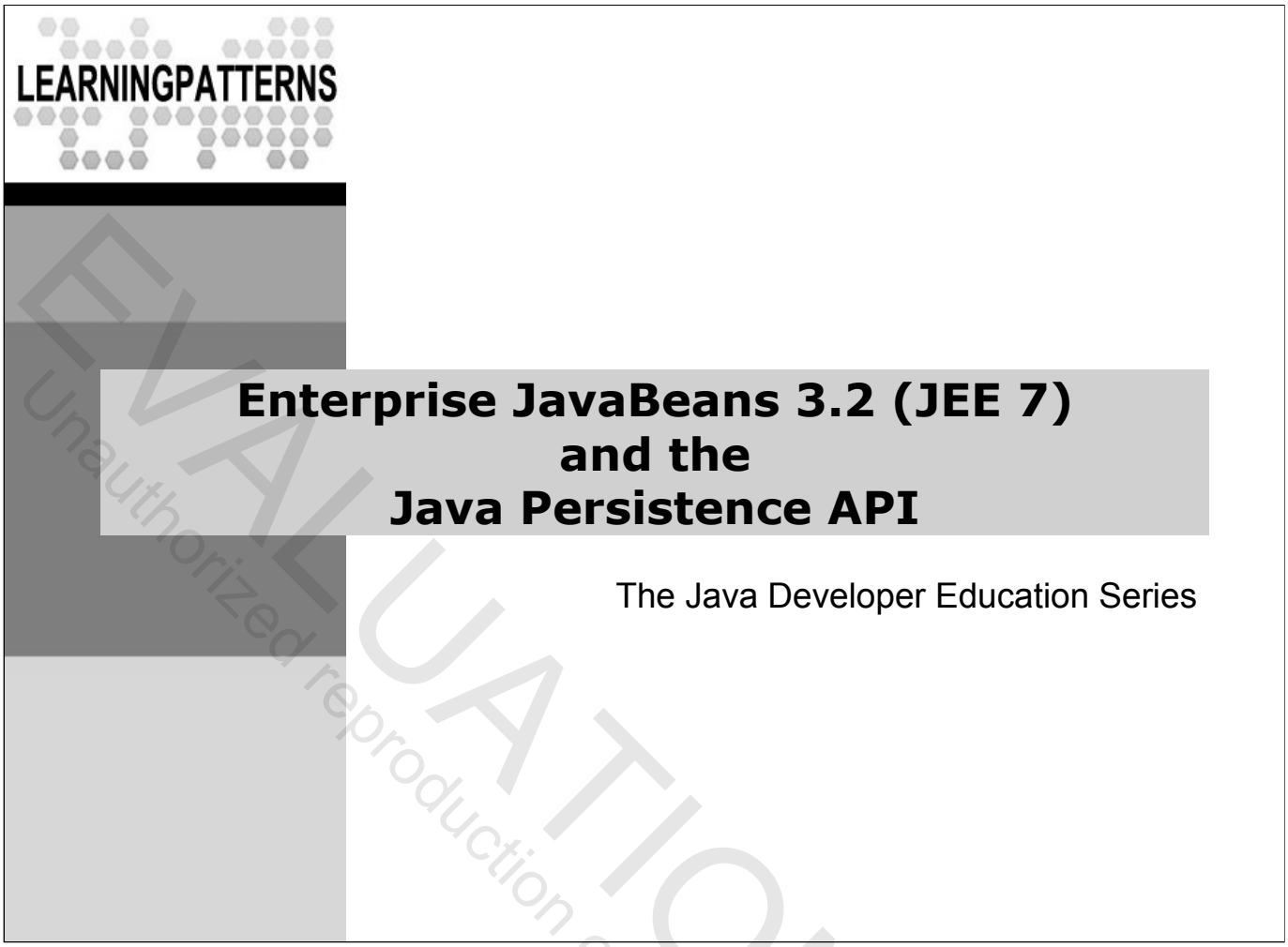
Table of Contents – EJB 3.2 and JPA 2



Notes:

- Java, EJB, Enterprise JavaBeans and all Java-based trademarks are registered trademarks of Oracle, Inc

Workshop Overview

- This course provides a thorough introduction to Enterprise JavaBeans V3.2 (JEE 7), including
  - The needs EJB is designed to address
  - The basic concepts and architecture
  - Thorough coverage of the EJB API and details on its use
  - Thorough coverage of the Java Persistence API V2 (JPA 2)
  - Design principles for correct usage
- The workshop consists of at least 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:

Workshop Objectives

- At completion you should be able to
  - Understand how EJB relates to the rest of Java/Java EE
  - Understand EJB concepts and architecture
  - Be familiar with the EJB API, including the Java Persistence API (JPA)
  - Be able to write and use EJBs
  - Be familiar with the JPA API, and be able to write and use persistent entities, including advanced capabilities like relationships and inheritance
  - Understand the tradeoffs involving EJB
  - Understand important design principles for EJB

Notes:

Workshop Agenda

- Session 1: **Introduction to EJB**
- Session 2: **Session Bean Architecture and API**
- Session 3: **Additional EJB Capabilities**
- Session 4: **Message-Driven Bean Architecture and API**
- Session 5:**Transactions and Security**
- Session 6: **Exceptions and Best Practices**
- Session 7: **Java Persistence API 2 (Entity Beans) Intro**
- Session 8: **Java Persistence API Inserts and Queries**
- Session 9: **Java Persistence API Associations**
- Session 10: **Java Persistence API Additional Capabilities**

Notes:

Course Prerequisites

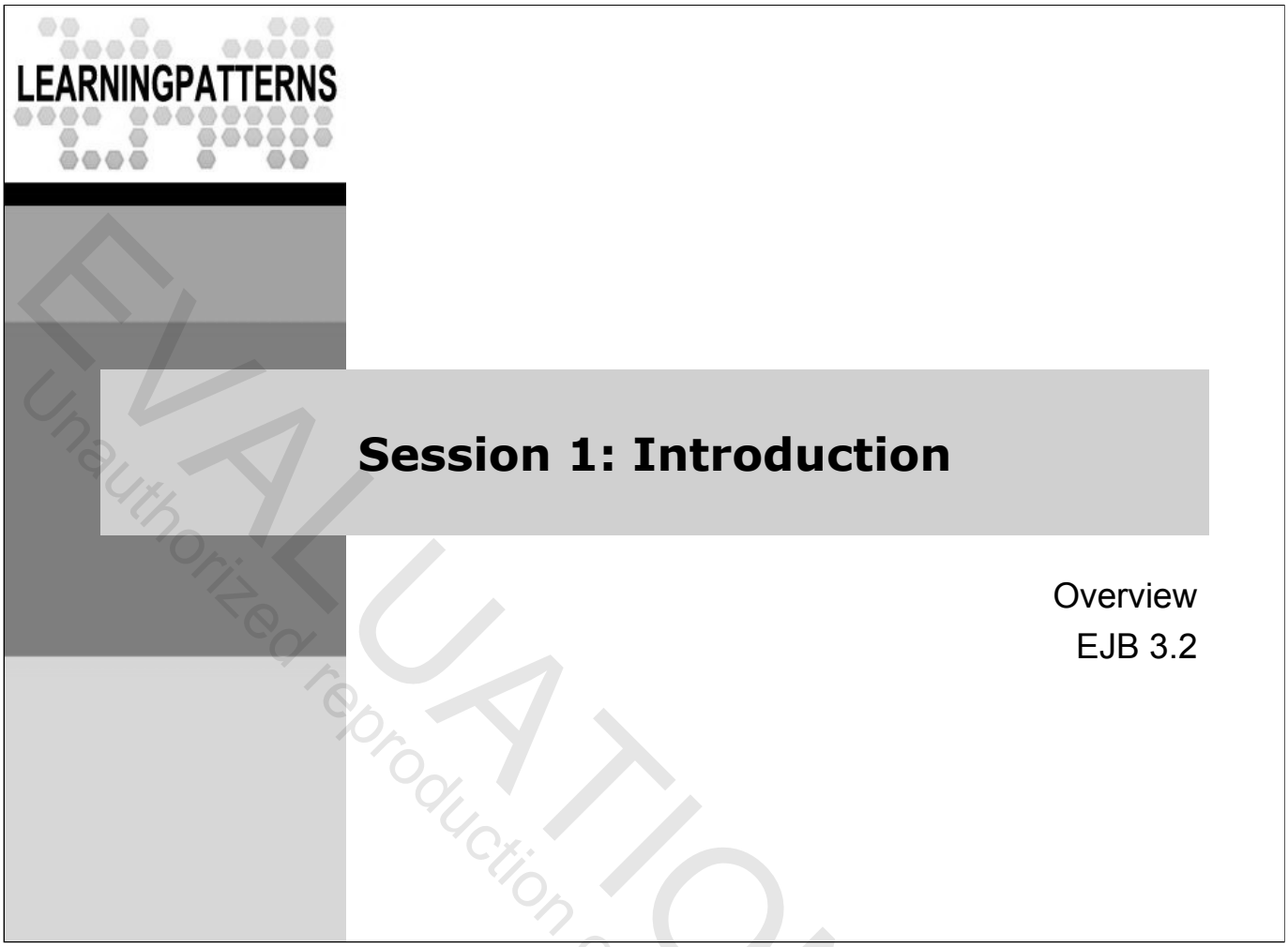
- Proficiency in Java and Object-Oriented programming
- General knowledge of Java EE (Enterprise Edition)
- Knowledge of relational databases

Notes:

Labs

- The workshop has numerous hands-on lab exercises, structured as a series of brief labs
  - Many follow a common fictional case study called **JavaTunes**
    - An online music store
  - There is a placeholder slide for each lab in the manual
  - The detailed lab instructions **are at the end of the 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
- The end of a lab is marked with a stop like this one:

Notes:

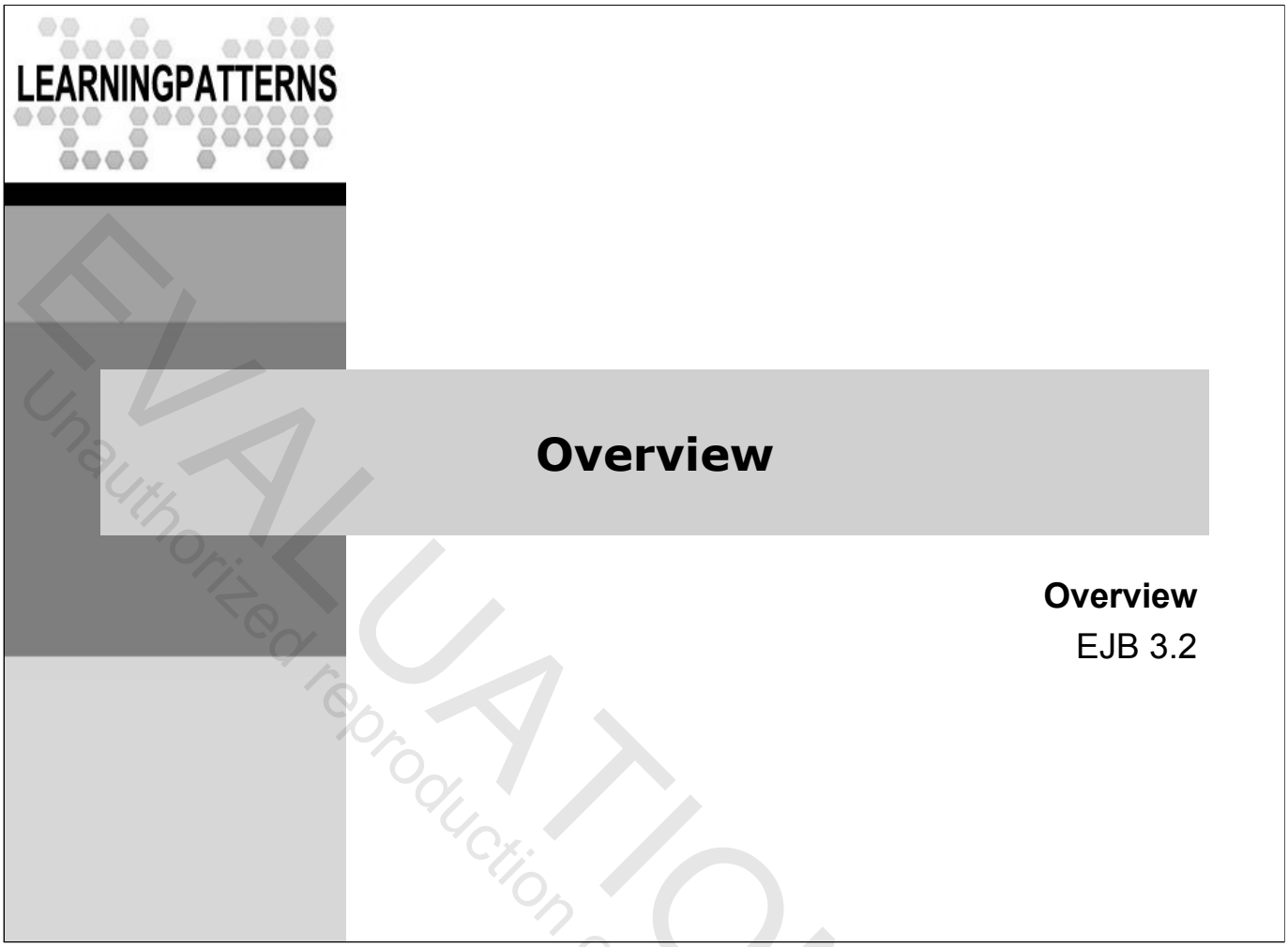


Notes:

Lesson Objectives

- Gain a high level understanding of EJB and EJB architecture
- Understand how EJB fits into the Java EE architecture
- Understand how EJB relates to other technologies
- Become acquainted with EJB 3.2, its goals, and the problems with earlier versions

Notes:



Notes:

What is EJB

- **EJB** is a framework for creating **server-side components** that are:
  - Transactional, Distributed, Portable, Reliable, Secure, Scalable
  - It simplifies the building of multi-tier distributed object applications
  - EJB is a technology to create business-tier components for these kinds of applications
- EJB provides a server-side framework for providing a core set of system services to Java components
  - Services such as low-level transaction and state management, multi-threading, and connection pooling

Notes:

EJB Goals

- Provide a **standard distributed component architecture** for Java applications
  - Allowing easy creation of distributed business applications
  - Portable across many vendors (write once, run anywhere)
  - Fitting into the Java EE (Enterprise Edition) architecture
  - Enabling the use of third-party development tools
  - Generally meant for creating **business tier** components
- **Relieve developers** from managing transactions, threads, security, resource management, while still providing access to low-level APIs
  - These kinds of issues are generally taken care of by the EJB framework

Notes:

EJB Goals (continued)

- Provide a persistence framework to simplify Object-Relational Mapping (ORM)
  - The issue of mapping a set of (Java) objects to information in a relational database is complex
  - The data is in different forms
  - Going from one form to the other is difficult, and writing the code is tedious
- Persistent entities provide a framework to **automate** the mapping of Java objects to relational data
  - A mapping is defined via metadata, and the framework generates the JDBC code to work with the data

Notes:

Types of Enterprise JavaBeans

- **Session Beans** provide a **business service**
  - Distributed, transactional
  - Bean instances live in a software environment called the **EJB container**
  - The container manages the lifecycle of instances, as well as distributed access, transactions, etc.
- **Message Driven Beans** (**MDB**) integrate EJB with **messaging** (JMS) systems
  - An MDB is an asynchronous message consumer
  - It consumes messages from a queue or topic
  - Makes asynchronous processing of incoming messages on the server simpler
  - Allows for concurrent processing of a stream of messages by means of container managed pooling

Notes:

Java Persistence API

- The brand new **Java Persistence API** defines a Java persistence framework
- **Persistent Entities** provide **Object-Relational Mapping** (ORM) capabilities
  - Persistent entities are ***lightweight persistent domain objects***
  - Primary concern is mapping objects to relational data
  - Persistent entities are not distributed objects, though they may be accessed in a distributed way using a session bean façade
- Persistent entities are not really "Entity Beans"
  - They are a separate part of the specification now
  - Can be used separately from other parts of EJB
  - In a new package, `javax.persistence`, not in `javax.ejb`

Notes:

- The Java Persistence API is a separate part of the same specification defining EJB (JSR-220)
  - It is required for all EJB containers
  - However, it can also be used separately if only Java Persistence is needed

EJB and Java EE (Enterprise Edition)

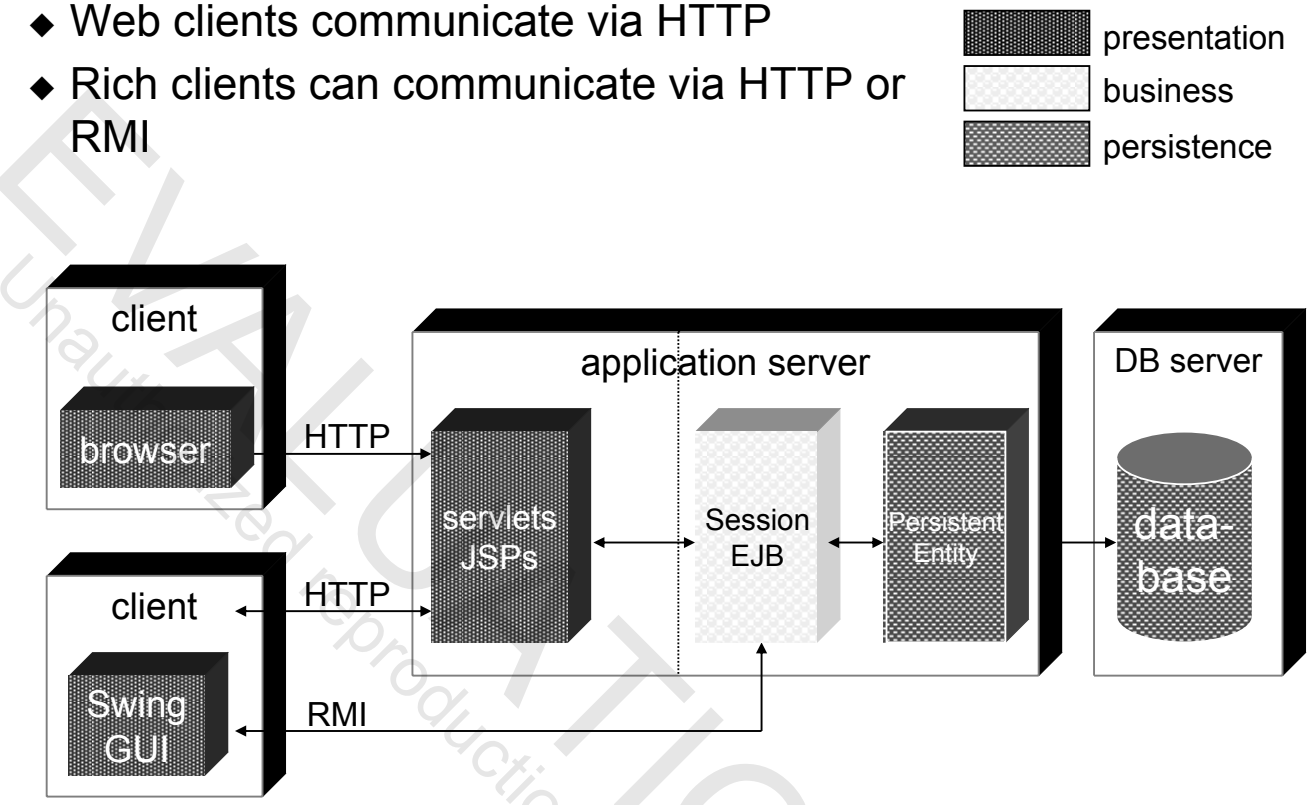
- Java EE is an architecture for building multi-tier enterprise applications
  - Umbrella for many other Java technologies including:
  - EJB, Servlets/JSP, JMS, RMI …
- EJB serves as the distributed component technology and persistence framework for Java EE
  - Generally, EJB is used on the server side
  - It is often invoked from the Web tier, but may be invoked by thick clients (e.g. Swing clients)
  - It can also be used in Service Oriented Architectures (SOA)

Notes:

- Java EE was previously known as J2EE
  - With the release of Java 5 / Java EE 5, the 2 was dropped from the names

EJB in Java EE Architecture

- Web clients communicate via HTTP
- Rich clients can communicate via HTTP or RMI



Notes:

- This architecture may be attractive because you can support both Web browser clients and Swing clients, and do so in several different ways.
  - Web browser clients interact with the EJB business tier indirectly, via the servlet/JSP presentation tier.
  - Swing clients can generate HTTP requests to the servlet/JSP presentation tier or can interact directly with the EJB business tier. You might want to do this to reuse an existing servlet/JSP interface or to use HTTP to get through a firewall.

- We will talk about MDB later

SOA and EJB

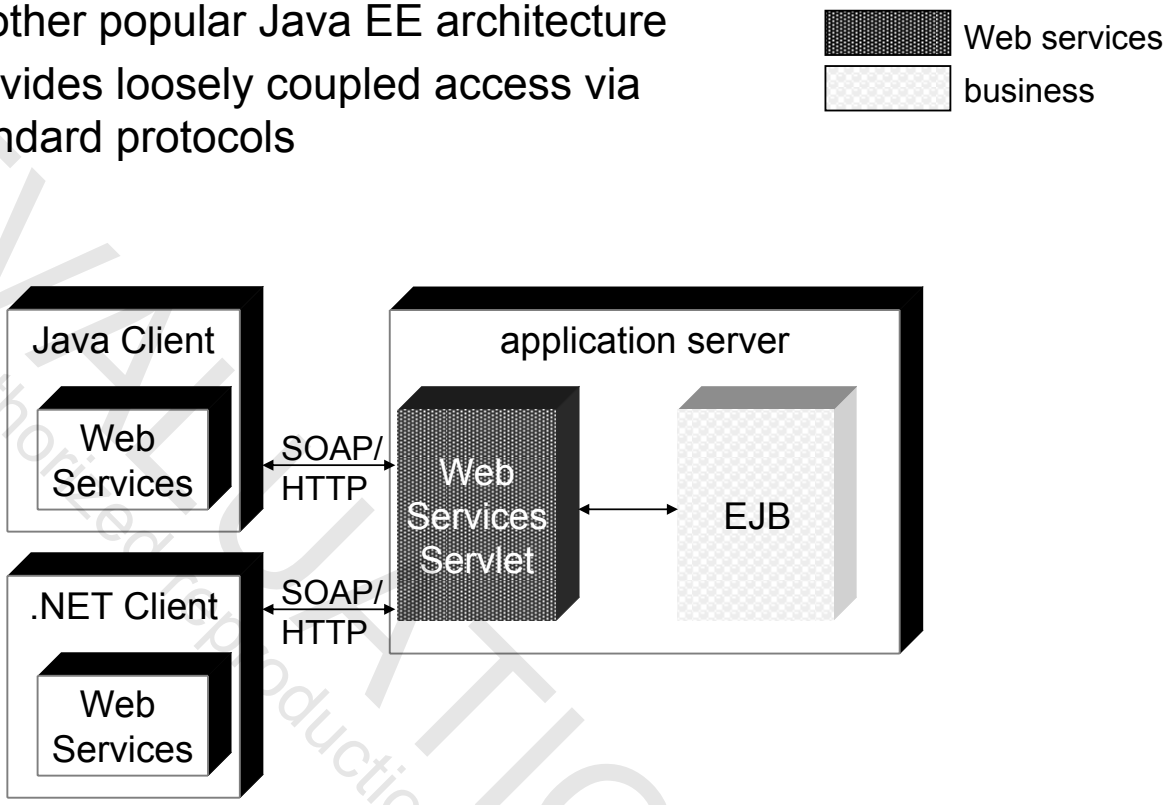
- SOA (Service Oriented Architecture) is an important basis for enterprise architectures
- Based on providing resources on a network
  - As independent services independent of their implementation
  - Results in loosely coupled architecture
- EJB can support SOA in multiple ways
  - The services can be exposed directly as a session EJB
    - SOA is usually thought of in terms of Web Services (e.g. SOAP), but EJB may be a useful alternative in some cases
    - This usually requires Java clients
  - A Web Service can be implemented using EJB
    - Web Services are just a façade for exposing a service
    - The service has to be implemented in some way – either as an EJB or as a regular Java object

Notes:

- Choosing to use EJB in a service oriented architecture is entirely valid
  - Web Services add a layer of complexity and inefficiency
  - If you don't need the advantages of Web Services, then don't use them
- In fact, you can even invoke EJB from non-Java clients
  - This uses CORBA
  - It's not really a practical architecture unless you're already using a CORBA architecture

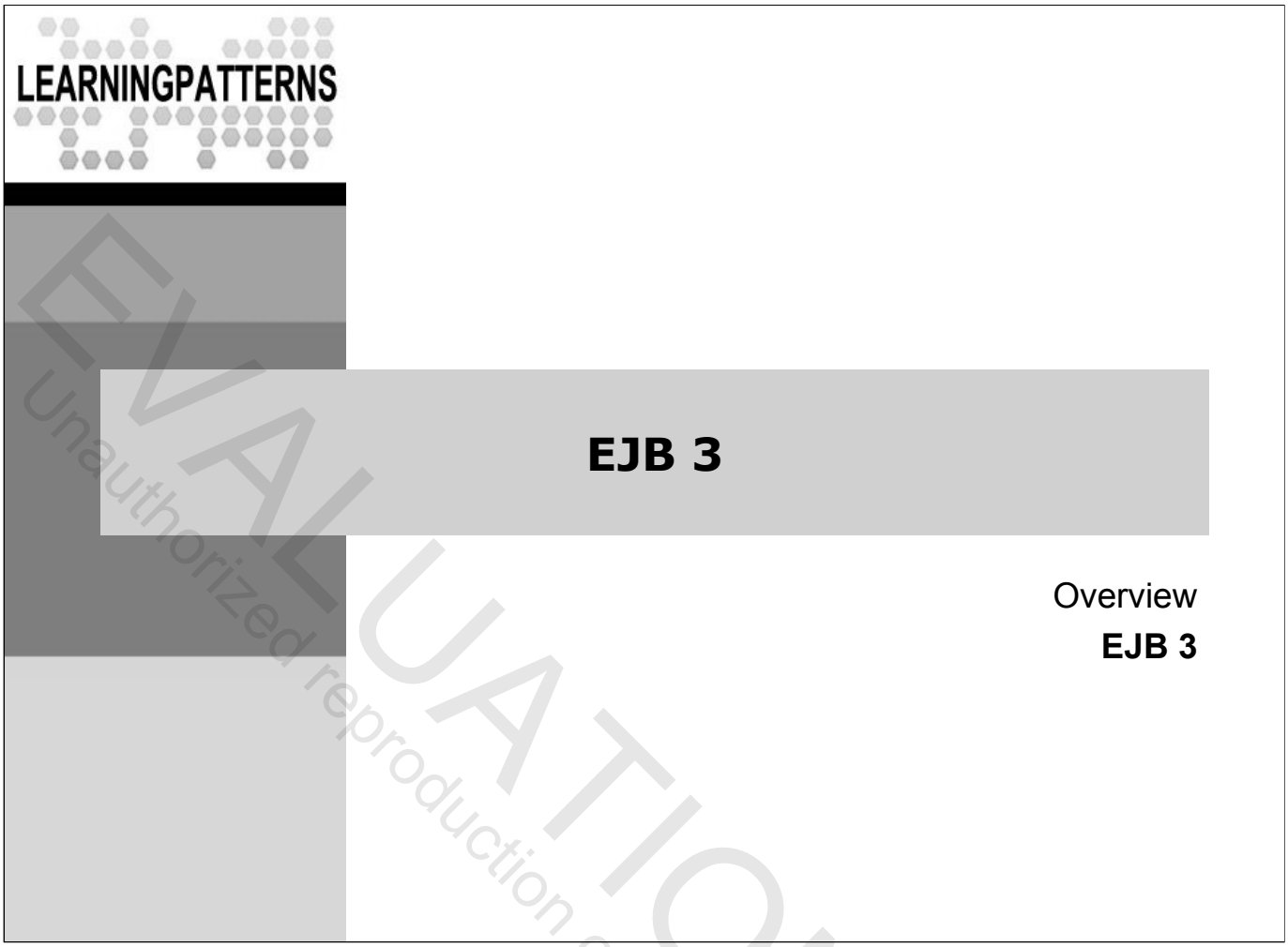
SOA with Web Services and EJB

- Another popular Java EE architecture
- Provides loosely coupled access via standard protocols



Notes:

- In this type of architecture, SOAP/HTTP is used as the communication protocol, and the service is implemented using EJB
- Servlets are used purely to support SOAP over HTTP, and are not really involved in presentation layer aspects



Notes:

EJB 3 Overview

- Complete reworking of EJB specification
  - Major differences from EJB 2.x
  - Some areas have changed completely
- Uses Java annotations heavily
  - Reduces the use of XML configuration files (Deployment Descriptor), and can eliminate them
- Persistence is completely different from earlier releases
  - Total overhaul of EJB persistence
  - Persistent entities much lighter weight
  - Based on mature ORM technologies such as Hibernate & Toplink
  - Persistent entities can't be invoked remotely
  - Persistence can be used separately from other parts of EJB

Notes:

EJB 2.x Problems

- Cumbersome and difficult to program
  - Each EJB required (at a minimum) an implementation class, a home interface, a remote interface, and a deployment descriptor
  - The API was invasive – all the interfaces and classes were directly tied to EJB specific types (via inheritance, implementing an interface, etc.)
  - Client programs were also exposed to some of this complexity
- Entity beans were not very useable
  - Some would say they were broken
  - The specification was also incomplete in terms of what was required in terms of ORM, and how it was to be done
  - They were not used very much

Notes:

EJB 3 Goals

- **Simplify development**
  - Annotations make programming simpler
  - Fewer classes required
  - API is simpler
- Use **POJO**s and **POJI**s
  - Bean implementations can be POJO (Plain Old Java Objects)
  - Remote interfaces can be POJI (Plain Old Java Interfaces)
  - These are much easier to program
  - Reduces coupling to EJB specific types
- Interceptor capability for session and message-driven beans
- Remove need for home interfaces

Notes:

EJB 3 Goals

- Make the most common usage easy
  - Defaults for most things, to reduce need for developer to specify common, expected behaviors
    - e.g. – no empty `ejbActivate()` methods as in EJB 2.x
- Resource and environmental dependencies easier
  - Dependency injection, annotations simplify programming
- Support entity persistence well
  - Simpler API
  - Much more powerful capabilities, including support for inheritance, polymorphism, relationships, embedded components
  - No required interfaces used - entities are POJOs

Notes:

EJB 3.1 and 3.2 Goals

- Improve usability and functionality of EJB, including:
  - EJB without business interface (for local view)
  - Define portable global JNDI names for lookup
  - Add singleton session bean capability
  - Easy-to-use calendar-based timer service
  - Simple asynchronous session beans
  - EJB packaging within WAR files
  - EJB lite - subset of complete EJB capability

Notes:

- EJB 3.1 and 3.2 are incremental updates to the EJB 3.0 specification
  - In general, they focus on increasing ease of use, and adding functionality in newly identified areas
  - The core functionality remains very similar to EJB 3.0

Session Bean Usage

- Session beans provide a number of useful capabilities
- **Transaction Management**
  - Session beans provide easy access to the transaction service available in EJB
  - Transactional behavior can be easily specified with annotations in the bean class
- **Resource Management / Concurrent Access**
  - EJB container manages bean instances, threads, memory issues
  - Very important for scalability in enterprise applications
- **Distributed Services / SOA**
  - Session beans can be accessed remotely (directly via RMI)
  - Can also be used to implement Web Services

Notes:

Session Bean Usage

- **Fault Tolerance / Scalability**
  - Most EJB containers support failover/high availability and some form of load balancing / clustering
- **Security**
  - Beans and individual methods in beans can be tied into the JEE security system to secure access to them
  - Can be done by setting security attributes (in bean class via annotations, or in XML DD)

Notes:

Persistent Entity Usage

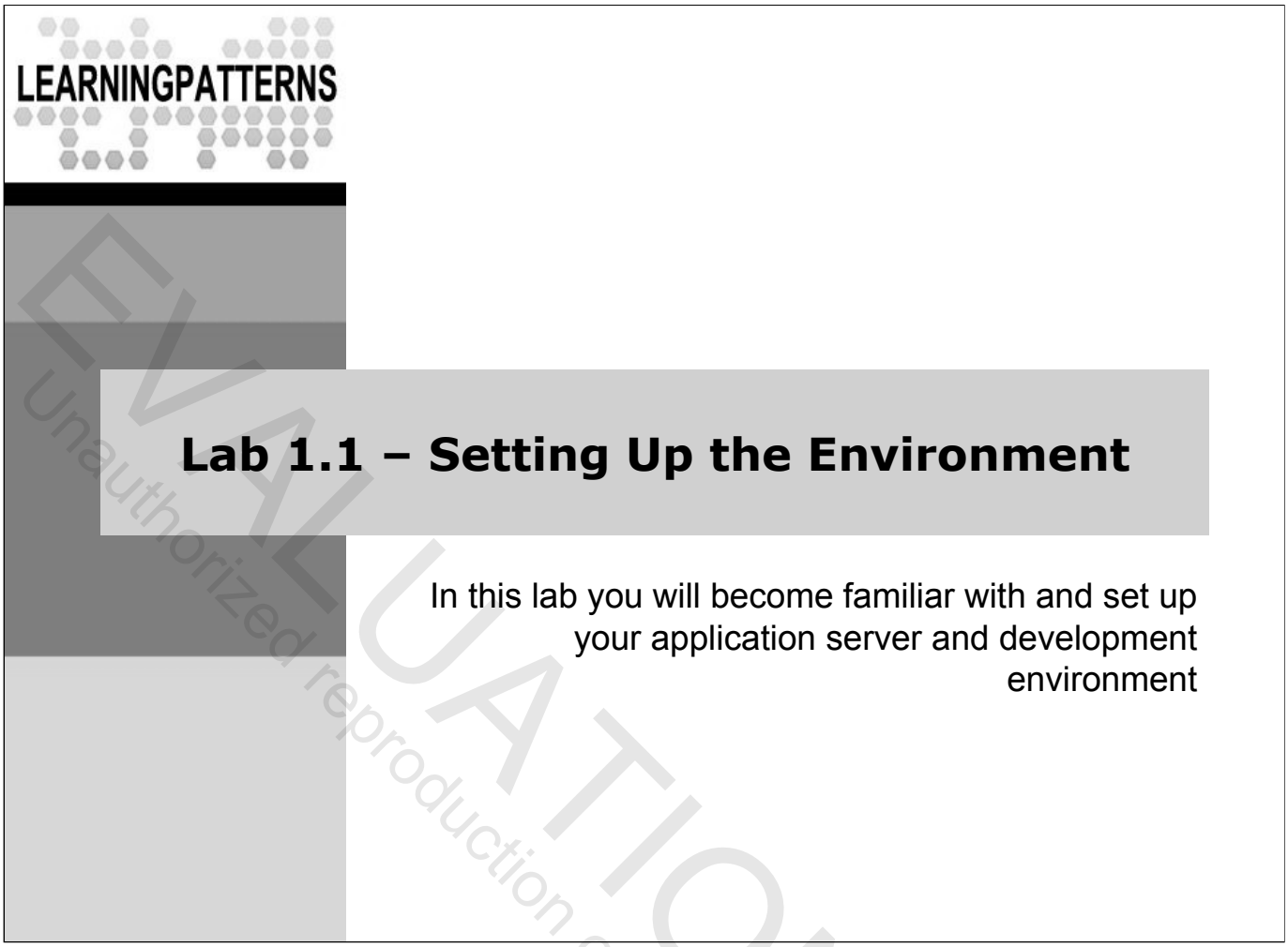
- **ORM** – Persistent entities are exclusively devoted to ORM
  - They model business data, and handle the interaction with the database
  - You define a **mapping** from the bean class to the database, and the **framework generates all the JDBC code**
  - Eliminates the need for complex, tedious, hand coded JDBC
- **Persistence available in Java SE** - The persistence framework (`javax.persistence`) can be used independently
  - Does not need Java EE server
  - Can be used directly from Java SE (Standard Edition) program

Notes:

MDB Usage

- **Integrate EJB/server with messaging**
  - Allows you to easily receive messages asynchronously on the server side
  - Previous to MDB, there was no standard way to do this
- **Transaction Management**
  - Allows you to easily start/control transactions when receiving JMS messages
  - Can't be done directly for asynchronous message receipt with the JMS API

Notes:



Notes:

Review Questions

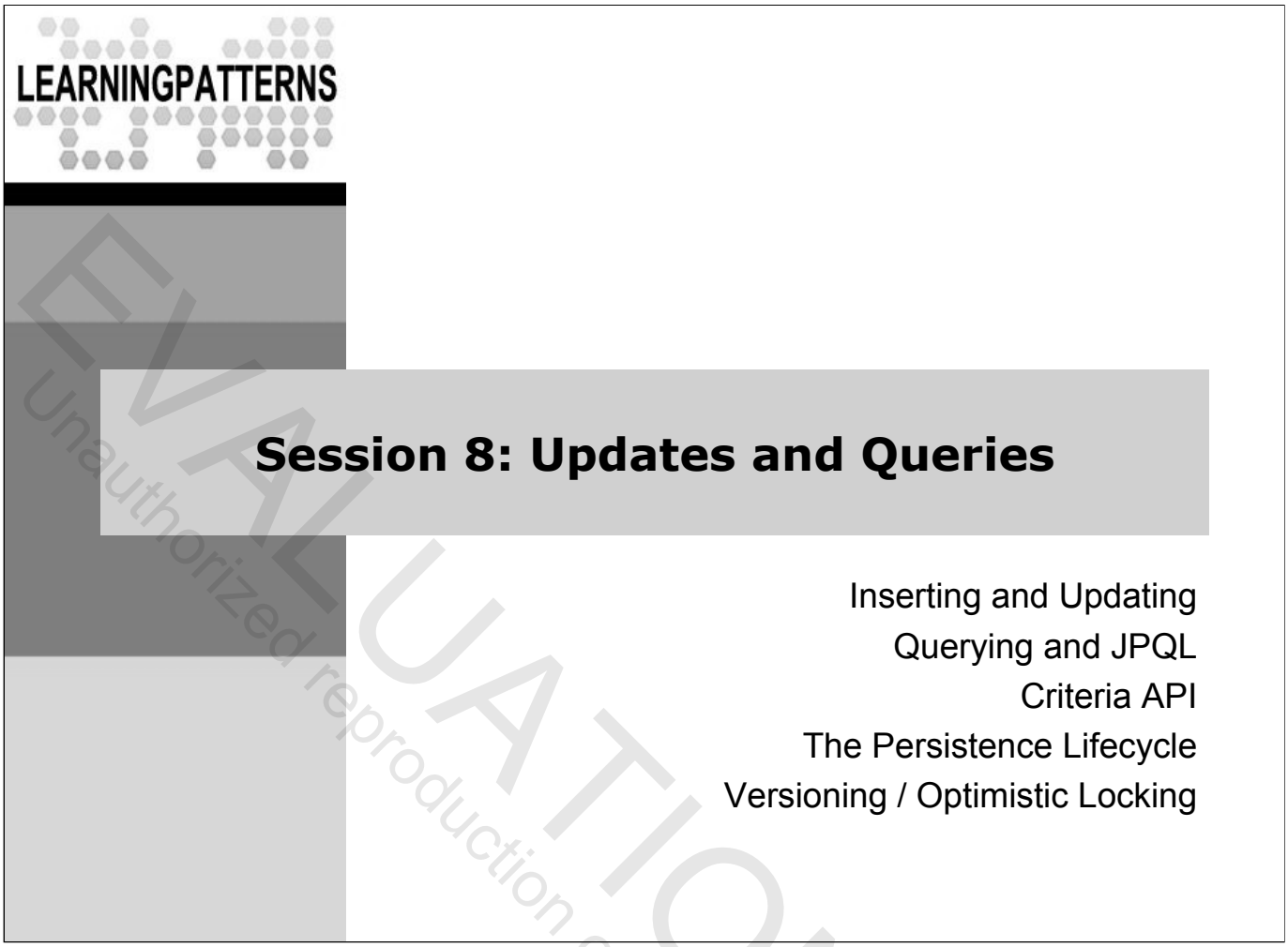
- What is EJB?
- What are the different kinds of beans defined in EJB, and what are they used for?
- How is EJB 3 better than previous versions of EJB?

Notes:

Lesson Summary

- **EJB** is a framework for creating **server-side components**
  - Transactional, Distributed, Portable, Reliable, Secure, Scalable
  - It also defines a persistence API
- EJB defines session beans, message-driven beans and persistent entities
  - **Session beans** provide distributed business services, and access to container services such as transactions, concurrency control, etc.
  - **Message-driven** beans receive JMS messages asynchronously, and integrate JMS with the EJB tier
  - **Persistent Entities** provide an ORM framework to map between Java objects and relational data

Notes:

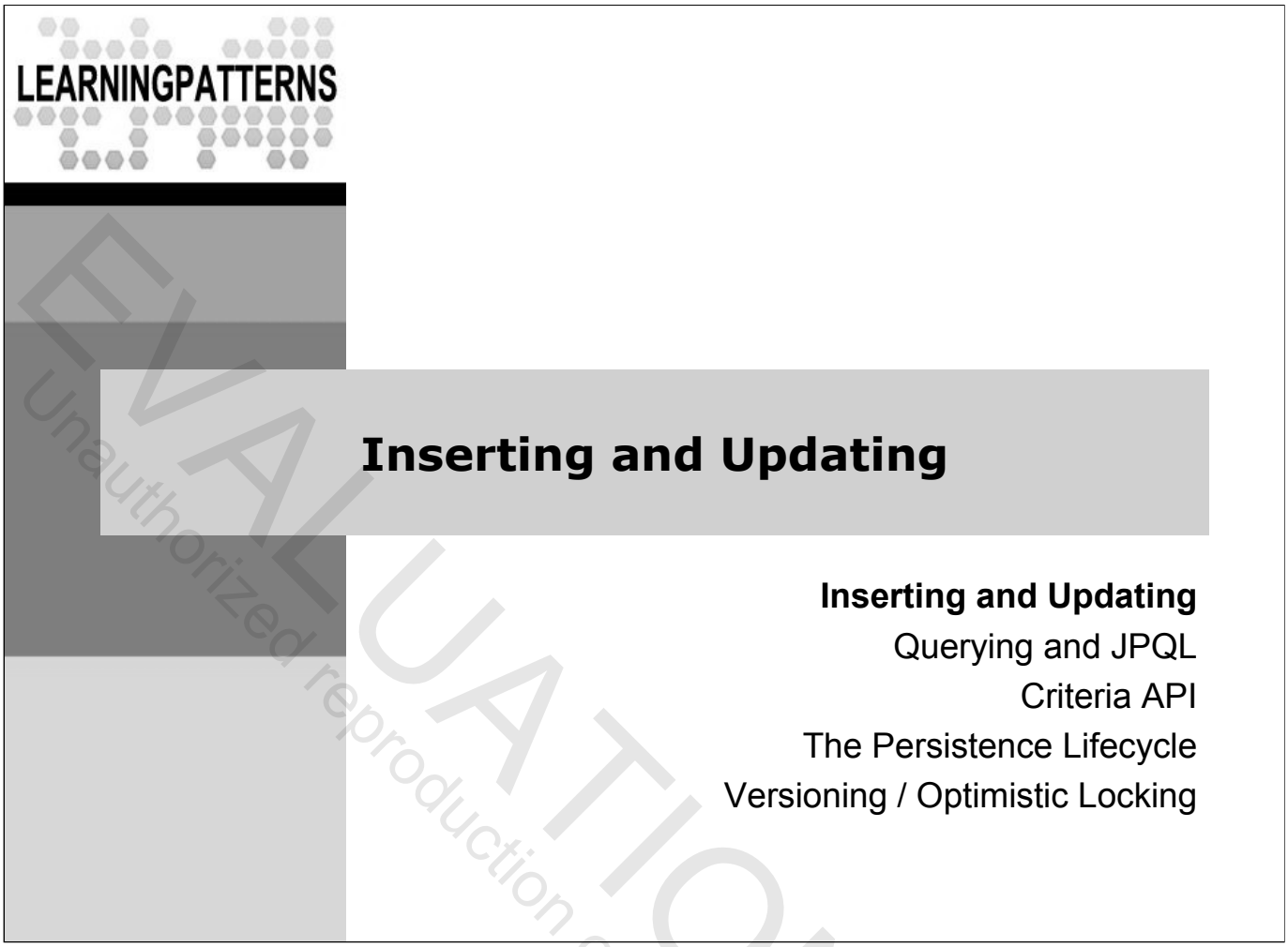


Notes:

Lesson Objectives

- Lean how to do updates and inserts on persistent objects
- Learn the Java Persistence Query Language (JPQL)
- Retrieve persistent objects from the DB using JPA and the query language
- Learn how to implement optimistic locking with JPA

Notes:



Notes:

Persisting a New Entity

- It's very easy to insert new instances (rows) into the DB
  - Simply create a new instance using *new*
  - Set values for the properties (except for the ID property)
  - Save the instance to the DB using an entity manager
- **EntityManager.persist()** is used to persist an instance
  `void persist(Object entity)`
- The instance will be inserted into the DB
  - When the transaction completes successfully, the insert will be made permanent
  - You can then retrieve its id value, if you need it

Notes:

- 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

Persisting a New Entity Example

```
// imports, etc. not shown ...
public class EventServiceBean implements EventService {

  @Inject EntityManager em;

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

```
// Code fragment
EventService es = ... // Event service lookup not shown.
Event newEvent = new Event();
newEvent.setTitle("A party");
es.addEvent(newEvent);
```


Notes:

- It is the responsibility of the application to ensure that an instance is managed in only a single persistence context
  - The behavior is undefined if the same Java instance is made managed in more than one persistence context.
- The `contains()` method can be used to determine whether an entity instance is managed in the current persistence context
- The contains method returns true:
  - If the entity has been retrieved from the database, and has not been removed or detached.
  - If the entity instance is new, and the persist method has been called on the entity or the persist operation has been cascaded to it.
- The contains method returns false:
  - If the instance is detached
  - If the remove method has been called on the entity, or the remove operation has been cascaded to it
  - If the instance is new, and the persist method has not been called on the entity or the persist operation has not been cascaded to it

Synchronization To the Database

- The call to **persist()** in the previous example may not immediately write to the database
  - The entity manager doesn't immediately write to the database when an object is saved or updated
  - This is a performance optimization to minimize writes
- The persistence provider performs database writes when the persistence context is synchronized to the database (at the latest at the end of a transaction)
  - We call this **flushing** the persistence context
- Committing a transaction will automatically flush all writes
  - You can also call `flush()` on the entity manager to force objects in memory to be synchronized to the database
  - You usually don't need to do this

Notes:

- The default behavior for a persistence context is to flush to the database when a transaction is committed
  - The persistence provider may also flush to the database whenever it determines it may be necessary
  - For example, it may flush before a query if the results of the query may be changed by pending writes
- We are using transaction-scoped persistence contexts here
  - This means that all activity happens in the context of a transaction
  - This is the most common type of persistence context
  - The behavior is different for other types of persistence contexts

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

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

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

Notes:

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,partyId);

// Remove the event
em.remove(partyEvent);
// When Tx commits, the deletion is persisted to database
```

Notes:

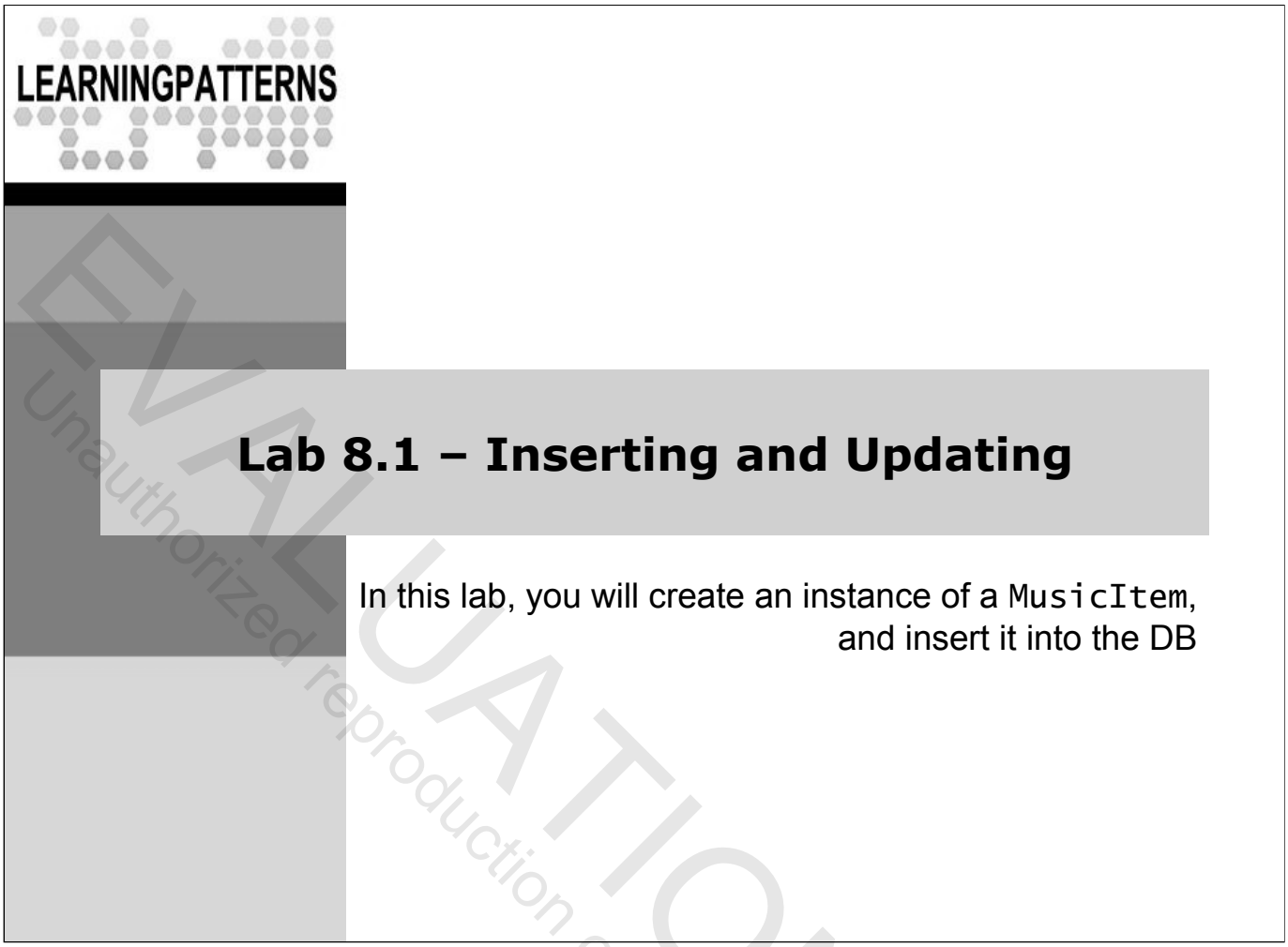
- The in-memory object becomes a transient instance again, with no representation in the database, and not in the scope of any persistence context
  - We'll talk about the lifecycle of persistent objects more later in a future session

Detached Entities

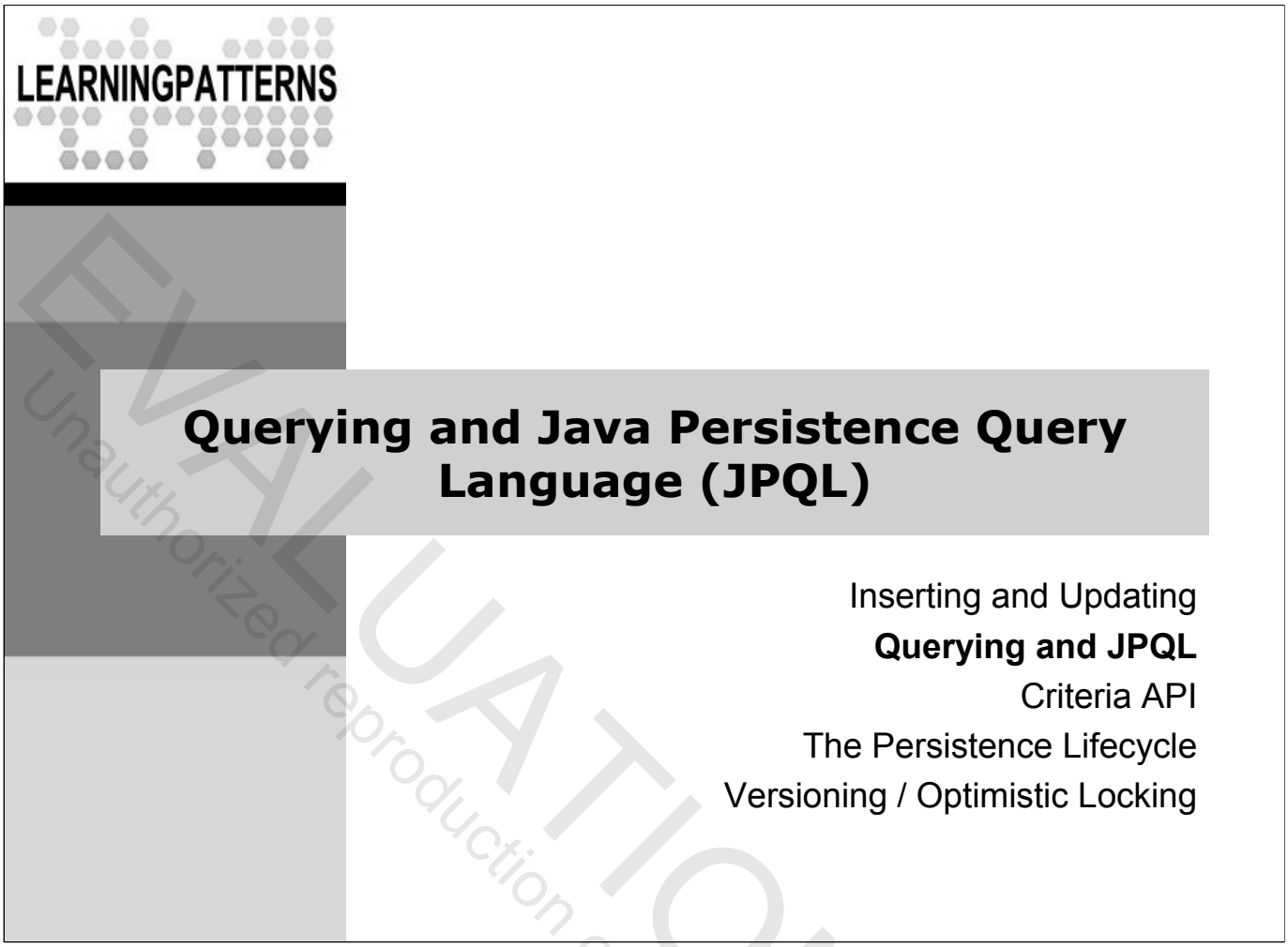
- It's also common in enterprise applications for an entity instance to remain in memory after the persistence context is closed
  - For example, you may retrieve an entity instance, and then pass it to the Web tier for a user to view, and perhaps modify
- Any entity instance that is no longer associated with an active persistence context is a **detached** object
  - We will look at this a little later when we talk about versioning and optimistic concurrency

Notes:

- A detached entity may result from the following:
  - Transaction commit if a transaction-scoped container-managed entity manager is used
  - Transaction rollback
  - Clearing the persistence context
  - Closing an entity manager
  - Serializing an entity or otherwise passing an entity by value—e.g., to a separate application tier, through a remote interface, etc.



Notes:



Notes:

Java Persistence Query Language

- **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
- Designed to query **object graphs**, rather than 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

Notes:

- 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

Notes:

- 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.

Querying and the Query Interface

- JPA provides a **`Query`** interface for executing queries
  - You obtain a Query instance from the entity manager using:

```
Query createQuery(String qlString);
```

  - 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 `Query` methods
  - **`List getResultList()`**: Return the query results as a list
    - Result objects with their properties are returned
  - **`Obect getSingleResult()`** : Convenience method to return a single instance that matches the query (null if no match)

Notes:

Executing a Query

- The example below uses `Query.getResultList()`
  - `getResultList` executes the query, retrieving all the entities into memory at once, and returns the result as a `java.util.List`
  - We use a for-each loop to go through the list

```
Query q = em.createQuery("SELECT e FROM Event e");
List resultList = q. getResultList();
for (Object cur : resultList) {
    Event e = (Event)cur();
    System.out.println("Event: " + e.getId());
}
```

- There are other useful methods on the Query type – e.g.
  - **`Query setMaxResults(int maxResults)`**
    - Sets maximum number of rows to retrieve
  - **`Query setFirstResult(int startPosition)`**
    - Sets the first row to retrieve
  - Look at the documentation !

Notes:

JPA 2 – Generic Query Enhancements

- JPA 2 provides the **TypedQuery** interface, which enhances type safety using generics
  - Accessed from the entity manager via this method (see notes):

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

- TypedQuery has generic versions of the query methods, e.g.
  - **List<X> getResultList()**: Return the query results as a typed list
  - **<X> getSingleResult()** : Single, typed, result
  - Of course, <X> will be the same type as <T> - the type you created the TypedQuery with

Notes:

- 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

JPA 2 – Generic Query Enhancements

- In the example below, everything is written in terms of Event
  - It is more type safe, and does not require casting
  - Otherwise, the code works basically the same as our earlier code

```
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());
}
```

- Other methods are similarly parameterized, e.g.
  - `TypedQuery<X> setMaxResults(int maxResults)`
  - `TypedQuery<X> setFirstResult(int startPosition)`
- There are also changes to the `Query` interface – we'll look at more of this later

Notes:

Other Query Methods

- There are a number of other query methods not directly related to performing the query
  - These all return the `Query` or `TypedQuery` instance itself unless shown differently below
  - **`int getFirstResult()`**: Position of first result that was set
  - **`FlushModeType getFlushMode()`**: Get flush mode
  - **`setFlushMode(FlushModeType flushMode)`**: Set flush mode
  - **`LockModeType getLockMode()`**: Get lock mode
  - **`setLockMode(LockModeType lockMode)`**: Set the lock mode
  - **`java.util.Map<java.lang.String,java.lang.Object> getHints()`**: Get hints in effect for this query
  - **`setHint(java.lang.String hintName, java.lang.Object value)`**: Set a query property or hint (see notes)
  - We'll talk more about some of these concepts (e.g. lock mode and flush mode) later

Notes:

- We'll talk more about when the runtime flushes to the database later in the course
  - The docs say this about the flush mode
    - When queries are executed within a transaction, if FlushModeType.AUTO is set on the Query or TypedQuery object, or if the flush mode setting for the persistence context is AUTO (the default) and a flush mode setting has not been specified for the Query or TypedQuery object, the persistence provider is responsible for ensuring that all updates to the state of all entities in the persistence context which could potentially affect the result of the query are visible to the processing of the query. The persistence provider implementation may achieve this by flushing those entities to the database or by some other means.
    - If FlushModeType.COMMIT is set, the effect of updates made to entities in the persistence context upon queries is unspecified.
    - If there is no transaction active, the persistence provider must not flush to the database.
- The JPA spec defines the following hint for use in queries

```
javax.persistence.query.timeout // time in milliseconds
```

  - Portable applications should not rely on this hint. Depending on the persistence provider and database in use, the hint may or may not be observed.
  - Vendors are permitted to support the use of additional, vendor-specific locking hints

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

Notes:

JPQL Operators and Expressions

- JPQL supports the same basic operators as SQL
  - Unary positive and negative: +, -
  - Regular arithmetic operations on numeric values: *, /, +, -
  - Binary comparison operators: =, <, >, <=, NOT, BETWEEN, etc.
  - Binary operators on collections: IS [NOT] EMPTY, [NOT] MEMBER [OF]
  - Logical operators for ordering expression evaluation: NOT, AND, OR
- JPQL also supports familiar literals
  - Strings in single quotes, e.g. `'Jane Doe'`
  - Java integers / floating point, including valid suffixes *
  - Dates using JDBC syntax – e.g. `e.date < {d '2010-12-31'}`

Notes:

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

Query Parameters

- JPQL allows you to specify and populate query parameters in a way similar to JBDC prepared statements
  - **Named parameters**, which are not available in JDBC, are preferred because they:
    - Are insensitive to the order they occur in the query string
    - May occur multiple times in the same query
    - Are self-documenting

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

  - **Positional Parameters** also available:
    - Cumbersome, but familiar to JDBC programmers
    - NOTE: **numbering starts at 1**, as for JDBC

```
- SELECT e FROM Event e WHERE e.id > ?
```

- **Never use simple string concatenation** to build queries (see note)

Notes:

- Note that the positional parameters are consistent in numbering with JDBC
  - This is different from Hibernate, where numbering starts from 0
  - This difference from JDBC was a potential cause for confusion/bugs in Hibernate programs
- You should never use simple string concatenation to create your query strings
  - This is very inefficient, because each query is created anew, and the query can't be cached
  - It is also a large security hole, as it leaves your query open to an SQL injection attack
- Parameters prevent SQL injection attacks, because the query itself is never altered
  - The text of the parameter string is effectively quoted by the database, preventing injection of SQL strings

Using Query Parameters

- Query provides several methods to set query parameters
  - **`setParameter(String name, Object value)`**: Set the named parameter to the given value
  - **`setParameter(int position, Object val)`**: Set by position
  - **`setParameter(String name, Date value, TemporalType temporalType)`**: Set parameter to a date
  - These methods all return the query instance itself
  - There are a few other variations on `setParameter` *

```
// 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();
```

```
// Positional parameter example – find events by title
TypedQuery<Event> q = em.createQuery(
     " SELECT e FROM Event e WHERE e.title = ?", Event.class);
q.setParameter(1, "Party");
List<Event> l = q.getResultList();
```

Notes:

- There are basically three kinds of `setParameter` methods
  - For all of them, the first argument is always the name or number (for named and positional parameters respectively)
  - The second argument is the value to be set
  - A `Date` and `Calendar` value require a third argument specifying whether it is a date, time, or timestamp
  - See the docs for all the different variations

- The example in the slide example finds all the events where the title is equal to "Party"

Named Queries

- **Named queries** let you define queries in the mapping file
  - Scattering SQL in your code makes it difficult to maintain
  - Named queries are also more efficient, because the runtime can parse this query once (at startup) and save it
  - **@NamedQuery** defines a a named query, as shown below
  - It can appear on any entity class
  - The scope of the query name is the persistence context, so common practice is to prefix the query with the name of the entity

```
// Imports omitted

@Entity
@NamedQuery(name="Event.findByTitle",
            query="SELECT e" +
                  "FROM Event e" +
                  "WHERE e.title = :title")
public class Event implements java.io.Serializable {
   // ...
}
```

Notes:

- Named queries can be pre-compiled by the provider's query optimizer
  - They don't change once they are defined
  - Dynamic queries (those made in your code with the createQuery method) will have to be compiled each time they are encountered
- In general, using named queries is considered a best practice
- Named queries that are scoped to specific entities will be available in a future release of JPA
- Named queries may also be declared in XML mapping files

Named Queries

- You can define multiple named queries with `@NamedQueries`
  - Which takes an array of `@NamedQuery` elements

```
@NamedQueries({
  @NamedQuery(name="Event.findByTitle",
              query="SELECT e FROM Event e WHERE e.title = :title"),
  @NamedQuery(name="Event.findByMinDate"
              query="SELECT e FROM Event e WHERE e.date > :date") })
```

- Create the query with `EntityManager.createNamedQuery()`
  - This is the JPA 2 type-safe version – see notes for JPA 1
  - We've chained the calls necessary to create/run the query *

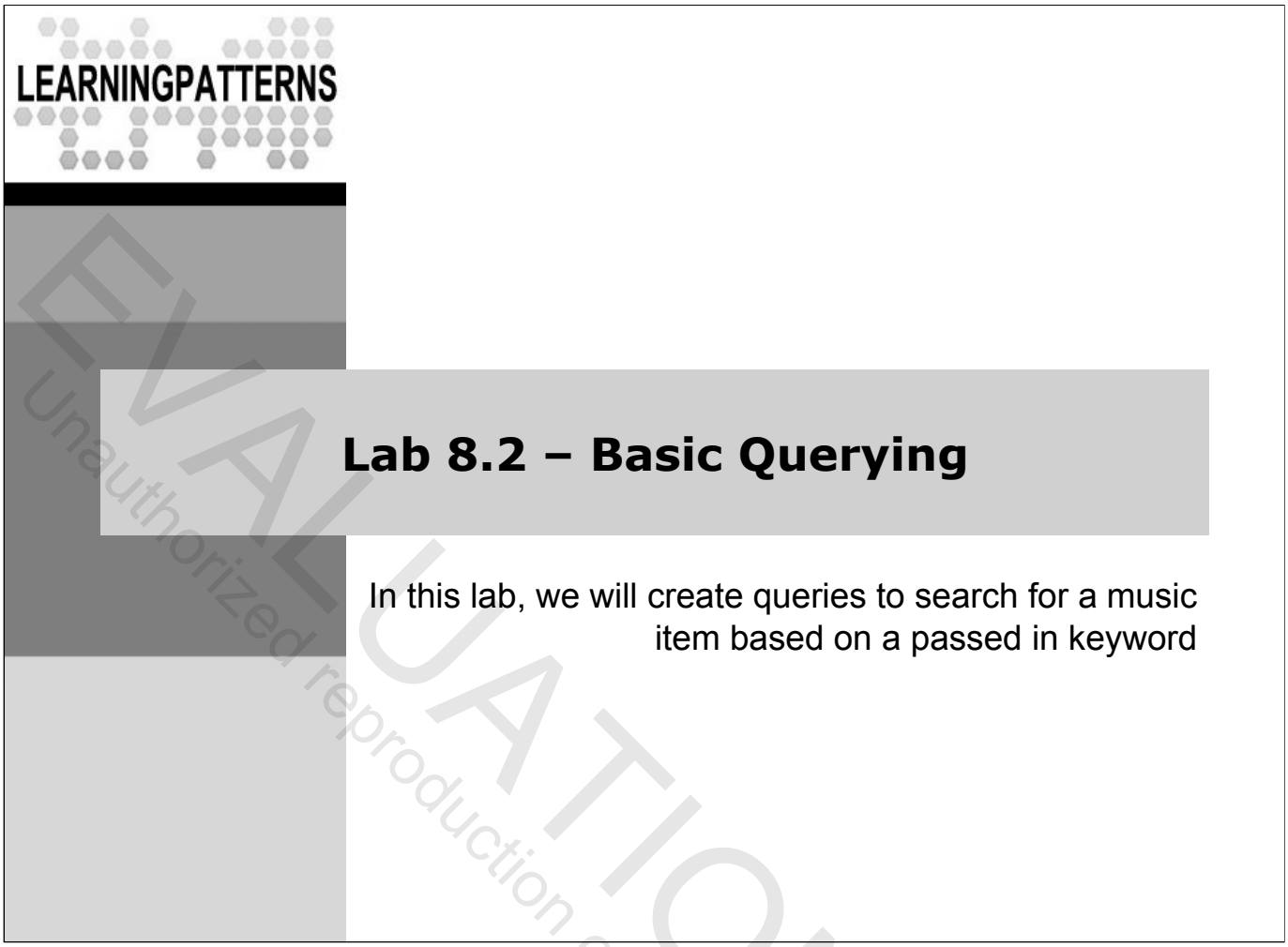
```
List<Event> results = em.createNamedQuery("Event.findByTitle",
                                           Event.class)
                        .setParameter("title", "Party")
                        .getResultList();
```

Notes:

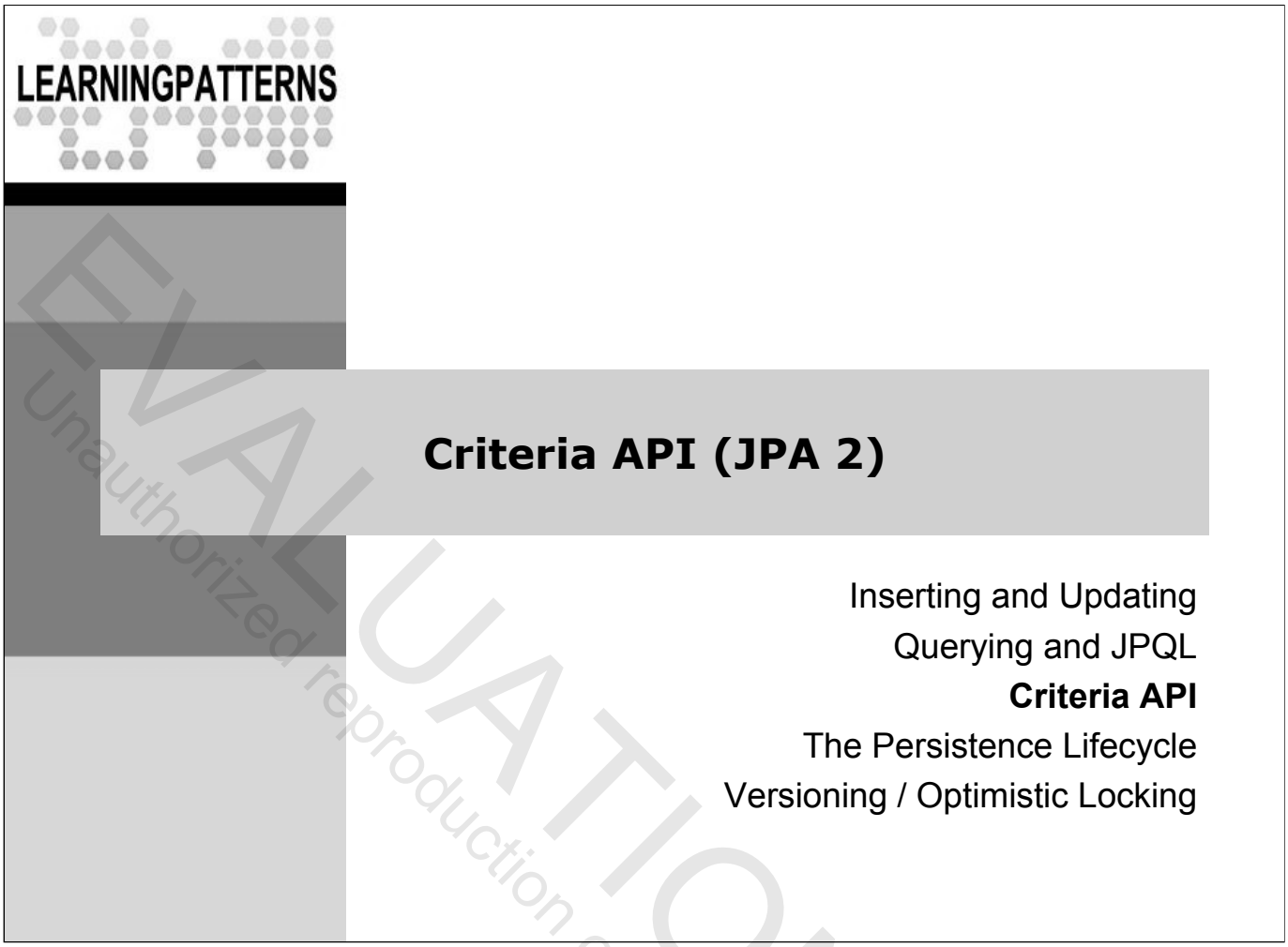
- Many of the Query methods return the query instance itself
  - This allows you to chain the query calls as in the slide example
  - This is very common practice, and makes for short, easily readable code
  - The formatting shown above is also common practice, and makes the code more readable than having it all on one line

- The untyped, JPA 1 version of the query in the slide would be:

```
List results = em.createNamedQuery("Event.findByTitle")
                 .setParameter("title", "Party")
                 .getResultList();
```



Notes:



Notes:

Criteria Overview

- The **Criteria API** allows you to create object-based queries
  - Rather than string-based using JPQL
  - Useful when you want to build a query programmatically
  - The queries are still **entity-based** – querying over the same abstract schema as JPQL
- This extensive API (in **`javax.persistence.criteria`**) includes the following interfaces:
  - **`CriteriaBuilder`**: Constructs criteria queries, compound selections, expressions, predicates, orderings
  - **`CriteriaQuery`**: Functionality specific to top-level queries
  - **`Expression`**: Expression for queries
  - **`Order`**: Defines ordering over query results
  - **`Predicate`**: An expression containing restrictions
  - **`Selection`**: Defines item to be returned in a query result

Notes:

- There are times when it is easier to build a query programmatically than to create a query string
  - For example, when someone is entering a search query through some sort of user interface, and you need to execute that query
- From the JPA spec [JPA 2 Specification, Final, 6.4]
  - The javax.persistence.criteria API interfaces are designed both to allow criteria queries to be constructed in a strongly-typed manner, using metamodel objects to provide type safety, and to allow for string-based use as an alternative:
    - Metamodel objects are used to specify navigation through joins and through path expressions. Typesafe navigation is achieved by specification of the source and target types of the navigation.
    - Strings may be used as an alternative to metamodel objects, whereby joins and navigation are specified by use of strings that correspond to attribute names.
  - Using either the approach based on metamodel objects or the string-based approach, queries can be constructed both statically and dynamically. Both approaches are equivalent in terms of the range of queries that can be expressed and operational semantics.

A Simple Criteria Example (1 of 3)

- **`CriteriaBuilder.createQuery(Class<T> resultClass)`** is used to create a `CriteriaQuery` instance (see notes)
  - It returns `CriteraQuery<T>` a query against the class argument
  - The example at bottom executes the equivalent of the query **`SELECT e FROM Event e`** which selects all the `Event` instances
  - `CriteriaBuilder` creates a `CriteriaQuery<Event>`, which is then used by the EM to create the query
  - Note: A `CriteriaQuery` is the equivalent of a JPQL string – it is not the actual query object
  - Next, we'll look at the code below in detail

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Event> cq = cb.createQuery(Event.class);
Root<Event> ev = cq.from(Event.class);
cq.select(ev);
TypedQuery<Event> tq = em.createQuery(cq);
List<Event> allEvents = tq.getResultList();
```

Notes:

- The full signature of createQuery is:

  ```
  <T> CriteriaQuery<T> createQuery(java.lang.Class<T> resultClass)
  ```

  - It is parameterized by the result class
- There is also a non-parameterized version with the signature

  ```
  CriteriaQuery<java.lang.Object> createQuery()
  ```

  - A CriteriaQuery object is created by means of one of the createQuery methods or the createTupleQuery method of the CriteriaBuilder interface. A CriteriaQuery object is typed according to its expected result type when the CriteriaQuery object is created. A TypedQuery instance created from the CriteriaQuery object by means of the EntityManager createQuery method will result in instances of this type when the resulting query is executed.
  - The following methods are provided for the creation of CriteriaQuery objects:
    - <T> CriteriaQuery<T> createQuery(Class<T> resultClass);
    - CriteriaQuery<Tuple> createTupleQuery();
    - CriteriaQuery<Object> createQuery();

A Simple Criteria Example (2 of 3)

- Criteria have equivalents of all the JPQL select clauses
  - SELECT, FROM, WHERE, ORDER BY, GROUP BY, HAVING
- A query root corresponds to an identification variable in FROM
  - **`Root<X> from(Class<X> entityClass)`** (in `CriteriaQuery`) creates and adds a query root for the given entity
  - This code returns an instance of Root corresponding to the Event type – the e in our equivalent JPQL "FROM e" clause

```
Root<Event> ev = cq.from(Event.class);
```

- `CriteriaQuery.select()` adds a select clause to a query
  - Passing a `Root` to select indicates we want the entity to be the result of the query (`Root` extends `Selection`)
  - This code adds the equivalent of the SELECT e in our JPQL

```
cq.select(ev);
```

Notes:

- The full signature of select is

```
CriteriaQuery<T> select(Selection<? extends T> selection)
```

  - It takes a `Selection` object as a parameter
  - `Selection` is extended by many other interfaces, including `Root`, `Predicate`, `Join`, and others
  - See the documentation !
- From the JPA spec [JPA 2 Specification, Final, sec. 6.5.2]
  - A CriteriaQuery object defines a query over one or more entity, embeddable, or basic abstract schema types. The root objects of the query are entities, from which the other types are reached by navigation. A query root plays a role analogous to that of a range variable in the Java Persistence query language and forms the basis for defining the domain of the query.
  - A query root is created and added to the query by use of the from method of the AbstractQuery interface (from which both the CriteriaQuery and Subquery interfaces inherit). The argument to the from method is the entity class or EntityType instance for the entity. The result of the from method is a Root object. The Root interface extends the From interface, which represents objects that may occur in the from clause of a query.
  - A query may have more than one root. The addition of a query root has the semantic effect of creating a cartesian product between the entity type referenced by the added root and those of the other roots.

A Simple Criteria Example (3 of 3)

- Creating the `CriteriaQuery` is the equivalent of creating a JPQL string
  - To create an actual query object, you pass the `CriteriaQuery` to `EntityManager.createQuery`
  - This code creates the query object for our criteria query

```
TypedQuery<Event> tq = em.createQuery(cq);
```

  - You can change the `CriteriaQuery` after passing it to the entity manager, but it will not effect the query already created
- Once you have the query object, you execute it as you would a JPQL query – it works exactly the same
  - The criteria query is just an object-based representation of the equivalent JPQL, and produces the same results

Notes:

- One might think, from the name `CriteriaQuery`, that it is an actual query object
  - You can see from the code though, that it is used by the EM to create a `TypedQuery`
  - It is more accurately thought of as a Java representation of the equivalent JPQL query string

Path Expressions

- Consider the query:

  `SELECT e FROM Event e WHERE e.title="Party"`
  - How would you reference e.title with a criteria query
- Path expressions (e.g. e.title) are supported via `Path.get()` - there are several versions, but we'll use the simplest:

  `<Y> Path<Y> get(java.lang.String attributeName)`
  - This creates a path to the named attribute
- To access `e.title`, we could use the code at bottom
  - We create a variable to hold the path `ev.get("title")`
  - Often, we create these inline, without a variable, as we'll see next

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Event> cq = cb.createQuery(Event.class);
Root<Event> ev = cq.from(Event.class);
Path pathToTitle = ev.get("title");
```

Notes:

- `Root` extends `Path`, and inherits all the methods of `Path`, including the `get()` methods

WHERE Clauses

- WHERE clauses are supported via `CriteriaQuery.where()`, and `CriteriaBuilder` methods which act as factories for expressions:
  - `CriteriaBuilder` contains methods that support all the predicates, methods and functions available in the JPQL, e.g:
  - `lessThan()`, `equal()`, `isNull()`, `isEmpty()`, `all()`, `avg()`
  - These can be used to generate arguments to `where()`
- The code at bottom is equivalent to **`WHERE e.title="Party"`**
  - `cb.equal(ev.get("title"), "Party")` generates a predicate comparing the event title to "Party", which is then passed to `where`
  - You could then execute the query as before

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Event> cq = cb.createQuery(Event.class);
Root<Event> ev = cq.from(Event.class);
cq.where(cb.equal(ev.get("title"), "Party"));
```

Notes:

- We are using a path expression (generated from the Root instance) and a where clause (generated using the CriteriaBuilder instance)
  - Remember that the JPQL equivalent is:

    `SELECT e FROM Event e WHERE e.title="Party"`
  - Let's compare our code to the JPQL
  - The path expression `ev.get("title")` is equivalent to `e.title`, since the root instance (`ev`) is equivalent to the identification variable "`e`" in our JPQL equivalent
  - `cb.equal(ev.get("title"), "Party")` is equivalent to `e.title="Party"`
  - `cq.where(…)` is equivalent to `WHERE e.title="Party"`

Typed Path Expressions

- There are times when the use of generics requires explicitly declaring the type of a path expression
  - Consider the query: **`SELECT e.title FROM Event e`**
  - This query returns a collection of string, so the query will be parameterized by `<String>` as shown below
  - Problem: `ev.get("title")` returns a generic object, so `cg.select(ev.get("title"))` returns objects
  - This will not compile, as the query is written in terms of strings
  - Using `ev.<String>get("title")` solves this problem; it is the (obscure) way to parameterize the return of get as a string

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<String> cq = cb.createQuery(String.class);
Root<Event> ev = cq.from(Event.class);
cq.select(ev.<String>get("title"));
TypedQuery<String> tq = em.createQuery(cq2);
List<String> titles = tq2.getResultList();
```

Notes:

- The get method is defined as follows:

  ```
  <Y> Path<Y> get(java.lang.String attributeName)
  ```

  - Note that the return type is parameterized
  - The way to specify the parameterization as a string is using the `ev.<String>get("title")` syntax
  - This admittedly obscure syntax is purely a result of the use of generics in the Criteria API, and it is part of Java generics, and not specific to criteria
  - The criteria API uses generics a great deal, and this sometimes complicates the syntax, though it does lead to more type safety

Typed Path Expressions

- Now consider the JPQL query:

```
SELECT e FROM Event e
    WHERE e.date < date('1990-01-01')
```

  - This checks for events where the date is less than 1990-01-01 *
  - The code at bottom does this – note that once again we need to parameterize the get method - using `ev.<Date>get("date")`
- `lessThan()` requires the arguments implement `Comparable`
  - Object (default return of get) doesn't, so we parameterize the call

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Event> cq = cb.createQuery(Event.class);
Root<Event> ev = cq.from(Event.class);
cq.select(ev).where(
    cb.lessThan(ev.<Date>get("date"), new Date(90, 1, 1)));
TypedQuery<Event> tq = em.createQuery(cq);
List<Event> allEvents = tq.getResultList();w
```

Notes:

- The JPQL clause below uses the date function to generate a date value from the ANSI formatted date string1990-01-01

```
e.date < date('1990-01-01')
```

  - This is the same as the JDBC standard uses

- Note also that we've chained the call to select and where
  - cq.select(ev).where(…)
  - This is common practice, as select() returns the query object

Combining Predicates (and/or)

- CriteriaBuilder includes methods to combine predicates:
  - **conjunction()**: A conjunction (and) that is always true
  - **disjunction()**: A disjunction (or) that is always false
  - **and(Expression<Boolean>x, Expression<boolean y)**: Conjunction of the given expressions
  - **or(Expression<Boolean>x, Expression<boolean y)**: Disjunction of the given expressions
  - The example selects events where date<1990-01-01, and the title="Party"

```
CriteriaBuilder cb = em.getCriteriaBuilder();
CriteriaQuery<Event> cq = cb.createQuery(Event.class);
Root<Event> ev = cq.from(Event.class);
Predicate criteria = cb.conjunction();
criteria = cb.and(criteria,
   cb.lessThan(ev.<Date>get("date"), new Date(90, 1, 1)));
criteria = cb.and(criteria,cb.equal(ev.get("title", "Party")));
cq.select(ev).where(criteria);
```

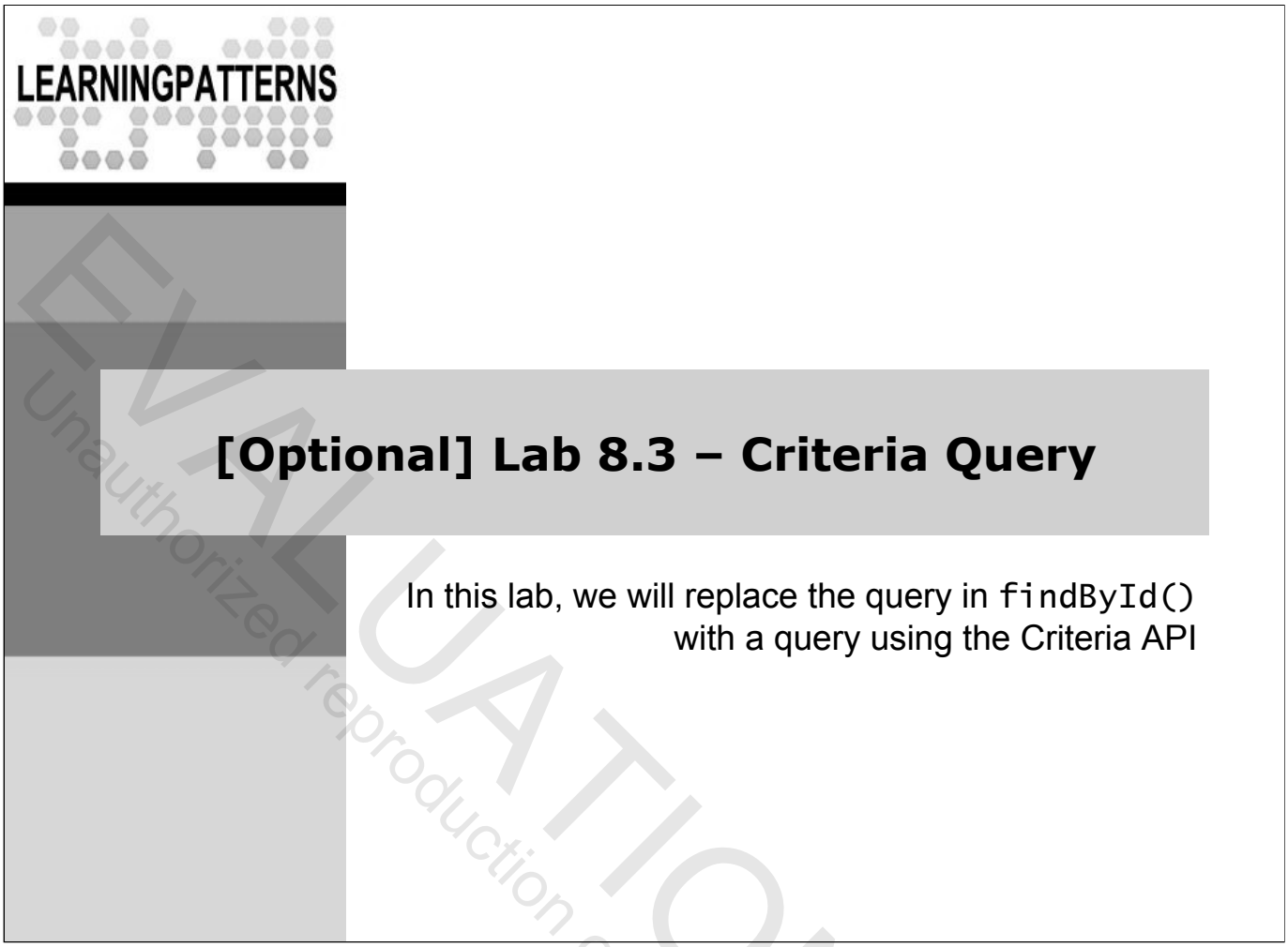
Notes:

- You can also just chain the creation of the criteria
  - That is a little harder to read in the code

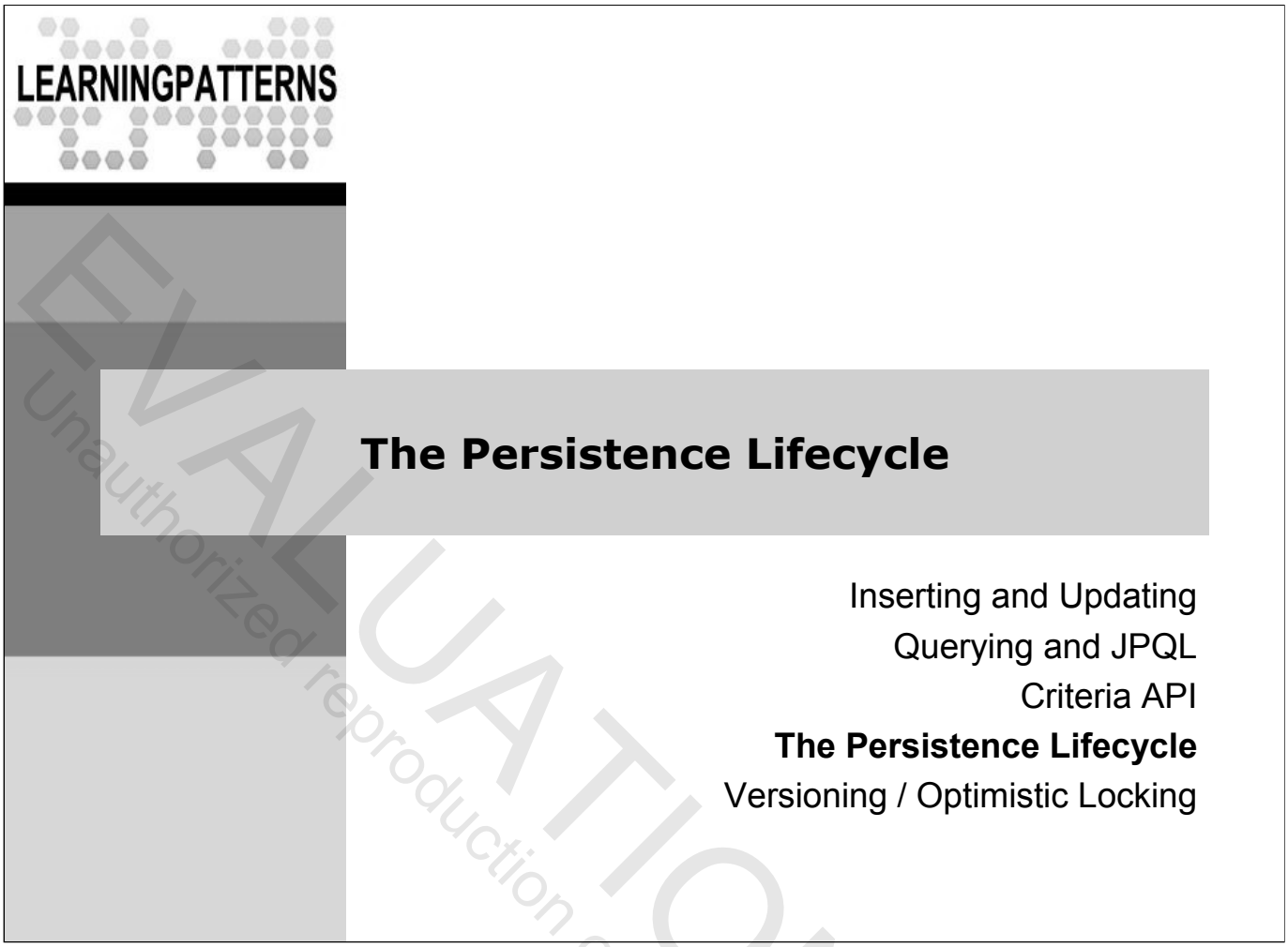
Additional Criteria API Capabilities

- We covered a portion of the API called the string-based API
  - This uses string names to refer to attributes in path expressions
- There is another, more strongly typed, approach that uses the Criteria MetaModel API
  - Coverage of this API is beyond the scope of the course
  - It uses many of the same types and methods, but instead of string names for attributes, uses types from the MetaModel API
- There is a huge amount of capability in the criteria API
  - For every capability in JPQL there is a way to accomplish the same thing using the Criteria
  - We've given you a good start in understanding the structure, but have not covered all the details (it would take all week!)
  - To get further information, the javadocs and the JPA 2 specification are a good start

Notes:



Notes:



Notes:

The Persistence Lifecycle

- Applications need to interact with the JPA persistence service when they are:
  - Propagating state in memory to the database
  - Propagating database state into memory
- This involves the following parts/concepts in a JPA system:
  - **The persistence context**: A cache of persistent entities associated with an entity manager
  - **Unit-of-Work**: A set of operations that are considered atomic and that define the scope of a persistence context
  - **Entity Lifecycle**: The states a persistent entity can be in
- All of these interact with each other, and understanding them is crucial to understanding how JPA works
  - We'll cover all of these now

Notes:

JPA Entity States

- With JPA, an instance of a persistent class can be in one of the following states
  - **New** (or **Transient**): Newly created and not saved
  - **Managed** (or **Persistent**): Has a persistent identifier, and is in the scope of a persistence context
  - **Detached**: Has a persistent identifier, exists in the database, but is not in the scope of a persistence context
  - **Removed**: Scheduled for removal from the database
  - Creating and working with entity instances and the session will cause transitions between the states
  - Generally, in a JPA application, you think about these states, and not about SQL statements being generated
  - JPA takes care of the SQL
  - Let's look at these states in detail

Notes:

Transient and Persistent State

- **New (Transient)** instance: Has just been instantiated using new, and hasn't been associated (saved) with an entity manager
  - No representation in DB, no identifier assigned
  - Not a transactional object - Modifications not known to the persistence context
  - If instance is lost, the data is lost
  - Become persistent when saved via the entity manager, or by creating a reference to it from another persistent object
- **Managed (Persistent)** instance: Has a primary key value
  - Possibly has a representation in the DB
  - May also be an instance retrieved by a query
  - By definition it is in a persistence context (covered next)
  - JPA will detect any changes to the instance and synchronize the state with the database when the unit of work completes

Notes:

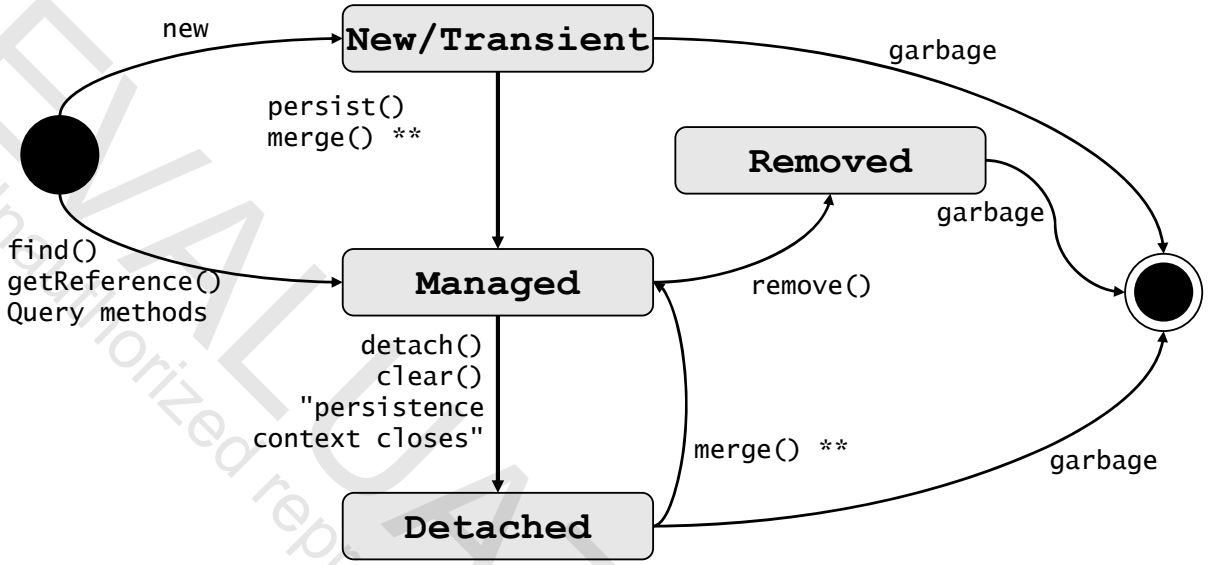
- We'll soon see that a persistent instance may not yet have a representation in the database because the entity manager may not have been flushed

Detached and Removed State

- **Detached** instance: Has a persistent id, was persistent, but no longer is (Either persistence context closed, or instance detached)
  - The reference is valid, and the instance may be modified
  - Can be reattached to an entity manager later, making it persistent
  - Persistent objects become detached when their entity manager ends, or when explicitly removed from the entity manager via `detach` or `clear`
  - We will look at this in more detail later
- **Removed** instance: An instance scheduled for removal from the database at the end of a unit of work
  - Is still managed by a persistence context and has a persistent id
  - Calling `remove()` schedules the the instance for removal (the removal of the database data), but does nothing to the in-memory instance
  - Only applicable to managed entities
  - A removed entity will be removed from the database at or before transaction commit or as a result of the flush operation

Notes:

JPA Object States and Transitions



Notes:

- All of the methods listed are methods on the `EntityManager`, except for methods on the `Query`
- `merge()` returns a new persistent instance
  - The original (detached) object doesn't change state
  - We'll talk about detached objects shortly
  - If `merge()` is called on a new instance, then a copy is made, and this copy is persisted as if `persist()` was called on it – the copy is then returned
- Once objects are no longer referenced, they are free to be garbage collected
  - At this point their life ends

The Persistence Context

- A **persistence context**, consisting of all currently managed entities, is associated with every active entity manager
  - All currently managed entities (all entities in the current unit of work) are stored in the persistence context
  - In fact, you can consider the persistence context to be a **cache** of managed entity instances
- The persistence context provides the following:
  - Functions as a first level cache
  - Guarantees a scope of object identity
  - Implements TX write-behind and automatic dirty checking
- The persistence context is **not directly visible** in JPA
  - No class or API for it – all interaction through the entity manager

Notes:

- The persistence context is not really visible in your program
  - There are no API calls that deal with it
  - It is an internal part of the entity manager which is always present
  - It is important to understand how it works
- From the JPA spec: "A persistence context is a set of managed entity instances in which for any persistent entity identity there is a unique entity instance. Within the persistence context, the entity instances and their lifecycle are managed by the entity manager."

Persistence Context Lifespan

- A persistence context represents a single **unit of work**
  - A unit of work can vary, depending on your needs
    - It may span a single operation, or (more likely) a DB transaction
    - It may even span several DB transactions
  - The lifespan of the persistence context is bounded by the beginning and end of the unit of work (sometimes called a conversation, or a "logical" transaction)
- Java SE typically uses an **application-managed** EM
  - The EM is explicitly created, used in a unit-of-work, and closed by code in your application (as in our earlier examples)
  - The EM and persistence context **have the same lifespan**
  - This works differently for a **container-managed** EM, such as a Java EE environment, as we'll see later

Notes:

Persistence Context Propagation

- Your persistence context (equivalent to application-managed EM) must be propagated where it's needed in a unit of work
  - You want to **use the same EM** (and its associated persistence context) within the unit of work
  - The **lifespan** of the EM must match that of the unit-of-work
  - You propagate the EM, and control its lifespan in your code for an application-managed EM
- Consider a system more complex than the ones we've used
  - Consider if we have an Event entity, an EventLocation Entity, and a Data Access Object (DAO) for each encapsulating the JPA code
  - If we need to use both DAOs in the same unit of work - you'll need to pass the EM reference to all of them
  - For container-managed EMs, this is easier

Notes:

- Data Access Objects are a common pattern that encapsulate all the workings of a particular persistence implementation
  - They may be implemented with straight JDBC, or with a tool like JPA
  - To the DAO client, it makes no difference
- If using a DAO based on JPA, there will need to be a way for it to access the EM
  - One simple way is to have methods that set the EM in each DAO
  - This is possible but cumbersome
  - And where does the code using the DAO get the EM from
- We'll look later at several integration strategies using different technologies that answer this question – specifically EJB3 and Spring

The Persistence Context as Cache

- The persistence context is a first-level cache for your entities
  - It is a cache that is associated with the EM
  - We'll look later at the second-level cache which is associated with the `EntityManagerFactory` and shared by all its EMs
- One important job of the first-level cache is to optimize the SQL generated by the JPA provider
  - JPA can use the cache to minimize the number of queries generated within a transaction
  - Updates to the database can be minimized
  - Queries to the database can be reduced because some entities may already be in the cache

Notes:

Persistence Context and Object Identity

- The persistence context contains every persistent entity in the scope of the EM
  - **Only one instance** with a given persistent identity can exist within a persistent context
- This means that within the scope of the persistence context, **database identity** (having the same primary key) and **Java identity** (having the same instance) are the same
  - 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
- This improves efficiency, as in the situation mentioned above the second retrieval does NOT have to go to the DB
  - It's one major efficiency advantage of JPA
  - It also simplifies programming, as you always know that within a given scope, you only have one instance for a given DB record

Notes:

Synchronization To the Database

- There can potentially be a lot going on in a JPA program
  - Objects being created and persisted, persistent objects being modified, and so on
  - This can represent a lot of activity in the database, spread out over many operations
  - When do these get propagated to the database?
- JPA uses **write-behind** to synchronize with the database
  - Changes made to persistent objects in the scope of a persistence context are not immediately propagated to the DB
  - This is done to improve efficiency
  - For example, multiple modifications to a single persistent object can be coalesced into a single UPDATE
  - Database transactions can also be kept shorter

Notes:

- The default behavior for a persistence context is to flush to the database when a transaction is committed
  - The persistence provider may also flush to the database whenever it determines it may be necessary
  - For example, it may flush before a query if the results of the query may be changed by pending writes
- JPA, depending on the underlying provider, may also take advantage of JDBC batch updates when executing multiple UPDATE, INSERT, or DELETE statements

Flushing the Entity Manager

- JPA performs database writes when the EM (and its associated persistence context) is synchronized to the DB
  - We call this **flushing** the EM
  - This is done (at the latest) when a transaction commits
- Flushing the EM may happen at a number of points
  - **Before some query executions** (As determined by the provider)
  - When the transaction commits
  - When flushing explicitly via **`EntityManager.flush()`**
    - Useful, for example if also using JDBC directly, and want to force pending EM operations to be flushed to the DB
- The flushing behavior can be changed to one of the following via `EntityManager.setFlushMode()`
  - **`AUTO`**: The default behavior described above
  - **`COMMIT`**: flush only at transaction commit - not before queries

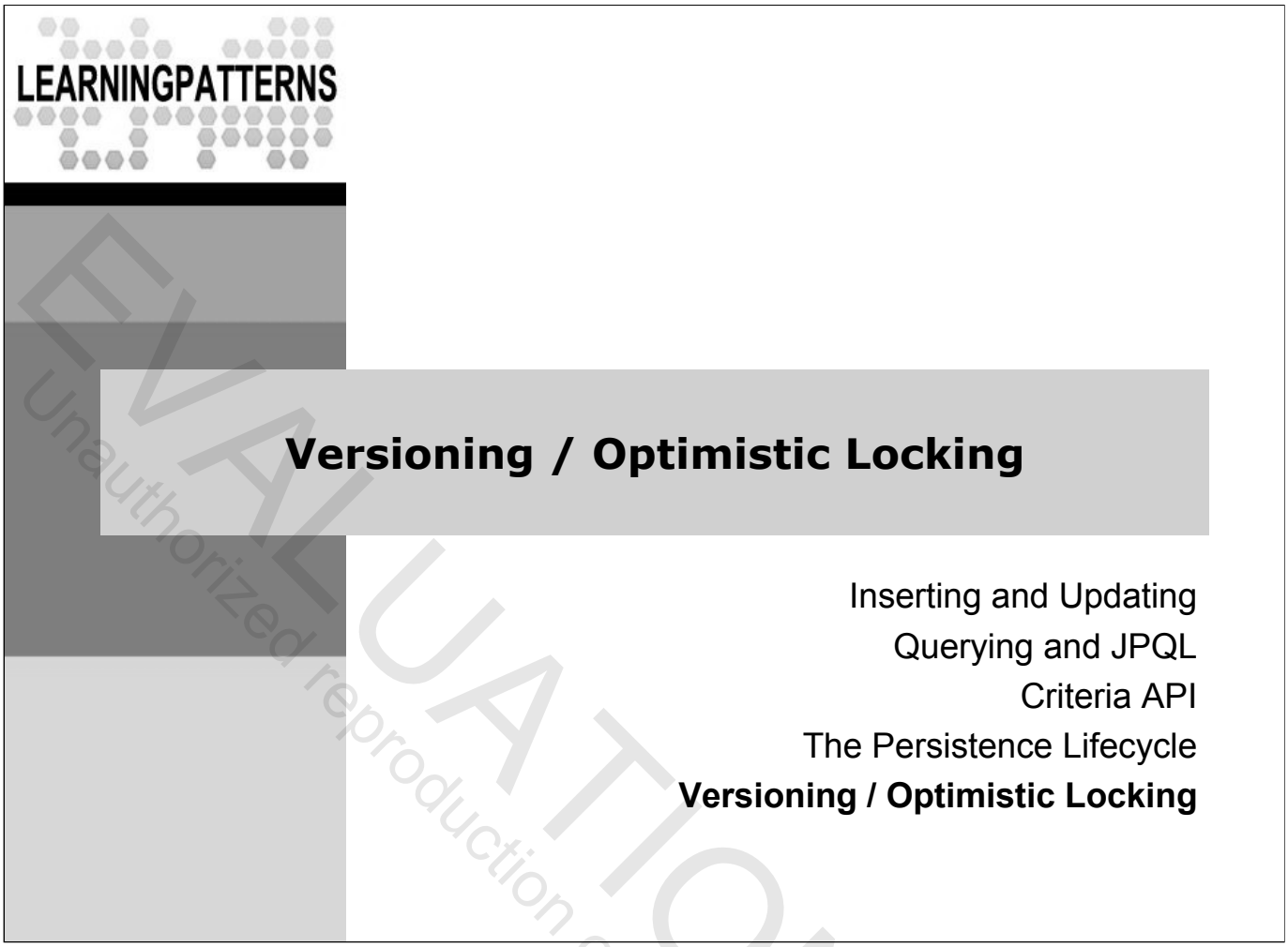
Notes:

- From the JPA 2 spec:
  - "When queries are executed within a transaction, if FlushModeType.AUTO is set on the Query or TypedQuery object, or if the flush mode setting for the persistence context is AUTO (the default) and a flush mode setting has not been specified for the Query or TypedQuery object, the persistence provider is responsible for ensuring that all updates to the state of all entities in the persistence context which could potentially affect the result of the query are visible to the processing of the query. The persistence provider implementation may achieve this by flushing those entities to the database or by some other means.
  - If FlushModeType.COMMIT is set, the effect of updates made to entities in the persistence context upon queries is unspecified.
  - If there is no transaction active, the persistence provider must not flush to the database. "
- Your persistence context will be flushed if a transaction commits, and it is synchronized to the TX – This is always the case:
  - If you are using a container-managed EM
  - If you're using an application managed EM, and use the JPA API to control the tx
- If you're using an application-managed EM, and JTA, then the EM will automatically be synchronized to the TX if the persistence context is created within the TX – if it is created before the TX starts it can be manually synchronized to the TX by calling EntityManager.joinTransaction

Yes, It's Complicated

- There are a lot of moving parts to a JPA application
  - And a lot of things are happening under the hood
  - This is dictated by the nature of the problem, and the goals of JPA
  - If you want a flexible, capable ORM solution that is also efficient, you have to have the mechanisms to support it
- It's important to get an understanding of these lifecycle concepts
  - They'll become clearer as you work with JPA more
  - Don't worry if you didn't catch it all the first time

Notes:



Notes:

Optimistic Locking

- Sometimes it's not practical to have a transaction span a complete business process
  - Especially when there is a human user involved
  - Long running transactions aren't generally a good idea
- One strategy is to use detached objects
  - Read an object, end any transaction used (the object is now detached)
  - Do some work / wait for user input
  - Start a transaction and modify the object
  - Try to save back to the database

Notes:

- Detached entity instances continue to live outside of the persistence context in which they were persisted or retrieved, and their state is no longer guaranteed to be synchronized with the database state
- The application may access the available state of available detached entity instances after the persistence context ends. The available state includes:
  - Any persistent field or property not marked fetch=LAZY
  - Any persistent field or property that was accessed by the application
- If the persistent field or property is an association, the available state of an associated instance may only be safely accessed if the associated instance is available. The available instances include:
  - Any entity instance retrieved using find().
  - Any entity instances retrieved using a query or explicitly requested in a FETCH JOIN clause.
  - Any entity instance for which an instance variable holding non-primary-key persistent state was accessed by the application.
  - Any entity instance that may be reached from another available instance by navigating associations marked fetch=EAGER

Using a Detached Instance

- You can update the persistent context with a detached instance by merging its data back in
  - Using the `EntityManager.merge()` method

  `<T> T merge(T entity)`: Merge the state of the given entity into the current persistence context

  - If the entity is already present in the persistent context, then its state is overwritten with the detached instance's values
  - If the entity is not present in the persistent context, then a new entity is created, and its state is initialized with the detached instance's values
  - The entity **from the persistent context** (which is managed) is returned, not the entity instance you passed in to `merge()`

Notes:

- If there is no entity with the same id in the database, then an exception is thrown

Using a Detached Instance Example

- Here's an example usage of a detached instance & `merge()`

```
// EventServiceBean as before – details not shown ...
public class EventServiceBean {
  public Event findById(Long id) {
    return em.find(Event.class,id);
  }

 public Event updateEvent(Event ev) {
     return em.merge(ev);
  }
}
```

```
// Code fragment using a detached instance
Long id = // You get an id from a user
EventService es = ... // Event service lookup not shown.
Event ev = es.findById(Event.class, id);
// Send to user, and assume user modifies the event's properties
// ... some time passes while user works
// Now merge the changes into the database.
Event mergedEvent = es.updateEvent(ev);
```

Notes:

Using a Detached Instance Example

- You can update the persistent context with a detached instance
  - Using the `EntityManager.merge()` method
  - Below are two code fragments showing some usage
  - Note that merge returns a managed entity – which is a different object from the argument to the merge method (see notes)

```
Long id = // You get an id from a user
Event ev = em.find(Event.class, id);
// Send to user in Web application
// Assume user modifies the event instance
```

```
// Consider a session bean that is later called to merge
// the state of the detached event into a persistent context
public void updateEvent(Event ev) {
  em.merge(ev);
}
```

Notes:

- The merge operation allows for the propagation of state from detached entities onto persistent entities managed by the EntityManager.
- The semantics of the merge operation applied to an entity X are as follows:
  - If X is a detached entity, the state of X is copied onto a pre-existing managed entity instance X' of the same identity or a new managed copy X' of X is created
  - If X is a new entity instance, a new managed entity instance X' is created and the state of X is *copied* into the new managed entity instance X'
  - If X is a removed entity instance, an IllegalArgumentException will be thrown by the merge operation (or the transaction commit will fail)
  - If X is a managed entity, it is ignored by the merge operation, however, the merge operation is cascaded to entities referenced by relationships from X if these relationships have been annotated with the cascade element value cascade=MERGE or cascade=ALL annotation
  - For all entities Y referenced by relationships from X having the cascade element value cascade=MERGE or cascade=ALL, Y is merged recursively as Y'. For all such Y referenced by X, X' is set to reference Y'. (Note that if X is managed then X is the same object as X'z
  - If X is an entity merged to X', with a reference to another entity Y, where cascade=MERGE or cascade=ALL is not specified, then navigation of the same association from X' yields a reference to a managed object Y' with the same persistent identity as Y

Versioning

- Consider what happens if someone has made changes to the DB between when an entity was detached, then merged in
  - You can get inconsistent results, unless you guard against it
- Java Persistence has built in support for optimistic locking
  - If you add a version property to your objects, JPA will automatically manage the version during updates to a row/instance
  - The app doesn't need to deal with version checking
- Version support requires
  - **A property** in the Java class
  - **Metadata** (e.g. annotations) to signify versioning
  - **A column in the DB** that stores the version
- Versioning can work with version number or timestamps
  - Version number is recommended, and we will show that here

Notes:

Version Property in Java Class

- A normal property, usually named version but can be anything
  - Annotated with **`@Version`**
  - Can specify all usual property characteristics (e.g. column name)
  - It's a normal property, except it's used by the provider to maintain versioning information, and it has no set method
  - Below, we assume that there is a column named version in the DB, so all we need is `@Version` on the version property

```
package com.javatunes.schedule;

public class Event {
    // Other properties omitted

    @Version private int version;

    public int getVersion() { return version; }
    // No set method defined since user should never change version
}
```

Notes:

- The Version field or property is used by the persistence provider to perform optimistic locking. It is accessed and/or set by the persistence provider in the course of performing lifecycle operations on the entity instance
  - An entity is automatically enabled for optimistic locking if it has a property or field mapped with a Version mapping.
- An entity may access the state of its version field or property or export a method for use by the application to access the version, but must not modify the version value
  - Only the persistence provider is permitted to set or update the value of the version attribute in the object
- The version attribute is updated by the persistence provider runtime when the object is written to the database
  - All non-relationship fields and properties and all relationships owned by the entity are included in version checks

Optimistic Locking Example

- In the example below, assume you have two event references
  - e: The "original" reference which becomes detached (with ID=2)
  - e2: A new instance that is retrieved for ID=2
- Let's also say that you have three persistent contexts
  - Represented by entity managers em1, em2, em3
  - These are active for the life of each subpart
  - The merge will fail because the version number of e is stale

```
Event e = em1.find(Event.class,new Long(2)); // Get object/row
// Assume persistent context for em1 is closed
```

```
Event e2 = em2.find(Event.class, new Long(2)); // Get same row
e2.setTitle(e2.getTitle()+ "x"); // Modify the row !
```

```
e.setTitle("Bar");  // This is the original (detached) instance
em3.merge(e);  // THIS WILL FAIL !
```


Notes:

- The persistence provider's implementation of the merge operation must examine the version attribute when an entity is being merged and throw an `OptimisticLockException` if it is discovered that the object being merged is a stale copy of the entity—i.e. that the entity has been updated since the entity became detached
  - Depending on the implementation strategy used, it is possible that this exception may not be thrown until flush is called or commit time, whichever happens first
- If only some entities contain version attributes, the persistence provider runtime is required to check those entities for which version attributes have been specified
  - The consistency of the object graph is not guaranteed, but the absence of version attributes on some of the entities will not stop operations from completing

Locking Objects

- In general, Java Persistence doesn't require you to deal with locking – It just "does the right thing"
- If you need to though, `EntityManager.lock()` will explicitly obtain a lock on an object

```
void lock(Object entity,LockModeType lockMode)
```

  – This obtains the specified lock level upon the given object
- **`LockMode.READ`**: A shared lock
  – Prevents dirty read and non-repeatable read
- **`LockMode.WRITE`**:
  – Prevents dirty read and non-repeatable read as above
  – Also forces an update to the version column
  – Basically forces a write

Notes:

- A read lock is generally implemented by the entity manager acquiring a lock on the underlying database row
- Implementations are not required to support this for non-versioned objects, and if it can't support a call, a `PersistenceException` must be thrown