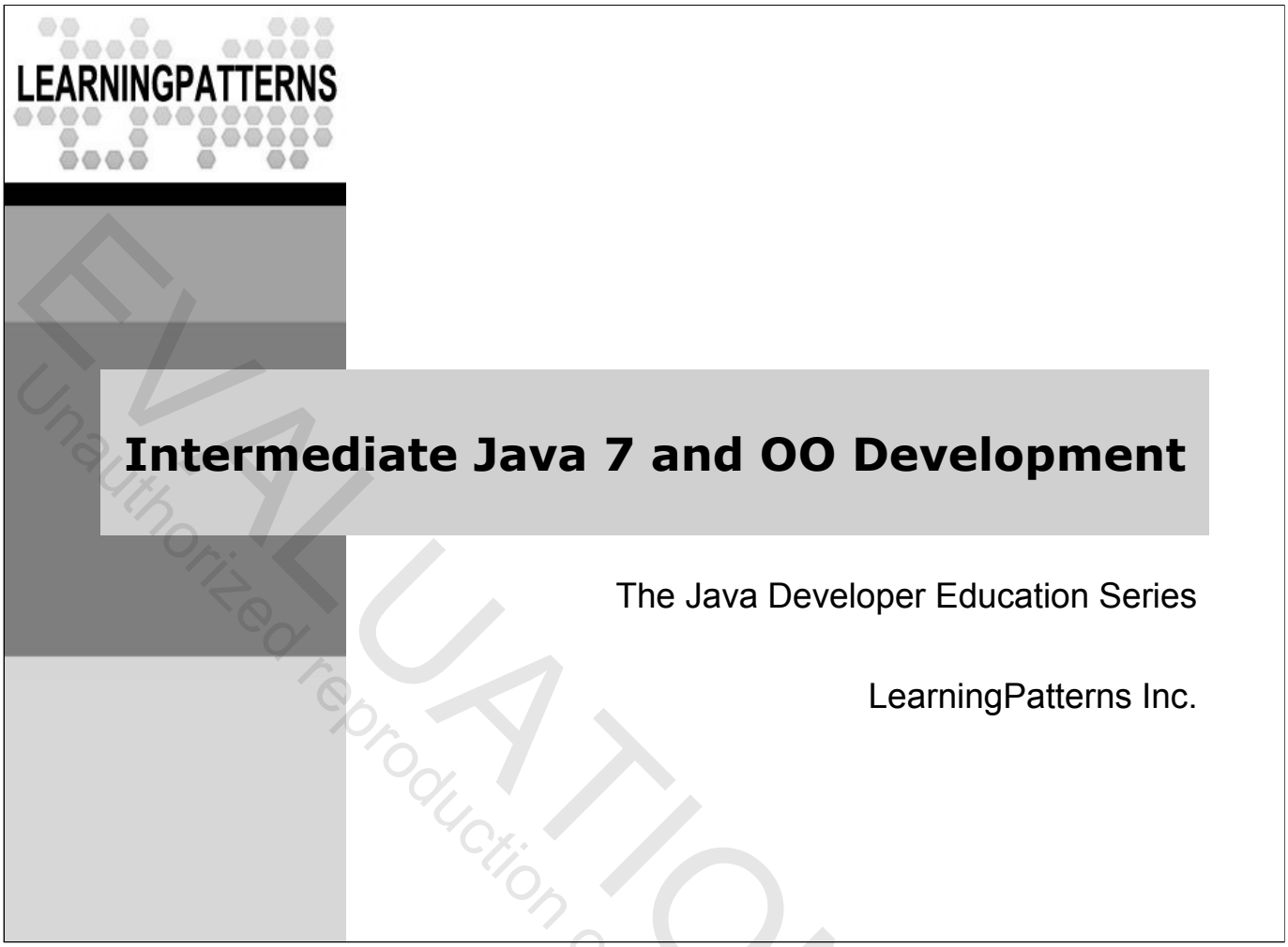
Table of Contents – Intermediate Java 7



Course Overview

- A very full intermediate level course:
  - Provides a strong grounding in using Java productively
  - Goes well beyond the basics
  - Covers important topics such as composition, inheritance, polymorphism, interfaces and exceptions, and JDBC/JPA
  - Also covers tools such as JUnit, and log4j to make your programming more productive and your code of higher quality
- Be prepared to work hard and learn a great deal!
- The course contains numerous hands-on labs
  - They exercise all the important concepts discussed
  - The lab solutions for the course are provided to you
- The course is suitable for Java 7 and later

Introduction

- The Java platform has evolved rapidly
  - However, many people are still using older versions
- The labs are designed to support those using Java 5 and any later versions

- Java 6 and 7 primarily introduced advanced features for running Java programs, and did not change the API, though it did add to it
  - However, those features are mostly beyond the scope of this course
  - Accordingly, this course can be used in any of these environments

Course Outline

- Session 1: **Getting Started**
- Session 2: **Review: Class and Object**
- Session 3: **Review: Flow of Control, String and Array**
- Session 4: **Review: Packages**
- Session 5: **Composition and Inheritance**
- Session 6: **Interfaces**
- Session 7: **Exceptions**
- Session 8: **Collections and Generics**
- Session 9: **Database Access with JDBC and JPA**
- Session 10: **Additional Language Features**
- Session 11: **Java I/O** (optional)
- Session 12: **Nested / Inner Classes**
- Session 13: **Reflection**
- Session 14: **Introduction to ant**
- Session 15: **maven Overview**
- Session 16: **Introduction to JUnit**
- Session 17: **Organizing Tests with JUnit**
- Session 18: **Introduction to Logging and log4j**

Workshop Objectives - Java

- Review Java basics and basic OO programming concepts
- Understand advanced OO principles such as inheritance and polymorphism
- Use Java packages to organize code
- Understand interfaces, their importance, and their uses
- Understand how exceptions are used for error handling
- Understand the basics of database access with JDBC and JPA
- Learn the basics of the Collections Framework
- See some of the new/advanced Java language features
- Understand and use basic I/O streams
- Understand and use inner classes
- Understand Java reflection, and examine classes at runtime

Introduction

Workshop Objectives - Tools

- **ant**:
  - Understand how ant and buildfiles work
  - Create buildfiles, and use ant to control a build
- **maven**:
  - Understand the basics of maven for managing dependencies and building projects
- **JUnit**:
  - Be familiar with testing and test-driven development
  - Use JUnit to create good testing structures for your Java code
- **Logging**:
  - Understand and use log4j for logging
  - Be aware of other logging choices available in Java

Course Methodology

- This course **reviews basic Java** and Object-Oriented (OO) programming
  - **Basic Java knowledge** and programming experience is required
  - Reasonable comfort in using Java is required to do the labs in a timely manner
- Covers more advanced concepts in depth
  - Including composition, inheritance, exceptions, JDBC, inner classes, reflection
  - Including the tools; ant, JUnit and log4j
- The concepts and syntax are reinforced by frequent practice in the form of hands-on labs

Labs

- The workshop has numerous hands-on lab exercises, structured as a series of brief labs
  - The detailed lab instructions are separate from the main student manual
- Setup zip files are provided with skeleton code for the labs
  - Students add code focused on the topic they're working with
  - There is a solution zip with completed lab code
- Lab slides have an icon like in the upper right corner of this slide
  - The end of a lab is marked with a stop like this one:

Typographic Conventions

- Code in the text uses a fixed-width code font, e.g.:

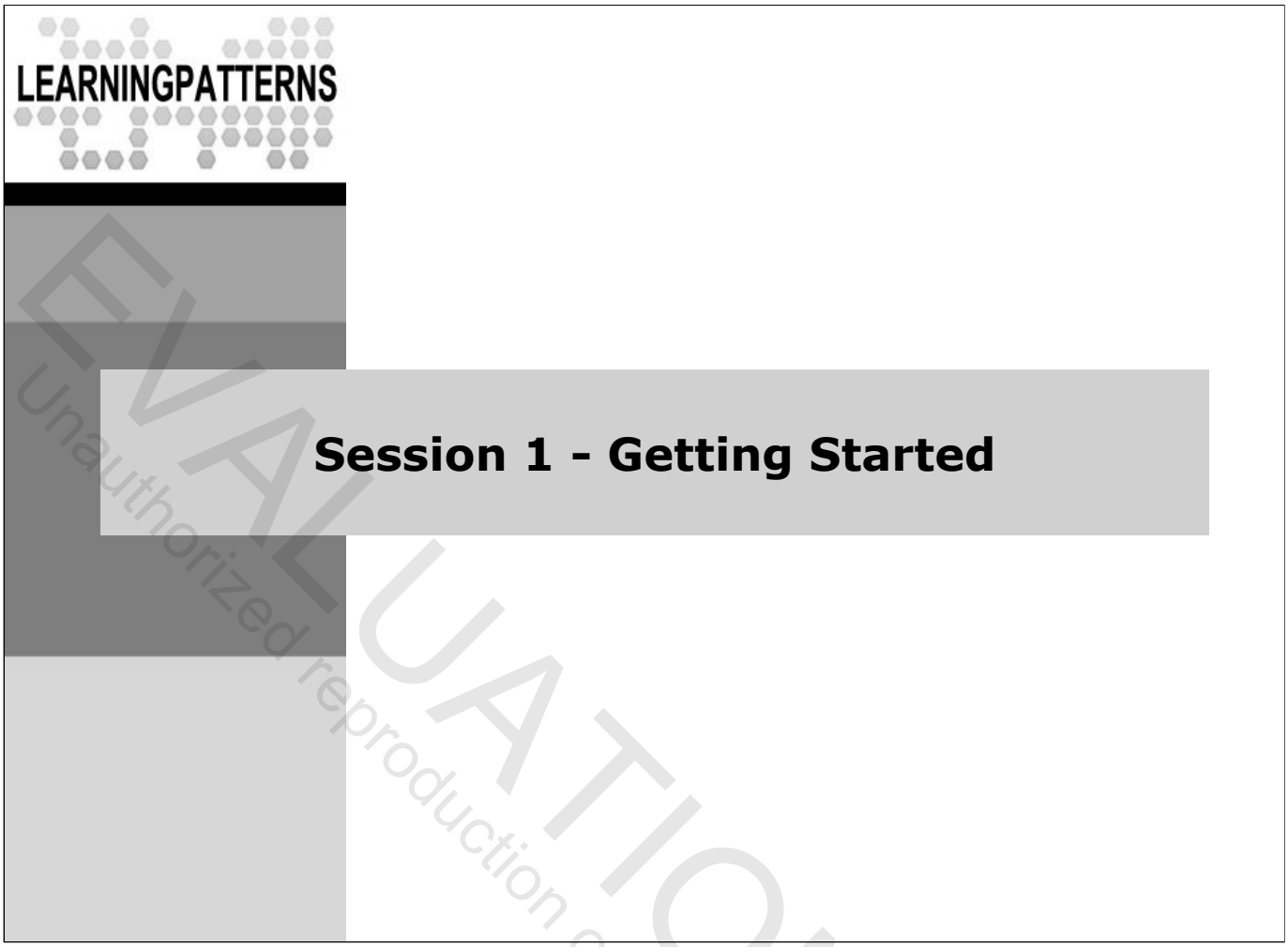
```
JavaInstructor teacher = new JavaInstructor()
```

  - Code fragments use the same font, e.g. `teacher.teach()`
  - We **bold/color** text for emphasis
  - Filenames are in italics, e.g. *JavaInstructor.java*
  - We denote more info in the notes with a superscript number (1) or a **star** *
  - Longer code examples appear in a separate code box - e.g.

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

Introduction

- (1) If there was more info about the slide content with the superscript (1), we would put it here
- Additional notes would appear here



Session Objectives

- Look at a simple Java program, and how it is compiled and run
- Set your computer up for Java development
  - Setting paths, environment variables, etc.
- Use a text editor to type in a simple Java class
- Use the Java Development Kit to compile and run the code

Session 1: Getting Started

A Simple Application Class

- To get started, we'll look at a very simple Java application, that displays the string "Hello World"
  - This is a simple non-graphic standalone application that runs on its own
- All programs in Java are just a **class** with a **`main`** method in it
  - The HelloWorld class needs to appear in a file **HelloWorld.java**
- The **`main`** method is a special method that is the starting point for every Java application
  - It must be declared as shown, and has to appear in a class
  - `System.out.println` how you print things to the console

```
public class HelloWorld {
   public static void main(String[] args) {
      System.out.println("Hello World");
   }
}
```

Session 1: Getting Started

- Java classes can be deployed into several runtime execution environments, including:
  - Standalone applications that are run explicitly, like our Hello World application.
  - Servlets, which are run in a Web server.
  - Applets, which are downloaded from a Web server and run in a Web browser.
- We will concentrate on applications in this course.
- `main` is the special method which is the starting point for every standalone Java application.
  - Applets and servlets work a little differently
- The `main` method has to appear inside a class.
  - `main` takes an argument – an array of `String`s.
  - This array holds arguments that are given on the command line when the application is run.
- `System.out.println` causes something to be printed to `System.out`, which is the standard output (the console) – more on this later.

Compiling HelloWorld

- To use a Java class, you have to **compile** the Java code

```
C:\StudentWork>javac HelloWorld.java
```

  - After compiling, a file called ***HelloWorld.class*** (containing Java **bytecode**) is created

- You use the Java Virtual Machine (**JVM**), to run a Java program
  - The JVM comes with Java as the **`java`** executable

```
C:\StudentWork>java HelloWorld
```

  - This would produce the output "Hello World" printed in your command prompt window

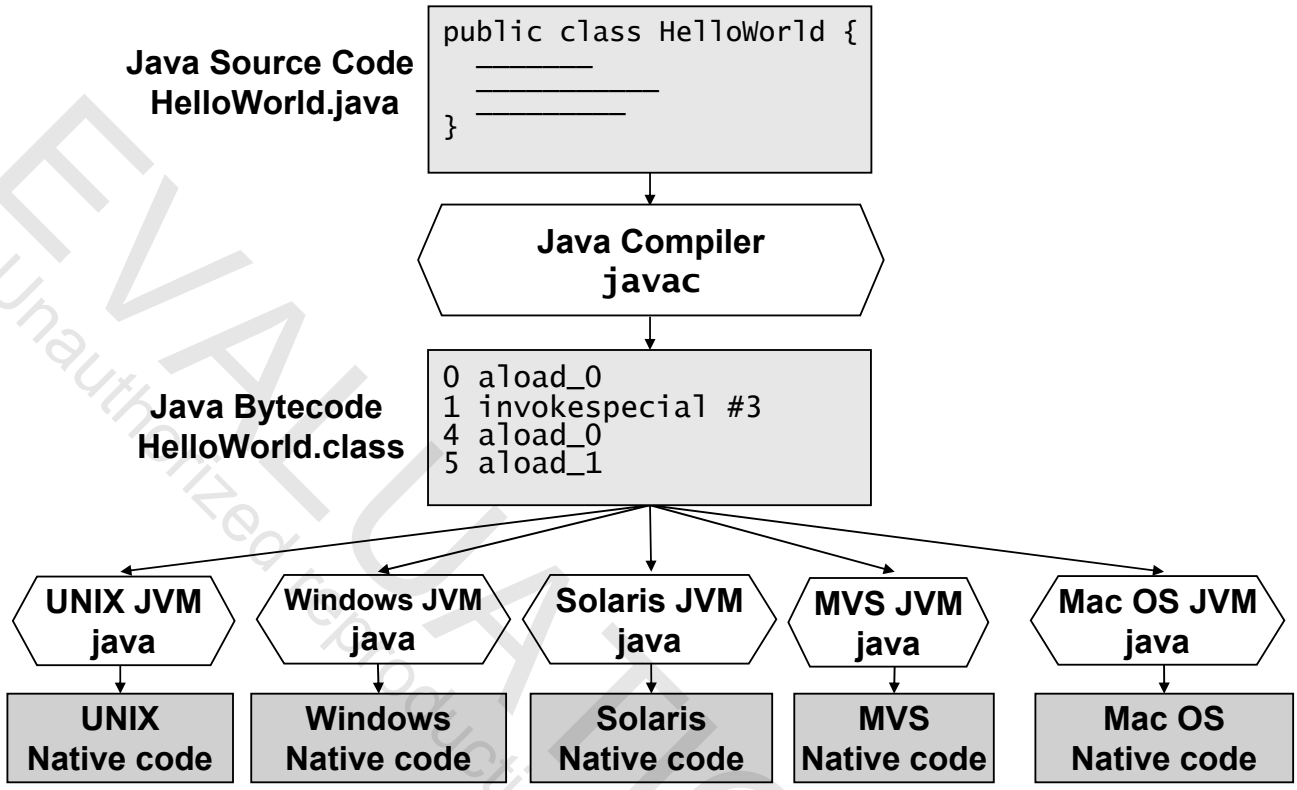
- *HelloWorld.class* is the bytecode for the program produced by the compiler

- We will talk more about all these details later in the course
- For now, we're giving you enough information to just run the program

Java Source and Java Bytecode

- You write a program in the Java programming language
  - And store it in a .java file, for example **MyClass.java**
- Before running it, you compile it, which translates it into an intermediate language called **Java bytecode**
  - Stored in a .class file, for example **MyClass.class**
  - These are platform independent codes that are interpreted by an interpreter, the **Java Virtual Machine**, or **JVM**
  - You can think of bytecode as the native instructions for the JVM
  - Bytecode helps make "write once run anywhere" possible
- The JVM converts the bytecode into the native code for the target machine it is running on

Life Cycle of a Java Program

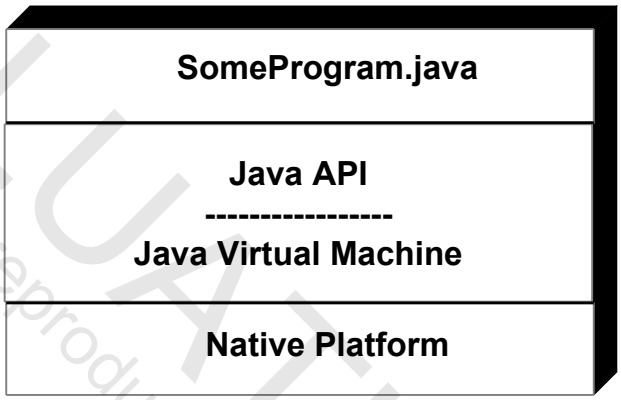


Session 1: Getting Started

- The JVMs for Solaris and Windows are available from Sun Microsystems.
- JVMs for other platforms are available from their respective vendors.

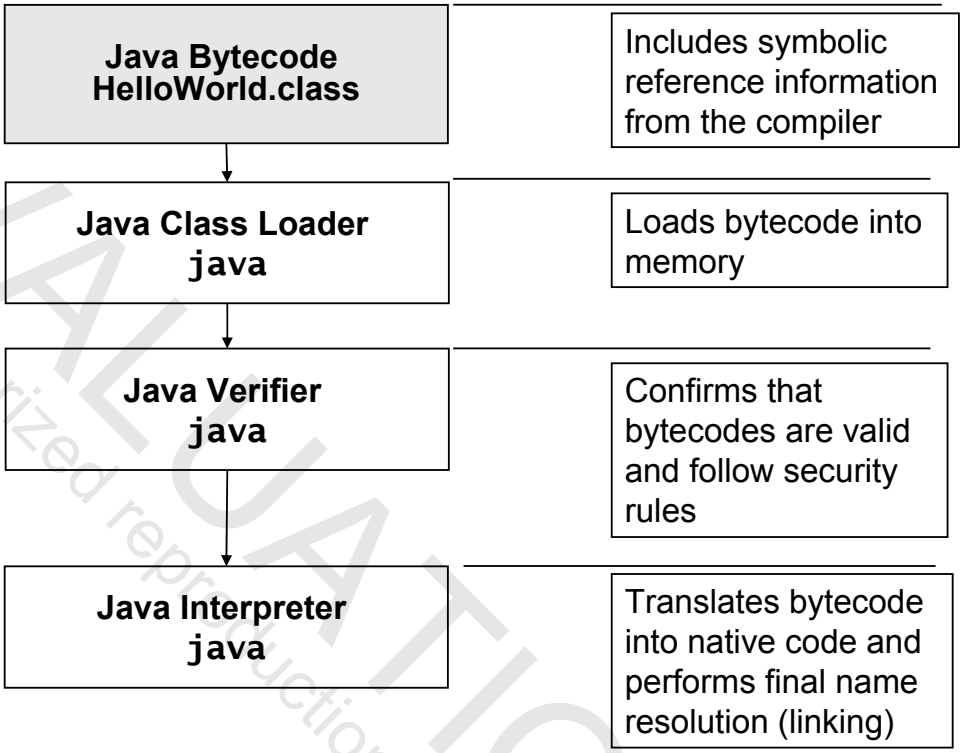
Java Programs Insulated From Environment

- Java provides a platform that programs can run on independently of the environment
  - It is a software only platform made up of the Java API and the JVM



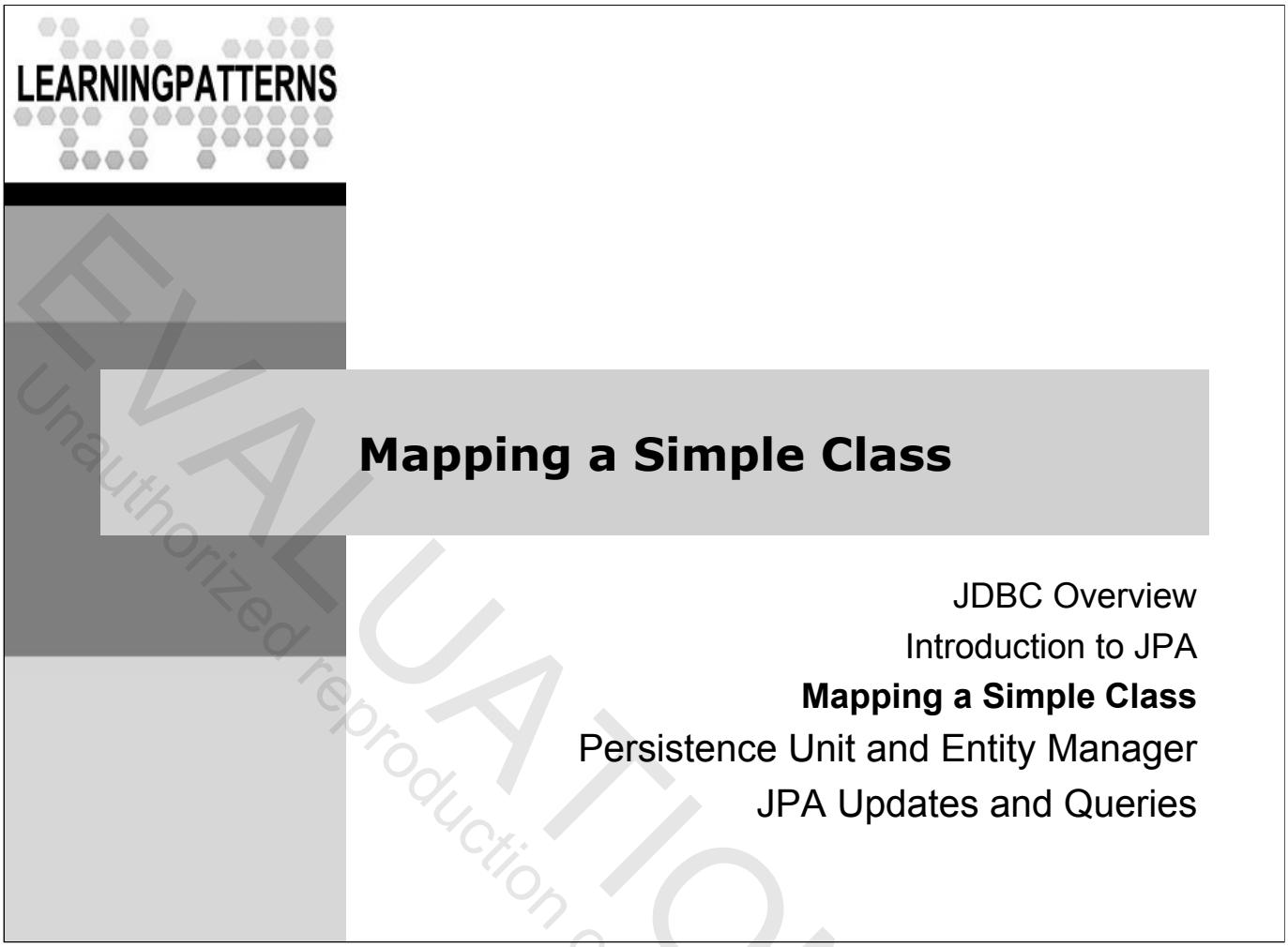
Java is Dynamic - The Runtime Process

- Several phases take place at runtime



Lab 1.1 - HelloWorld

In this lab, we'll become familiar with the Eclipse development environment and run a simple Java program



Entity Classes

- Entity: A **lightweight persistent domain object**
  - Represents data stored in a DB
- Entity metadata describes the mapping to the DB
  - With **annotations** (usually preferred) or XML
  - Core annotations are in the package **`javax.persistence`**
- Entities must:
  - Implement a **no arg constructor** (so JPA can instantiate it)
  - Contain fields to **hold persistent state**
    - Must be **non-public** - clients use get/set methods for access
  - Provide an **identifier** (usually called id - maps to DB primary key)
- Persistence operations (read, write, etc.) in a separate class
  - **`EntityManager`**, covered later

- 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, but they:
  - 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
- 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

Annotation Overview

- Mechanism for adding metadata to your source code
  - It's a **rubber stamp** usable on fields, methods, types, etc
- Annotations are just special comments that can be inspected
  - Used by tools (like the JPA runtime), which read them
  - Annotations start with an @ - `@Entity`
- Think of an envelope stamped **Fragile** and **Next Day**
  - They don't change the envelope or its contents
  - They might change how a postal worker processes the envelope
- Similarly, JPA annotations affect how JPA processes entities
  - By generating JDBC/SQL code conforming to their annotations
  - In the end, saving you a lot of work !!

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
}
```

- 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

javax.persistence.Entity Annotation

- **@Entity** declares the class to be a persistent entity
- Entity name default: Unqualified name of the class (Event)
  - Used in queries and other places (more later)
  - The name can also be set with the name element of `@Entity`

  `@Entity(name="OurEventEntity")`
- Table name default: Entity name (Event)
  - Use **@Table** to declare a table name, as shown below
  - For example, if the table was called "EVENTS" you could use:

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

- `@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

The Event Class

- The event class has three properties
  - **id** (`Long`), **date** (`java.sql.Date`), **title** (`String`)
  - id holds a unique identifier for each event
  - Provides get/set methods for all the properties
  - Has no-argument constructor
- Default: **all non-transient properties are persistent**
  - They are stored in the database
  - Default: Column name is the same as the property name
  - The DB column type also uses reasonable defaults
- Annotate non-persistent properties with **`@Transient`**
  - Fields that use the Java `transient` modifier are also not persisted

- 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

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

- 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

Field Access or Property Access

- In class `Event`, JPA will access the persistent fields directly
  - Because we annotated the persistent fields
  - It's possible to annotate the accessor methods
    - JPA will then access the properties via those methods
  - Useful if you have logic in the accessors (e.g. validation)
- Don't mix field and accessor styles within a class
  - This is not portable

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

- *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]

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)
);
```

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

Generated Id Property

- **@GeneratedValue** specifies a primary key generation strategy - possible values 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;
```

- `@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

Mapping Properties

- Default: Properties are persistent and mapped to columns with the property name
  - Change this as needed
  - Below, we map the date property to the EVENT_DATE column
  - The title property maps to the TITLE column by default
    - So we leave it as is

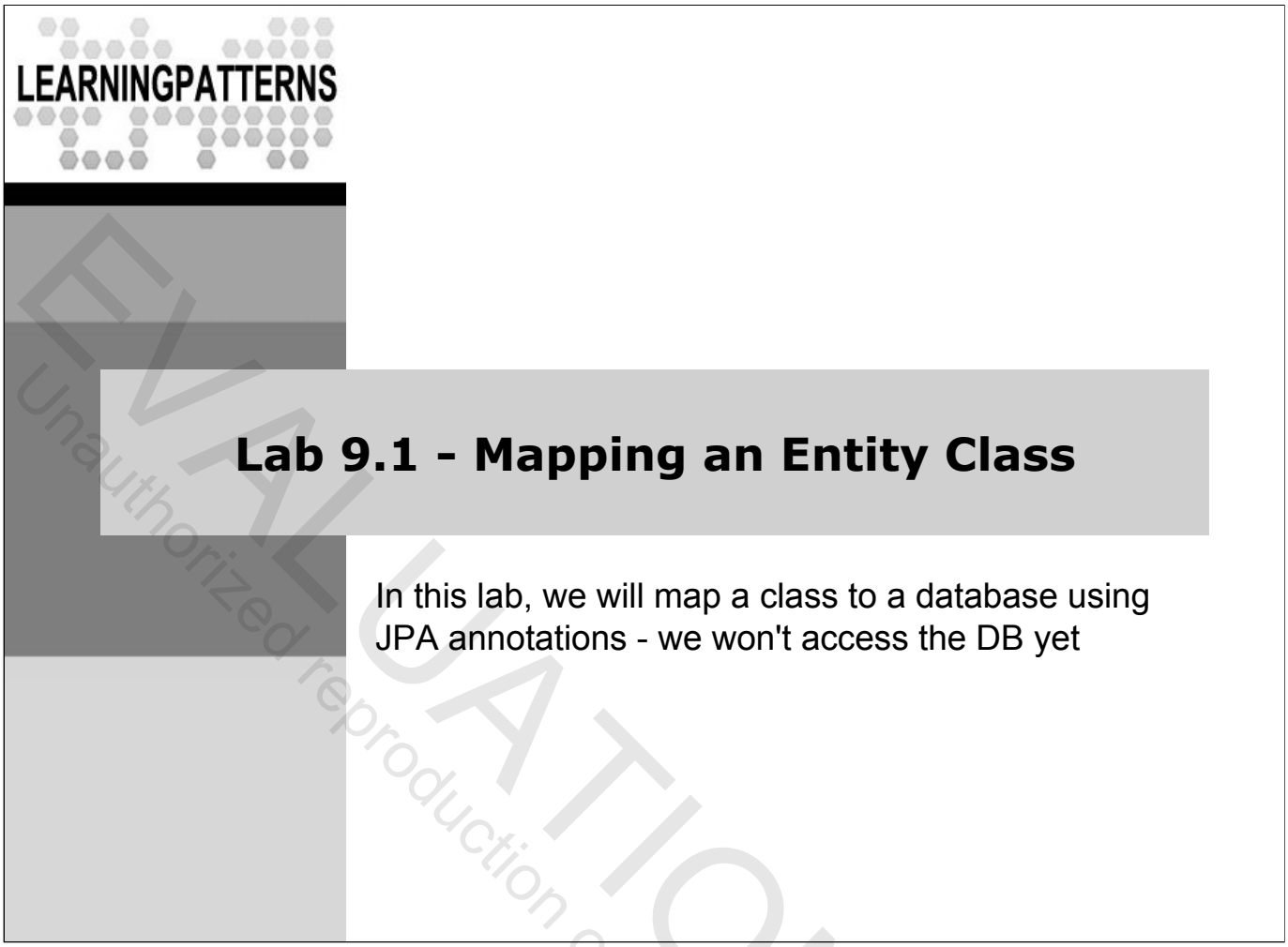
```
// package / imports not shown in most examples ...

@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 ...
}
```

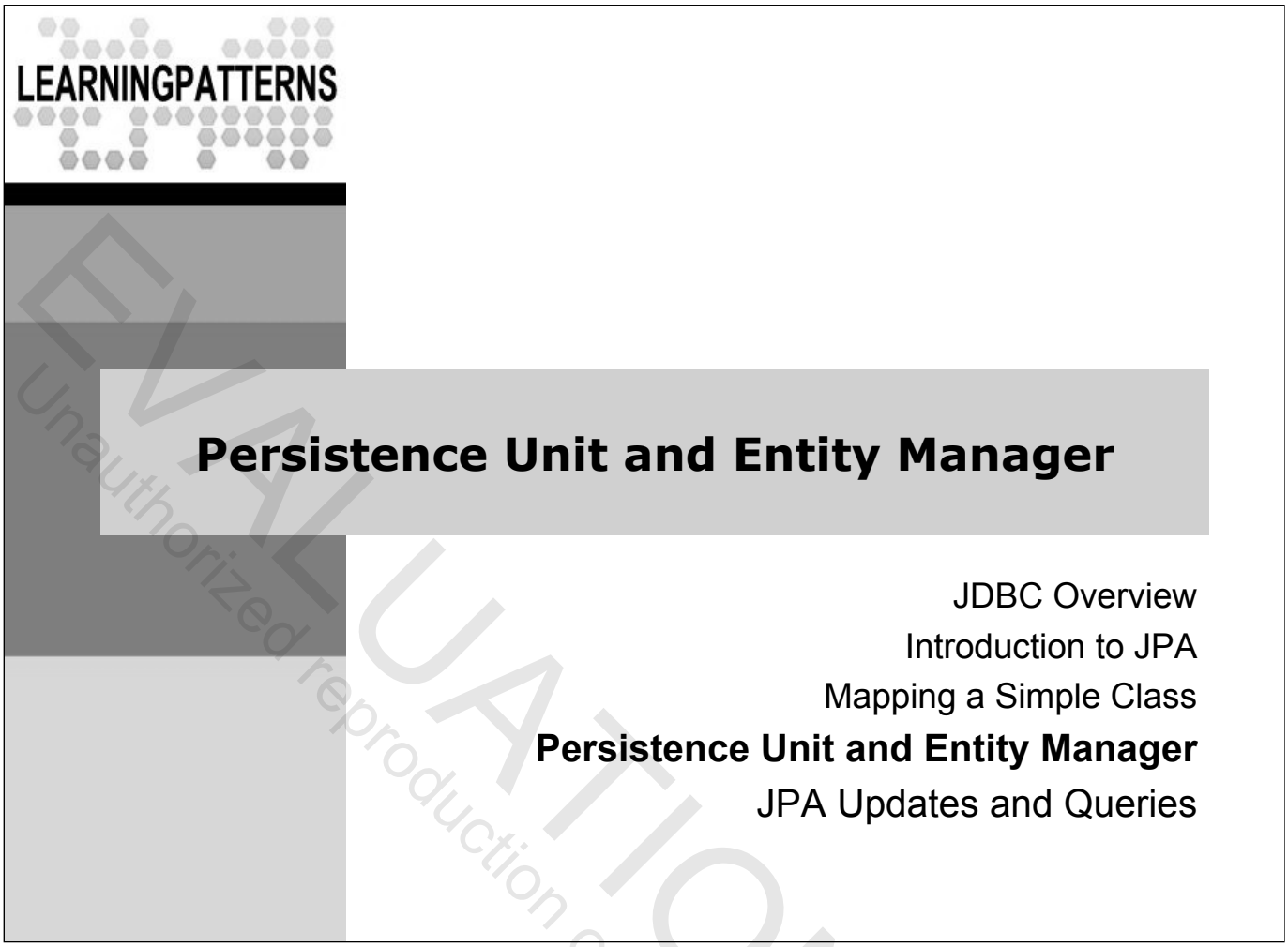
Basic Mapping Types

- JPA mapping supports many types automatically
  - Mapping them to their JDBC type in the DB
  - If the DB type is different, it tries to convert
  - Persistent properties may be:
    - 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

- 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





The Persistence Unit

- A **persistence unit** defines the set of entities an entity manager can manage (covered next)
  - Plus DB connection info, and other information
- Persistence units are named/configured in a configuration file
  - Usually called ***persistence.xml***, and shown on next slide
  - The central configuration file for JPA
  - Should be located under META-INF folder
  - Allows database configuration for your provider
- The classes included in a persistence unit are defined by:
  - Annotated entities (may be in jar files)
    - They are usually auto-detected by scanning
    - Can also be listed in *persistence.xml* (1)

- *persistence.xml* 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

- (1) You can list persistence classes in <class> elements in *persistence.xml*, e.g.

```
<class>com.javatunes.persist.MusicItem</class>
```

persistence.xml Structure

- **<persistence-unit>** defines the persistence unit
  - The **name** attribute specifies the persistence unit name (required)
  - 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 in the example (1)

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

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

- 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"

- (1) Different databases may use slightly different dialects of SQL, or have different capabilities in different areas
  - Usually JPA and other ORM implementations, like Hibernate, require you to configure the database you are using
  - Hibernate, as shown in the example, uses a hibernate.dialect property to do this
  - It takes a fully qualified class name as its value - which must be a class written to a specific Hibernate API (which is defined as in interface in the Hibernate API)

Accessing Entities

- Persistent entities in a persistent unit are defined by
  - Annotated classes in the root of a persistence unit or in jar files
  - XML mapping files or an explicit list of classes (1)
  - Often these are detected via scanning
- Interact with persistent entities via an **`EntityManager`** (EM)
  - It contains the API for persistence operations
  - e.g. creating, reading, writing, querying for entities
- The EM **manages** all entities that are obtained through it (1)
  - This set of entities is called a **persistence context**
  - Each EM has its own persistence context
  - There is only one instance with a given id in a persistence context

- (1) A persistence context is associated with an EM, and is comprised of the set of all managed entities within it
  - The EM manages these entities, for instance it makes sure that there is only one instance with a given persistent identity within its 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

- 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>
```

The EntityManager API

- Some important `EntityManger` methods include:
  - **`<T> T find(Class<T> entityClass, Object primaryKey)`**: Find an entity by its primary key
    - We'll look at the syntax of this later - it's easy to use it
  - **`void persist(Object entity)`**: Make a new entity instance managed and persistent
  - **`void refresh(Object entity)`**: Refresh instance state from db instance from the db, overwriting changes made to the entity
  - **`void remove(Object entity)`**: Remove the entity instance
- An **`EntityManagerFactory`** (EMF) produces the EM
  - Obtain an EMF via the `Persistence` class (for Java SE - shown later)
  - Using the persistence unit name in *persistence.xml*
  - In other environments (e.g. JEE) the EM is generally injected (1)

- (1) In environments that support Dependency Injection (DI), the EM is generally injected into classes that use it
  - For example, Spring or Java EE
  - These environments also support container managed transactions and TX scoped entity managers, which make JPA very easy to use
  - Coverage of these capabilities is beyond the scope of this course

EntityManager/EntityManagerFactory Example

- In Java SE, the **Persistence** class reads persistence.xml and creates an EMF
- Below we get an EMF for the `javatunes` persistence unit
  - Whose name is configured in *persistence.xml*
  - We get an EM from it
  - We close the EM and EMF when done to release all resources

```
// Get an entity manager factory
EntityManagerFactory emf =
            Persistence.createEntityManagerFactory("javatunes");
// Get an entity manager
EntityManager em = emf.createEntityManager();
try {
   // Do persistence related work here
 }
finally {
  em.close();  // Close the entity manager
  emf.close(); // Close the entity manager factory
}
```

Working with Transactions

- A **resource-local** entity manager generally used for Java SE
  - The app controls transactions via the entity manager API
  - Configured in the *persistence.xml* file:

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

- Use a **`javax.persistence.EntityTransaction`** object obtained from the EM to control transactions
  - In the example below, we use `EntityTransaction` as well as `EntityManager.find()` to retrieve an item by id
    - Note that we pass `Event.class` as an argument to find
    - Indicates that the generic `find` method returns an instance of `Event`

```
EntityManager em = // Initialization as shown previously
em.getTransaction().begin();  // Begin a transaction
Event m = em.find(Event.class, new Long(1));
em.getTransaction().commit();  // Commit a transaction
```

Complete JPA Example

- We get an EMF and then an EM
  - We use the EM to control transactions, and get an `Event`
  - We close the EMF and EM after we use them

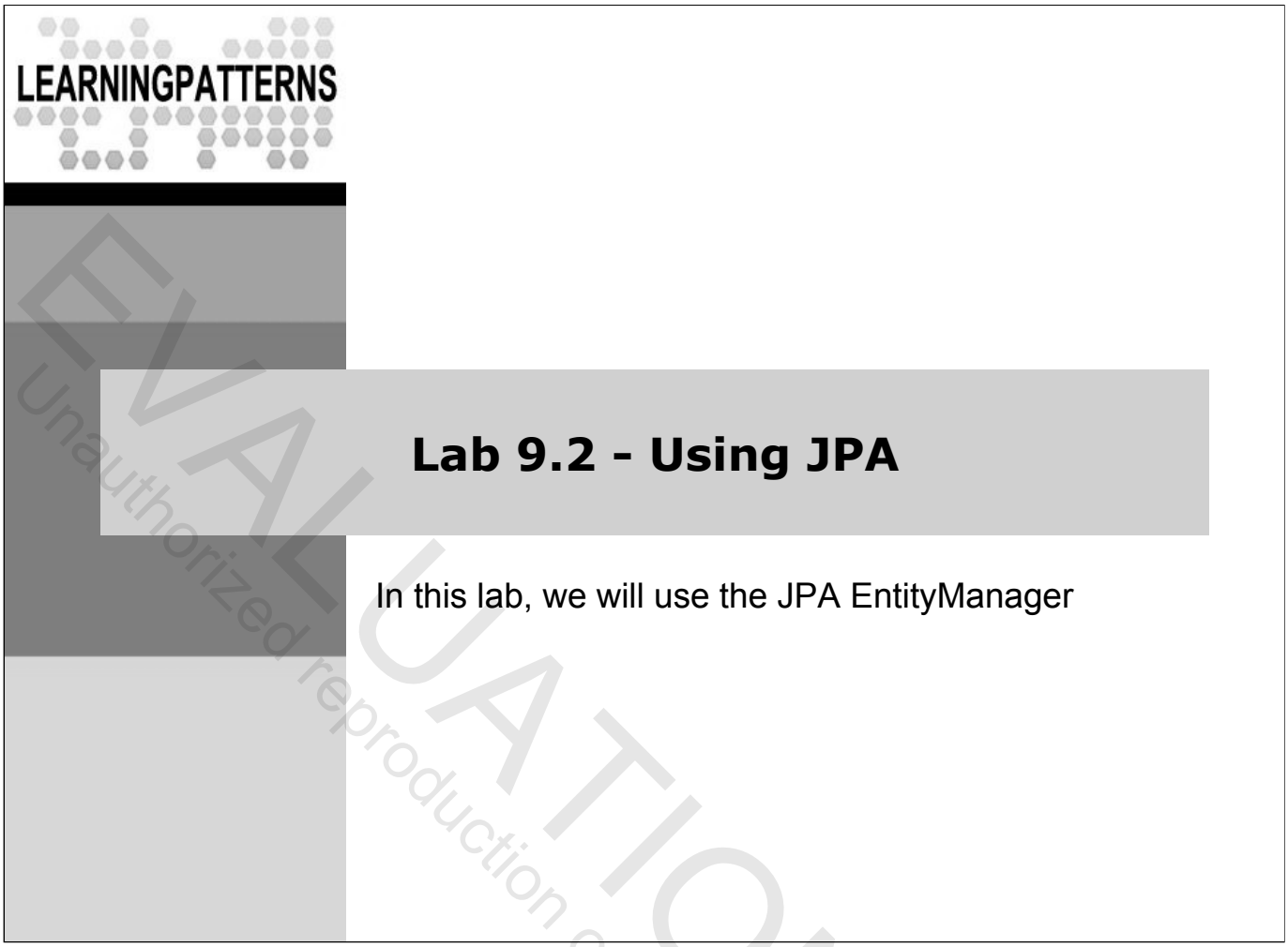
```
// Get an entity manager factory
EntityManagerFactory emf =
            Persistence.createEntityManagerFactory("javatunes");
// Get an entity manager
EntityManager em = emf.createEntityManager();
try {
  em.getTransaction().begin();  // Begin a transaction
  Event m = em.find(Event.class, new Long(1));
  em.getTransaction().commit();  // Commit a transaction
}
finally {
  em.close();  // Close the entity manager
  emf.close(); // Close the entity manager factory
}
```

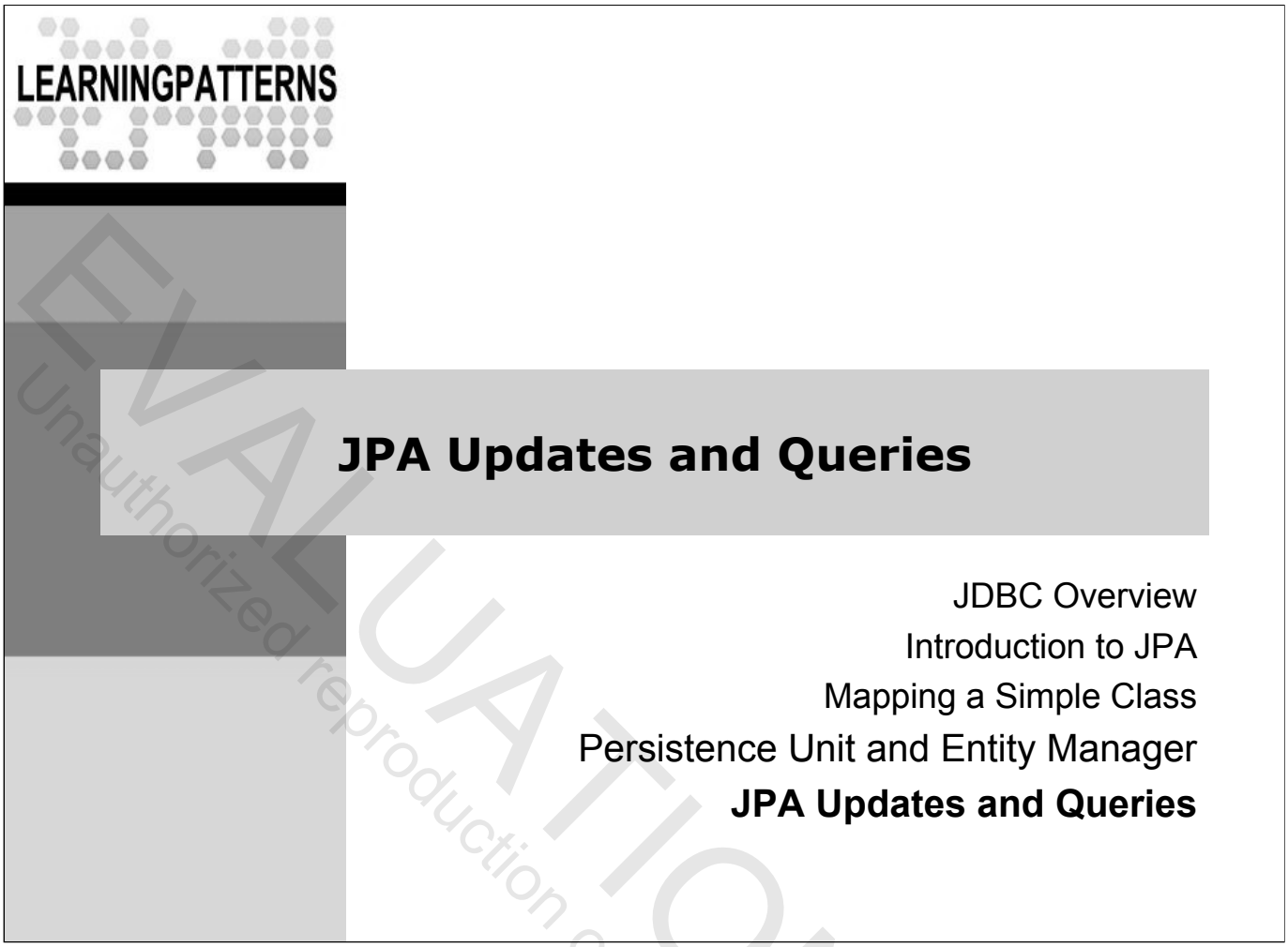
Summary

- Standard steps for using JPA
  - Get an **`EntityManagerFactory`**
    - With Java SE, we use the `Persistence` class for this
  - Get an **`EntityManager`** from the `EntityManagerFactory`
    - Note: In a Java EE environment, these are both done differently (1)
  - **Start a TX** using the `EntityManager`
  - **Do your work**, e.g. do a `find()`, and any other work needed
  - **Finalize the TX**, and **close all resources**
- The setup is complicated - is it worth it ?
  - It also uses sophisticated Java language capabilities (1)
  - Once it's set up, it saves a huge amount of work over plain JDBC/SQL
  - In general, it is well worth it

- JPA takes advantage of several more advanced techniques that you may just be learning
  - Interfaces, exceptions, delegation, etc
  - This is a good example of why the Java language has these capabilities

- In a Java EE environment an EM is usually injected
  - Meaning that you don't have to write code to access one
  - You still need to configure your persistence unit in *persistence.xml*, but the access of the `EntityManager` (or an `EntityManagerFactory`) is done transparently for you by the container
  - Details are beyond the scope of this course





Persisting a New Entity

- Very easy to insert new instances (rows) into the DB
  - Just create a new (transient) instance using *new*, and set its values
    - Generally the id is a synthetic key and not set (1)
  - Save the instance to the DB via an entity manager
- `EntityManager.`**`persist()`** persists an instance

  `void persist(Object entity)`
- The instance is inserted into the DB
  - When the TX commits it is stored in the DB (and often assigned an id)

```
try { // Entity manager (em) initialization not shown
  em.getTransaction().begin();  // Begin a transaction
  Event newEvent = new Event();
  newEvent.setTitle("A party");
  em.persist(newEvent);
  em.getTransaction().commit();  // Commit a transaction
}
```

- (1) Generally, in modern databases, the id property is a synthetic key that is generated in the database
  - It generally has no business meaning, and is used solely to uniquely identify a row in a database
  - Accordingly, you do not set an id when persisting an object representing a new row
  - Rather, it will be generated by the database

- The semantics of the persist operation, applied to an entity *X* are as follows:
  - If X is a **new entity**, it becomes managed
  - The entity X will be entered into the database at or before transaction commit or as a result of the flush operation.
  - If X is a **preexisting managed entity**, it is ignored by the persist operation. However, the persist operation is cascaded to entities referenced by X, if the relationships from X to these other entities is cascade=PERSIST or cascade=ALL (covered later)
  - If X is a **removed entity**, it becomes managed

Updating a Persistent Instance

- If you have a persistent instance (one currently associated with a persistence context) you can just update that instance
  - The changes will be persisted when the TX commits
  - Remember, for a managed instance, JPA detects any changes and **synchronizes the state with the database** when the TX completes

```
// Assume the code fragment occurs in a transaction context and
// you have an initialized EntityManager reference (em)

Long parthEventId = new Long (5); // Assume this is the id we want
Event partyEvent = em.find(Event.class,partyEventId);

// Change will be automatically persisted
partyEvent.setTitle("A GREAT party");

// When Tx commits, the changes are persisted to database
```

Removing an Instance

- It's also very easy to delete an instance from the database
  - The instance must be in the entity managers persistence context
  - You can then call **`remove()`** on the instance
  - When the context is synchronized with the database, the row will be deleted
  - Note that very often rows are not deleted in production systems
  - It's more common to keep old data around because it may be needed for historical queries

```
// Assume a transaction, and EntityManager reference, as before

Event partyEvent = em.find(Event.class,new Long(5));

// Remove the event
em.remove(partyEvent);

// When Tx commits, the deletion is persisted to database
```

- The in-memory object becomes a transient instance again, with no representation in the database, and not in the scope of any persistence context
  - We'll talk about the lifecycle of persistent objects more later in a future session

Java Persistence Query Language (JPQL)

- **Java Persistence Query Language** (**JPQL**) is an OO query language that is part of JPA
  - Similar to SQL in syntax and structure
  - Leverages knowledge about SQL
  - If you don't know the identifiers of the objects you are looking for, you need a query since you can't use `find()` on the id
- Queries **object graphs**, not relational tables
  - Fully object-oriented
  - Understands associations between objects
  - Supports inheritance and polymorphism
- Structure is similar to SQL
  - `SELECT`, `FROM` and `WHERE` clauses
  - Can use lower or upper case for keywords

- The JPA spec says this about JPQL
  - The Java Persistence query language is a query specification language for string-based dynamic queries and static queries expressed through metadata. It is used to define queries over the persistent entities defined by this specification and their persistent state and relationships.
  - The Java Persistence query language can be compiled to a target language, such as SQL, of a database or other persistent store. This allows the execution of queries to be shifted to the native language facilities provided by the database, instead of requiring queries to be executed on the runtime representation of the entity state. As a result, query methods can be optimizable as well as portable.
  - The query language uses the abstract persistence schema of entities, including their embedded objects and relationships, for its data model, and it defines operators and expressions based on this data model. It uses a SQL-like syntax to select objects or values based on abstract schema types and relationships. It is possible to parse and validate queries before entities are deployed.
  - *The term abstract persistence schema refers to the persistent schema abstraction (persistent entities, their state, and their relationships) over which Java Persistence queries operate. Queries over this persistent schema abstraction are translated into queries that are executed over the database schema to which entities are mapped.*
  - Queries may be defined in metadata annotations or the XML descriptor.

JPQL Basics – SELECT Statement

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

```
SELECT e FROM Event e
```

  - This query will return all the Event instances in the database
  - It is an **object based** query **selecting from an entity**, not a table
  - Notice also that only the Event alias appears in the SELECT clause, because the result type of the select is the Event entity
  - The result of this query is a **collection of Event entities**
- **Path expressions** navigate to a property via dot notation
  - For example, the following returns a collection of the event dates:

```
SELECT e.date FROM Event e
```

- Structure is similar to SQL

- JPQL is similar to SQL so it can leverage the knowledge and tools that are available for SQL
- The generated SQL will be something like

```
SELECT e.EVENT_ID, e.EVENT_DATE, e.TITLE FROM EVENTS e
```

- The JPA spec defines a SELECT statement as a string which consists of the following clauses:
  - A `SELECT` clause, which determines the type of the objects or values to be selected.
  - A `FROM` clause, which provides declarations that designate the domain to which the expressions specified in the other clauses of the query apply.
  - An optional `WHERE` clause, which may be used to restrict the results that are returned by the query.
  - An optional `GROUP BY` clause, which allows query results to be aggregated in terms of groups.
  - An optional `HAVING` clause, which allows filtering over aggregated groups.
  - An optional `ORDER BY` clause, which may be used to order the results that are returned by the query.

Executing a Query

- JPA 2 provides the **TypedQuery** interface for executing queries
  - Accessed from the entity manager via this method (see notes):

```
<T> TypedQuery<T> createQuery(java.lang.String qlString,
                              java.lang.Class<T> resultClass)
```

  - It is an OO representation of a query – including methods to:
    Set query parameters, execute a query, execute an update, and set paging parameters on a query
  - When created with this method, these are called **dynamic queries**
- Once you have a `Query` instance you can execute it to retrieve instances with the following methods
  - **`List<X> getResultList()`**: Return list with query results
  - **`<X> getSingleResult()`** : Single, typed, result

- The syntax of the generic createQuery method may look strange if you haven't used Java generics before with methods - let's break it down
  - The method signature is

```
<T> TypedQuery<T> createQuery(java.lang.String qlString,
  java.lang.Class<T> resultClass)
```

  - The first <T> in the return type simply indicates that this is a generic method, parameterized by the type parameter <T>
  - The `TypedQuery<T>` return value indicates that the return type is generic (that is, it will take on different types based on the <T> parameter)
  - The `java.lang.Class<T>` argument indicates that when you call the method, you pass in the class which specifies what type <T> actually is in that call
- The documentation for this method states:
  - Create an instance of `TypedQuery` for executing a Java Persistence query language statement. The select list of the query must contain only a single item, which must be assignable to the type specified by the `resultClass` argument

Example Query Execution

- The example at bottom uses `Query.getResultList()`
  - `getResultList()` executes the query, retrieving all the entities into memory at once, and returns the result as a `java.util.List`
- There are other useful methods on TypedQuery – e.g.
  - **`TypedQuery<X>`** `setMaxResults(int maxResults):`
    - Sets maximum number of rows to retrieve
  - **`TypedQuery<X>`** `setFirstResult(int startPosition)`
    - Sets position of first row to retrieve
- Many other methods - see the documentation

```
TypedQuery<Event> q = em.createQuery("SELECT e FROM Event e",
                                     Event.class);
List<Event> resultList = q.getResultList();
for (Event curEvent : resultList) {
  System.out.println("Event: " + curEvent.getId());
}
```

Where Clause

- Of course, we can also provide selection criteria for a query in a **where** clause
  - You use path expressions that navigate to entity properties and fairly standard expressions to create the criteria

```
SELECT e FROM Event e WHERE e.id > 10
```

  - This query does what you expect it to do
  - It returns all event instances with an id greater than 10
- Notice that we can access properties of an entity in a query
  - JPQL use a familiar dot notation to access properties
  - In the query above, `e.id` refers to the id property of the returned events
  - We are working in terms of entities – not DB rows/columns

JPQL Capabilities

- JPQL supports the same basic operators as SQL
  - Unary positive and negative: +, -
  - Arithmetic (+ - …), Binary (= < > IS EMPTY …), and more
  - It also supports familiar literals (e.g. `'Jane Doe'`), numbers, etc
- You specify query parameters as shown below, using **named parameters**

  `SELECT e FROM Event e WHERE e.id > **:id**`
  - You then populate them using set methods as shown at bottom

```
// Named parameter example - find events by title
TypedQuery<Event> q = em.createQuery(
    "SELECT e FROM Event e WHERE e.title = :title", Event.class);
q.setParameter("title", "Party");
List<Event> l = q.getResultList();
```

- Support for the use of hexadecimal and octal numeric literals is not required by the JPA specification

Data Access Objects

- It's best to encapsulate JPA/database code
  - There is a lot of detailed code
- A **DAO** (**Data Access Object**) is one encapsulation approach
  - Gathers all persistence functionality into one place
  - Easier to write and manage
  - Hides persistence details from external users, and insulates them from the implementation
  - Writing a DAO can still be complex because of the many details involved

- Each different use of the database may require mapping of result sets, handling of exceptions, etc.
  - That's a lot of code to write and keep track of
- The DAO pattern is very valuable because it insulates the rest of the application from the underlying database access technology
  - This makes the application simpler
  - It also allows you to swap in different database access technologies transparently to the rest of the application
  - For example, you can start out with simple JDBC, move to Hibernate, then evolve to the Java Persistence API without affecting the rest of your code

Simple DAO

- Below, we give a simple example of a DAO for events
  - Initialized with an `EntityManager` upon creation
  - Uses the `EntityManager` for persistence operations

```
// imports and other details not shown
public class EventDAO {
  private EntityManager em;

  public EventDAO(EntityManager em) { this.em = em; }

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

  public Event update(Event event) { return em.merge(event); }

  public Event find(Long id) {
   return em.find(Event.class, id);
  }
}  // Other possible methods not shown
```

Using the DAO

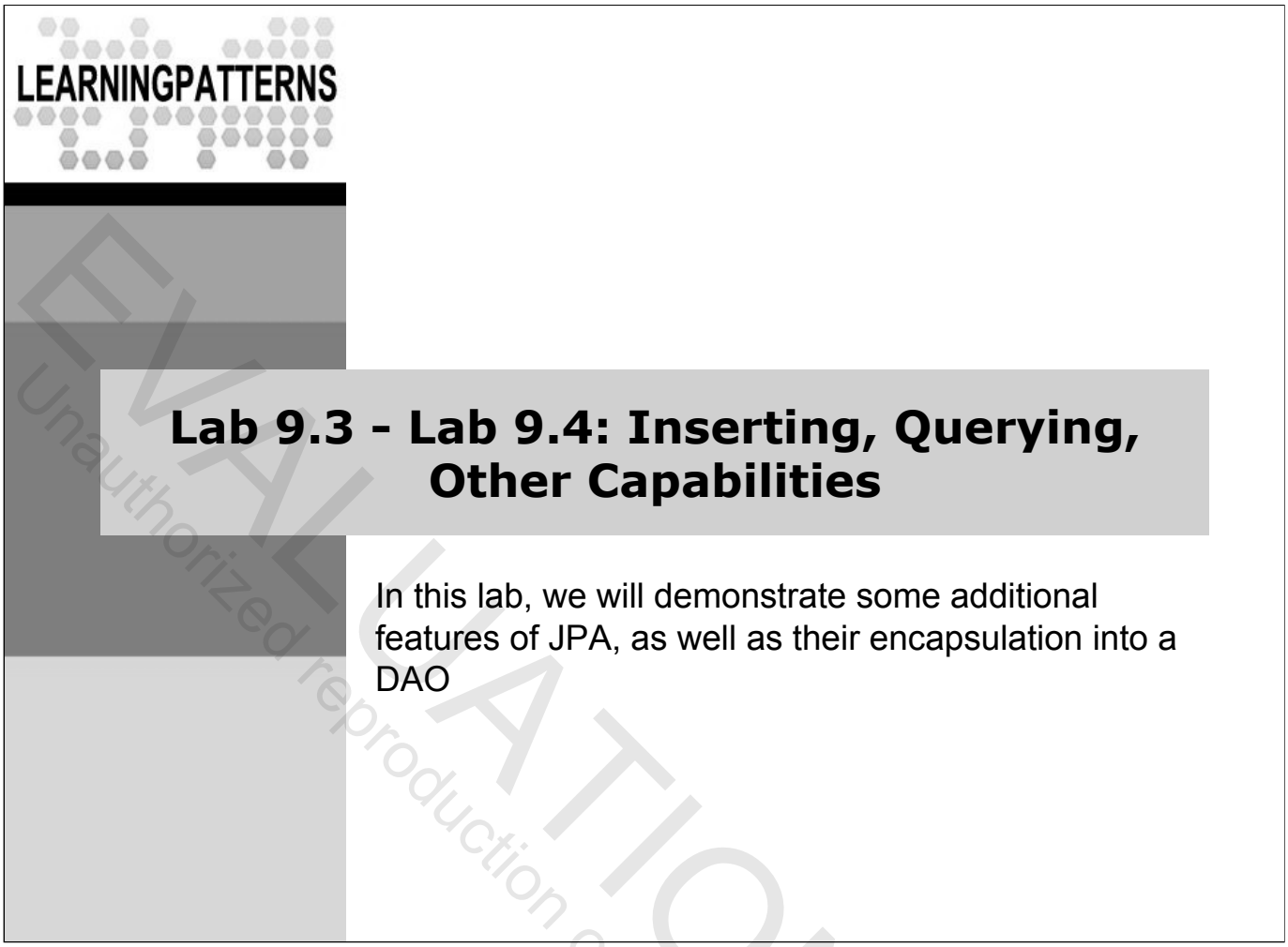
- It's easy to use this simple DAO, as shown below
  - You're still managing the `EntityManagerFactory` and `EntityManager` directly in your code
  - Other technologies (e.g. Spring and JEE) make this easier (1)
  - In the lab, we'll use a utility class for this

```
// Code fragment using the DAO, and JPA directly
// 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
  EventDAO dao = new EventDAO(em);
  Event e = dao.find(new Long(1));
  em.getTransaction().commit();  // Commit a transaction
}
finally { em.close(); }
```

- In our simple example, it is fairly straightforward to pass in the EntityManager to the DAO
  - However, consider a use case where a number of complex interactions take place, all of which need to work with persistent entities
  - In that case, you would likely need to pass the EntityManager around to all the DAO objects that are used
  - This can get tedious and tiresome to code

- Another issue with this is that the complex and repetitive code to create the EntityManager is exposed in your code

- (1) More robust technologies such as Java EE and Spring make it much easier to acquire the `EntityManager`
  - Generally, these are injected by the Spring/JEE container into the objects that need them
  - Configuration and initialization of the EM are taken care of by the container based on configuration information that you supply



Session Review

- What is JDBC?
- What does the Java Persistence API do, and why do we need it?
- What is an entity?
- How do you map an entity to the database?
- What is the id property of an entity?
- What does the entity manager do?
- How do you persist a new instance of an entity class?
- How are queries defined with Java Persistence?