Additional JPA Topics

- Customizing Mapping
- Composite Keys
- The Persistence Context
- Entity States

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Customizing Mapping

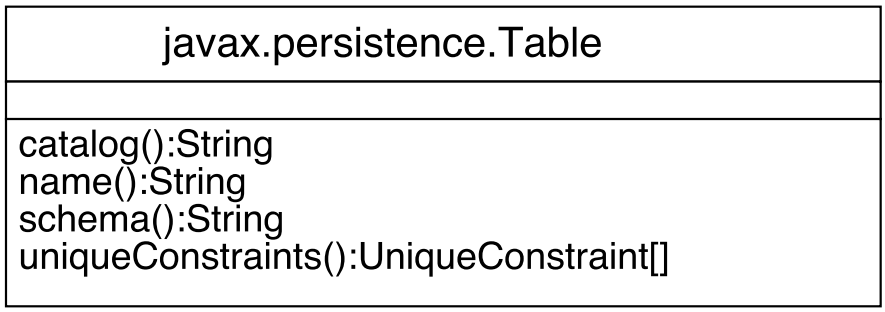
- JPA provides annotations that let you control how JPA maps entities to the database
- These annotations are especially useful in **bottom-up** and **meet in the middle** development scenarios

In a top-down scenario, your JPA provider creates the database tables based on the entities. In bottom-up, you use a tool to generate Java classes from database tables. In meet-in-the-middle, you have existing database tables, but design the object model separately.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Mapping Entities to Tables

- The **@Table** annotation maps an entity to a database table or view




```
1    @Entity
2    @Table(name="INS_POL")
3    public class Policy {}
```

2 - 3

All elements of this annotation are optional.

```
catalog                The database catalog
name                   Table or view name
schema                 The database schema
uniqueConstraints      List of columns that should have UNIQUE constraint
```

Note that the uniqueConstraints element is only useful in top-down scenarios - it tells the JPA provider whether to generate a UNIQUE constraint on the specified columns.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Secondary Tables

- Especially in **meet in the middle** scenarios, you might have the data for a single entity scattered across multiple tables
- JPA provides **@SecondaryTable** and **@SecondaryTables** to map this

javax.persistence.SecondaryTable
catalog():String name():String pkJoinColumns():PrimaryKeyJoinColumn[] schema():String uniqueConstraints():UniqueConstraint[]

The only required element for this annotation is the name, which specifies the secondary table's name in the database.

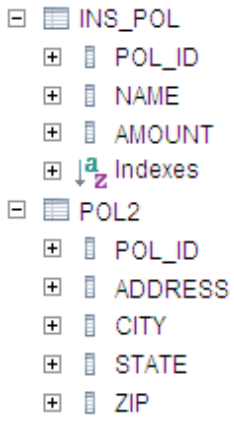
```
catalog                 The database catalog
name                    Table or view name
pkJoinColumns           How to map from primary to secondary table. If the secondary ta
                        has a foreign key column named the same as the primary table's
                        omit this element
schema                  The database schema
uniqueConstraints       List of columns that should have UNIQUE constraint
```

The SecondaryTables annotation is similar, but accepts an array of SecondaryTable annotations.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Secondary Table Example

```
1    @Entity
2    @Table(name = "INS_POL")
3    @SecondaryTable(name = "POL2")
4    public class Policy
5    {
6        @Id
7        @Column(name = "POL_ID")
8        private int policyID;
9        @Column(name = "name")
10       private String policyName;
11   . . .
12
13       @Column(table="POL2")
14       private String address;
15   . . .
16   }
```



Note that the @Column annotation defaults to using the primary table.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Coumpound Primary Keys

- Sometimes you might have existing tables with multiple columns acting as primary key
- JPA provides **@IdClass** and **@EmbeddedId** annotations for that purpose
- Using @IdClass makes writing JPQL a little cleaner

LINEITEM	
PK	ORDERID
PK	ITEMID
	QUANTITY

With @IdClass, you specify @Id on mulitple properties in the entity that comprise the key and use @IdClass on the entity class to reference a separate class that repeats that information. Any JPQL can reference key properties as normal:

```
SELECT li FROM LineItem WHERE
   li.orderId = 1 AND li.itemId = 1
```

With @EmbeddedId, the entity has a single property with the primary-key class type, annotated with @EmbeddedId to reference the separate key class. The primary-key class must itself be annotated with @Embeddable. Then the JPQL must have an additional clause to reference the primary-key's property name:

```
SELECT li FROM LineItem WHERE
   li.key.orderId = 1 AND li.key.itemId = 1
```

The JPA community is divided on which approach is "better" and it often is simply just a personal choice.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Using @IdClass Example

LineItem.java

LineItemKey.java

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

```
1    package domain;
2
3    import javax.persistence.Entity;
4    import javax.persistence.Id;
5    import javax.persistence.IdClass;
6
7    @Entity
8    @IdClass(LineItemKey.class)
9    public class LineItem
10   {
11       @Id
12       private int orderId;
13       @Id
14       private int itemId;
15
16       private int quantity;
17
18       public int getOrderId()
19       {
20           return orderId;
21       }
22
23       public void setOrderId(int orderId)
24       {
25           this.orderId = orderId;
26       }
27
28       public int getItemId()
29       {
30           return itemId;
31       }
32
33       public void setItemId(int itemId)
34       {
35           this.itemId = itemId;
36       }
37
38       public int getQuantity()
39       {
40           return quantity;
41       }
42
43       public void setQuantity(int quantity)
44       {
45           this.quantity = quantity;
```

```
46          }
47
48
49      }
```

```
1    package domain;
2
3    import java.io.Serializable;
4
5    @SuppressWarnings("serial")
6    public class LineItemKey implements Serializable
7    {
8        private int orderId;
9        private int itemId;
10
11       @Override
12       public int hashCode()
13       {
14           final int prime = 31;
15           int result = 1;
16           result = prime * result + itemId;
17           result = prime * result + orderId;
18           return result;
19       }
20
21       @Override
22       public boolean equals(Object obj)
23       {
24           if (this == obj)
25               return true;
26           if (obj == null)
27               return false;
28           if (getClass() != obj.getClass())
29               return false;
30           LineItemKey other = (LineItemKey) obj;
31           if (itemId != other.itemId)
32               return false;
33           if (orderId != other.orderId)
34               return false;
35           return true;
36       }
37
38       public int getOrderId()
39       {
40           return orderId;
41       }
42
43       public int getItemId()
44       {
45           return itemId;
```

```
46          }
47
48      }
```

Views

- You can map JPA entities to **views** rather than tables - just specify the view name in the @Table annotation
- Note that in some databases, views are read-only so updates to entities might not work

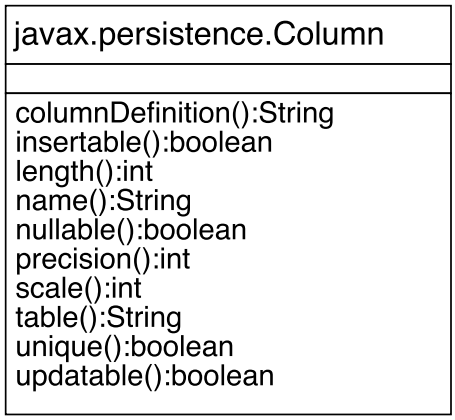
```
1    //CREATE VIEW V_EMP(EMP_ID, NAME) AS
2    //  SELECT EMP_ID, NAME FROM EMPLOYEE
3    @Entity
4    @Table(V_EMP)
5    public class Employee
6    {
7       @Id
8       @Column(EMP_ID)
9       private int employeeId;
10
11      private String name;
12   . . .
13   }
```

According to Wikipedia:

In database theory, a view is the result set of a stored query - or map-and-reduce functions - on the data, which the database users can query just as they would a persistent database collection object. This pre-established query command is kept in the database dictionary. Unlike ordinary base tables in a relational database, a view does not form part of the physical schema: as a result set, it is a virtual table computed or collated from data in the database, dynamically when access to that view is requested. Changes applied to the data in a relevant underlying table are reflected in the data shown in subsequent invocations of the view.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Mapping Properties to Columns




```
1    @Entity
2    public class Policy
3    {
4    . . .
5       @Column(name="name", table="POL2")
6       private String policyName;
7    }
```

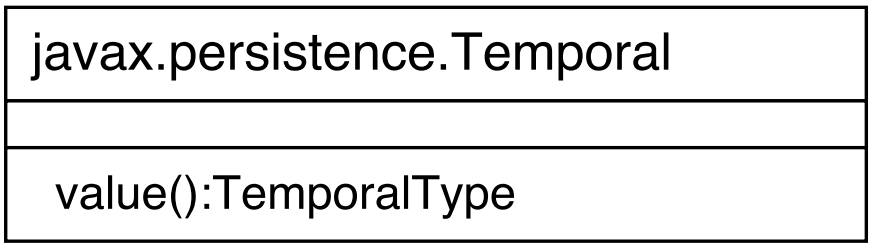

---

columnDefinition	DDL SQL fragment for the column. Useful for top-down or if the column's type is different than what JPA expects
insertable	Boolean (default is true) for whether column should be included in SQL inserts
length	Length of string columns (default 255) for top-down generation
name	Column name
precision	Precision for decimal types, i.e. the number of total digits
scale	Scale for decimal types, i.e. the number of digits after decimal point
table	Column's table (default is entity's table)
unique	Whether property is a unique key (default is false)
updateable	Boolean (default is true) for whether column should be included in SQL updates

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Mapping Dates and Times

- JPA provides the **@Temporal** annotation so you can control how Java dates/times map to the database




```
1    @Entity
2    public class Policy
3    {
4    . . .
5       @Temporal(TemporalType.DATE)
6       private java.util.Date createDate;
7    }
```

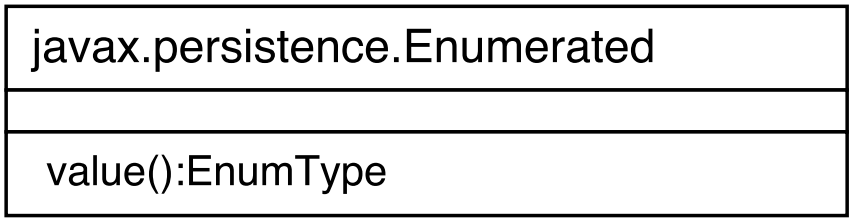
This annotation maps java.util.Calendar and java.util.Date objects to a database column.

`value:  TemporalType.DATE, TemporalType.TIME, TemporalType.TIMESTAMP`

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Mapping Enumerations

- JPA provides the **@Enumerated** annotation to map Java enumerated types to a column




```
1    @Entity
2    public class Policy
3    {
4    . . .
5       enum PolicyType {Life, Auto, Boat}
6
7       @Enumerated(EnumType.ORDINAL)
8       private PolicyType policyType;
9    }
```

This annotation maps either the ordinal value (zero-based) or string value, e.g. "Boat", of an enumerated type property to a database column.

`value:  EnumType.ORDINAL (default), EnumType.STRING`

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Mapping LOBs

- JPA provides the **@Lob** annotation to map large object to a column
- Strings and other character-based types map to CLOB, all others map to BLOB

```
1    @Entity
2    public class Policy
3    {
4    . . .
5      @Lob
6      private byte[] scannedSignature;
7    }
```


This annotation has no elements (arguments). It is often used in conjunction with @Basic(fetch=LAZY).

To work with an image as shown above, you might provide a "get" method like:

```
public Image getScannedSignature()
{
    Image result = null;

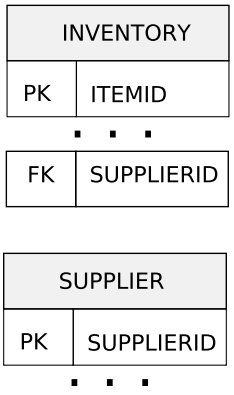
    if (scannedSignature != null && scannedSignature.length > 0)
    {
        result = new ImageIcon(scannedSignature).getImage();
    }
    return result;
}
```

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Mapping Relationships with Foreign Keys

- When you use JPA to map relationships, you can use **@JoinColumn** or **@JoinColumns** to control mapping

javax.persistence.JoinColumn
columnDefinition():String insertable():boolean name():String nullable():boolean referencedColumnName():String table():String unique():boolean updatable():boolean




```
1    @Entity
2    public class Inventory
3    {
4    . . .
5     @ManyToOne
6     @JoinColumn(name = "SUPPLIERID")
7     private Supplier theSupplier;
8    }
```

You put the @JoinColumn annotation in the OWNING entity, i.e. the entity whose table contains the foreign key. You don't need to do anything in the inverse.

You use @JoinColumns if you are using compound keys. Here are the elements for @JoinColumn:

```
columnDefinition     SQL fragment for top-down generation
insertable           Boolean (default is true) for whether
                     column should be included in SQL inserts
length               Length of string columns (default 255)
                     for top-down generation
name                 Column name
nullable             Whether foreign key column is nullable
referencedColumnName Column name in related table.
                     Default is that foreign-key and referenced
                     column names match
table                The table that contains the column
unique               Whether the property is a unique key
updateable           Boolean (default is true) for whether
                     column should be included in SQL updates
```

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Mapping Relationships with Join Tables

- When you use JPA to map relationships that use **join tables**, you can use **@JoinTable** to control mapping

javax.persistence.JoinTable
catalog():String inverseJoinColumns():JoinColumn[] joinColumns():JoinColumn[] name():String schema():String uniqueConstraints():UniqueConstraint[]

MEM_ASSOC	
FK	AID
FK	LID

```
1    @Entity
2    public class MembershipList
3    {
4    . . .
5     @ManyToMany
6     @JoinTable(name="MEM_ASSOC",
7        joinColumns=@JoinColumn(name="LID"),
8        inverseJoinColumns=@JoinColumn(name="AID"))
9     private Collection<Associate> associateList;
10   }
2 - 14
```

You put the @JoinTable annotation in the OWNING entity. You don't need to do anything in the inverse. Here are the elements for @JoinTable:

```
catalog                 The database catalog
inverseJoinColumns      @JoinColumn(s) of foreign-key for non-owning entity
joinColumns             @JoinColumn(s) of foreign-key of owning entity
name                    Table or view name
schema                  The database schema
uniqueConstraints       List of columns that should have UNIQUE constraint
```

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Specifying OrderBy

- When you use JPA to map a multi-valued relationship, you can use **@OrderBy** to control the order of the objects in the collection
- You can use **ASC** (default) or **DESC** for ascending or descending

```
1    @Entity
2    public class Associate
3    {
4    . . .
5
6        @OneToMany
7        @OrderBy("amount DESC")
8        private Collection<Dues> duesList;
9    }
```

This annotation has no elements (parameters).

The value you specify to order by must be a property in the related entity.

If you do not specify @OrderBy, by default, JPA orders related entities by primary key.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Using Java Maps

- JPA lets you configure a Java **Map** collection to implement a one-to-many or many-to-many relationship
- A very common case is to use the related entity's primary key as the map key and use **@MapKey** annotation

```
1    @Entity
2    public class Associate
3    {
4    . . .
5        @OneToMany
6        @MapKey(name = "duesID")
7        private Map<Integer, Dues> duesMap;
8    }
```

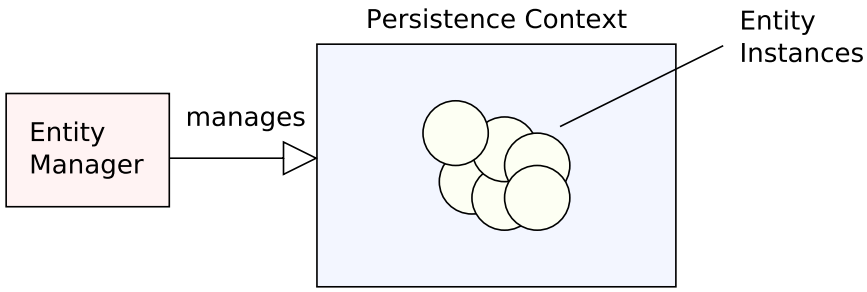
JPA does not limit you to using entities as the value in a Map and provides the @ElementCollection and @CollectionTable annotations for the case when the keys/values are not entities.

The @MapKey annotation has a single element, which specifies the name of the field that will act as the key. Using the related entity's primary key is common, but it's not required. If there's some other unique property, you can use that instead.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The Persistence Context

- The EntityManager maintains a **persistence context** that acts as a cache of managed entity instances
- JPA provides both **application managed** and **container managed** entity managers
- You can use either in a JEE application, but in JSE, only application managed



JPA also defines the concept of a "persistence unit", which is a named set of classes for entities. You typically define the persistence unit in XML and reference its name when you create the entity manager factory.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Application Managed Persistence Context

- With an application-managed entity manager, the application must explicitly create the EntityManager from an EntityManagerFactory

```
1    EntityManagerFactory emf = Persistence
2          .createEntityManagerFactory("myunit");
3
4    EntityManager em = emf.createEntityManager();
5
6    . . .
7
8    em.close();
9    emf.close();
```


Java SE applications MUST use this kind of persistence context. Java EE applications CAN use this type, but can also "container managed" persistence contexts.

For this approach, the persistence.xml must declare that the persistence-unit uses the RESOURCE_LOCAL transaction-type.

The EntityManagerFactory is a lightweight object, but the EntityManager (and its associated persistence context) can occupy a lot of storage and is not thread-safe.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Container Managed Persistence Context

- For applications that run on a full JEE application server, you can use **@PersistenceContext** or **@PersistenceUnit** to obtain a reference to an EntityManager or EntityManagerFactory

```
1    @Stateless
2    public class AutoSessionBean implements ManageAutos
3    {
4        @PersistenceContext(unitName="myunit")
5        private EntityManager entityManager;
6    . . .
7    }
```

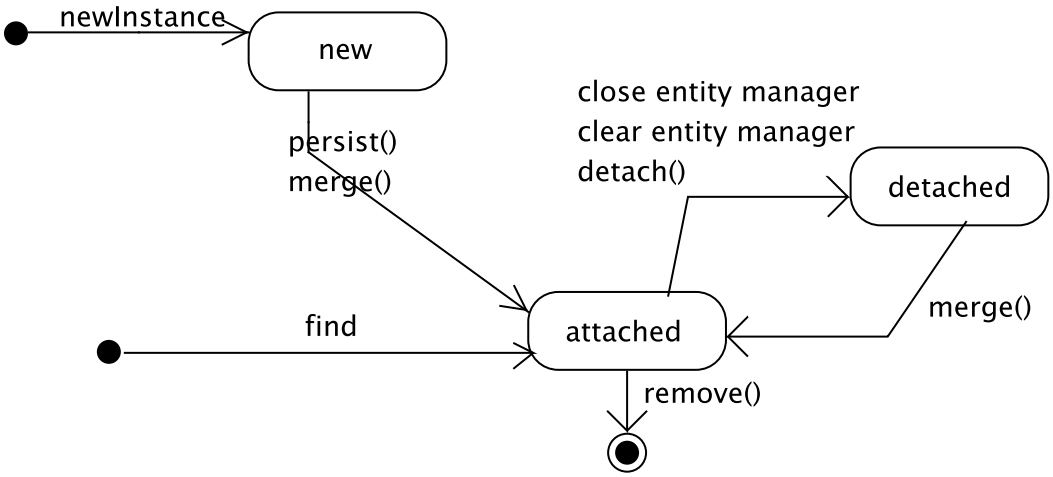
If you use @PersistenceContext to inject an EntityManager reference, the persistence.xml file must specify transaction-type as JTA. If you use @PersistenceUnit to inject an EntityManagerFactory, transaction-type must be RESOURCE_LOCAL.

Note that applications that run on a JEE app server also can choose between so-called "transaction-scoped" and "extended" persistence contexts. The basic rule is that if the application uses stateful session EJBs, it can use extended, but all others use the default transaction-scoped. See the EJB specification for details.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Entity States

- **Detached (unmanaged)** – The object is not known by the entity manager (i.e. not in a **persistence context**)
- **Attached (managed)** – The entity manager tracks the object's state and flushes it to the database at the end of a transaction



When you first create an entity using the Java "new" operator, it exists only in memory and is not under the control of the Java Persistence entity manager.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Persist vs Merge

- The EntityManager supports **persist()** and **merge()** methods and there is some overlap in function:
  - persist() is geared toward creating new entities
  - merge() *can* attach new entities, but is geared toward integrating detached entities back into the persistence context

```
1    MyEntity me = new MyEntity();
2    entityManager.persist(me);
3
4    . . .
5
6    entityManager.detach(me);
7
8    // change "me"
9
10   entityManager.merge(me);
```

See http://spitballer.blogspot.com/2010/04/jpa-persisting-vs-merging-entites.html for details.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Optimistic Concurrency and Versions

- JPA assumes **optimistic concurrency**: applications will handle concurrency issues
- In other words, by default, there's no locking at the database level, and if two transactions modify the same entity, the last to commit "wins"
- JPA supports **version columns** in the database to help detect conflicts

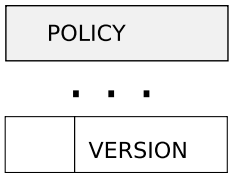
Instead of using optimistic concurrency, another approach is to write code to configure an EntityManager to use pessimistic concurrency – for example, specify LockModeType.PESSISTIC_WRITE when you query entities.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Using a Version Column

- JPA provides the **@Version** annotation to configure the property/column in which JPA will store version information
- Application should not themselves modify this property
- At the start of a transaction, JPA will set the version property and will throw the OptimisticLockException at the end of the transaction if some other transaction modified it

```
1    @Entity
2    public class Policy
3    {
4    . . .
5        @Version
6        private int version;
7    }
```



2 - 23

The @Version annotation has no elements. Note that the JPA specification states that the version column must in the entity's primary table or else the code is not portable.

The version column can be typed to short, int, long or java.sql.Timestamp. Note that you don't need to update the version column yourself; the JPA provider mangages and checks it automatically. You do need to ensure that it has an initial value, e.g. one.

Some JPA providers let you configure so-called "field locking" if your database table does not or cannot contain a version column, but this is non-standard.

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Chapter Summary

In this chapter, you learned:

- More about mapping JPA entities to database columns
- About the persistence context and entity states

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Copyright © Descriptor Systems, 2001-2014. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum