

10

Applying the JPA

ORACLE

Copyright © 2012, Oracle and/or its affiliates. All rights reserved.

Adolfo De-la-Rosa (adolfodelarosa2012@gmail.com) has a non-transferable license to use this Student Guide.

Objectives

After completing this lesson, you should be able to:

- Identify relationships in an application
- Build and deploy a JPA application in a Java SE environment
- Apply a two-tier design in the HenleyApp application



ORACLE

Copyright © 2012, Oracle and/or its affiliates. All rights reserved.

Topics

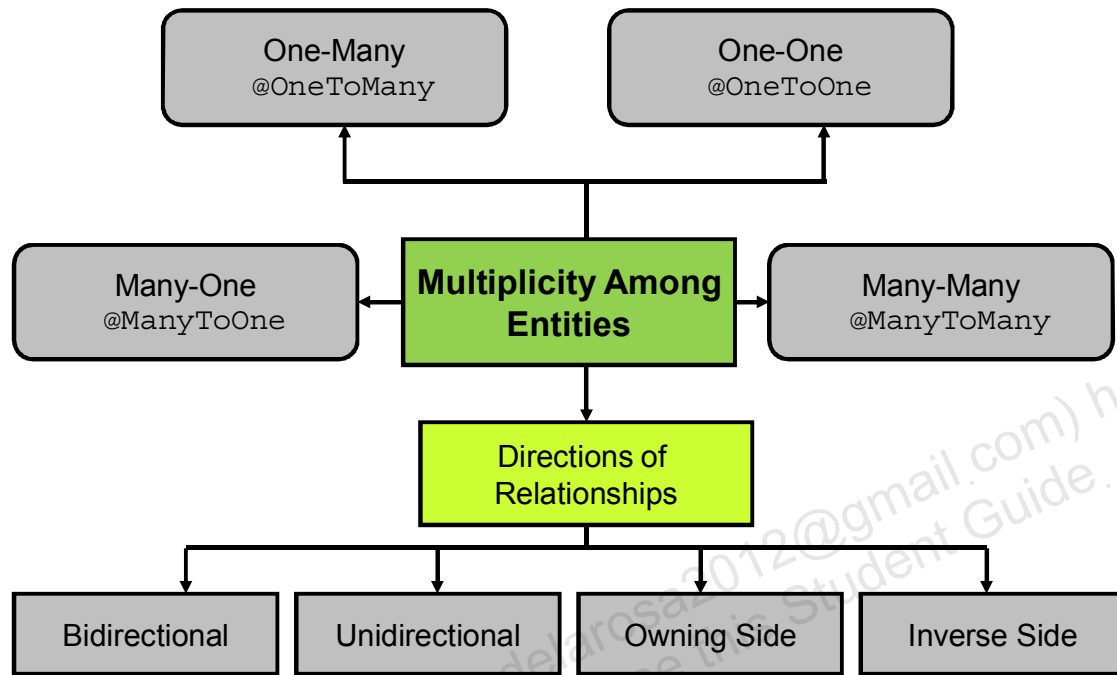
- Entity relationships
- The Criteria API
- Applying the JPA in the HenleyApp two-tier application



ORACLE

Copyright © 2012, Oracle and/or its affiliates. All rights reserved.

Entity Relationships



ORACLE

Copyright © 2012, Oracle and/or its affiliates. All rights reserved.

Entities must have relationships with other entities, which helps to develop the domain model for an enterprise application.

Relationship mapping can be:

One-to-many: An entity instance can be related to multiple instances of the other entities. One-to-many relationships use the `javax.persistence.OneToOne` annotation on the corresponding persistent property or field.

Many-to-one: This multiplicity is the opposite of a one-to-many relationship. Many-to-one relationships use the `javax.persistence.ManyToOne` annotation on the corresponding persistent property or field.

Many-to-many: The entity instances can be related to multiple instances of each other. Many-to-many relationships use the `javax.persistence.ManyToMany` annotation on the corresponding persistent property or field.

Directions of a relationship is bidirectional or unidirectional.

- Bidirectional relationships:
 - Must be managed by the application
 - Have an owning side and an inverse side
- The owning side controls the database write.
- The inverse side must specify the owning side by using the `mappedBy` element.

Relationship Direction

Unidirectional	Bidirectional
Only one entity has a pointer to the other.	Both entities point to one another.
The relationship has only an owning side.	The relationship has both an owning side and an inverse side.
<ul style="list-style-type: none">• The owning side of a relationship determines how the Persistence runtime makes updates to the relationship in the database.• The owning side controls the database write.	

ORACLE

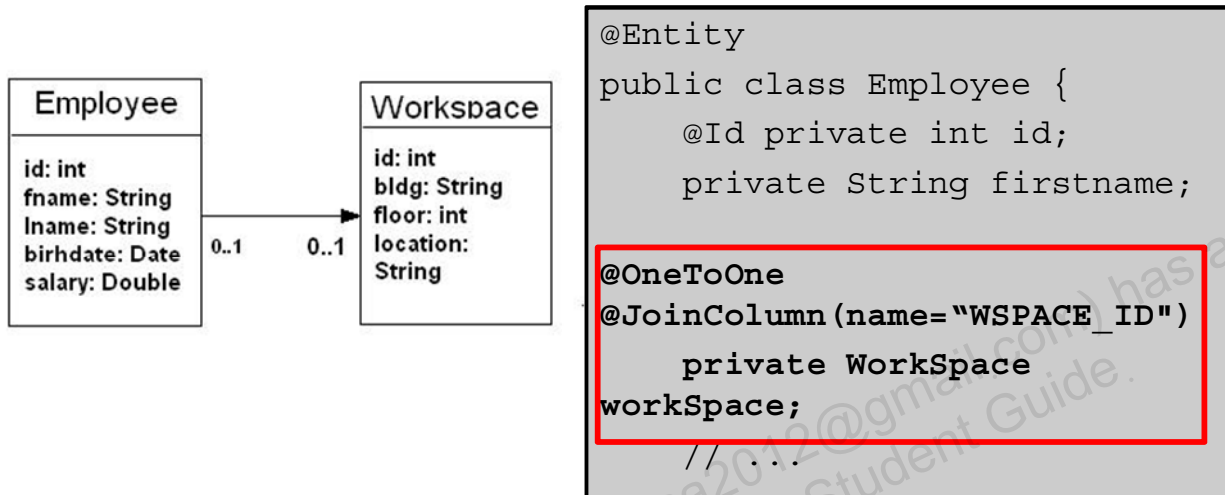
Copyright © 2012, Oracle and/or its affiliates. All rights reserved.

Identifying the relationship direction is usually a data modeling decision, not a Java programming decision, and it would likely be decided based on the most frequent direction of traversal.

The subsequent slides have examples.

Unidirectional One-to-One Relationships

Each entity instance is related to a single instance of another entity.



ORACLE

Copyright © 2012, Oracle and/or its affiliates. All rights reserved.

Example of a one-to-one association would be an employee who has a workspace. Every employee gets assigned his or her own workspace; therefore, a one-to-one relationship from Employee to WorkSpace can be created as shown in the figure in the slide.

0..1 indicates the range.

In the Employee and Workspace table mapping, The foreign key column in the EMPLOYEE table will have WSPACE_ID that refers to the WORKSPACE table. However, since the direction is unidirectional, the Workspace entity will not have an Employee field or property in it. Employee knows about Workspace, but Workspace does not know which Lineltem instances refer to it.

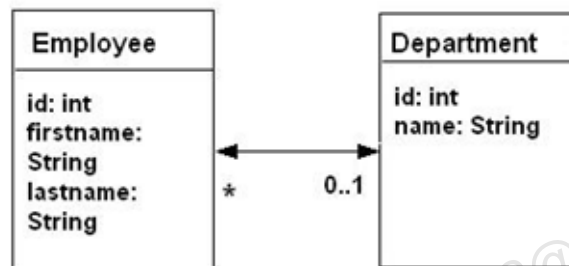
One-to-one relationships use the javax.persistence.OneToOne annotation on the corresponding persistent property or field. The code snippet in the slide uses the @OneToOne and @JoinColumn annotations.

This is a unidirectional one-to-one mapping. The entity table that contains the join column determines the entity that is the owner of the relationship. The other entity table (that is, Workspace) does not need a join column at all.

However, in one-to-one bidirectional relationships, the owning side corresponds to the side that contains the corresponding foreign key.

Many-to-One and One-to-Many Relationships

- In many-to-one, multiple instances of an entity can be related to a single instance of the other entity.
- In one-to-many, an entity instance can be related to multiple instances of the other entities.



ORACLE

Copyright © 2012, Oracle and/or its affiliates. All rights reserved.

The Employee and Department example shows a many-to-one mapping from Employee to Department and a one-to-many mapping from Department to Employee. They are equivalent, because bidirectional many-to-one relationships imply a one-to-many mapping back from the target to source, and vice versa.

Because Department knows which Employee instances it has and Employee knows which Department it belongs to, they have a bidirectional relationship.

Employee and Department Entities

```
@Entity public class Employee {
    @Id protected Integer id;
    @ManyToOne -----> Owning side
    @JoinColumn (name="DEPT_ID")
    protected Department dept;
}

@Entity public class Department {
    @Id private Integer id;
    @OneToMany(mappedBy="dept") -----> Inverse side has mappedBy attribute
    private Collection<Employee> emps;
}
```

ORACLE

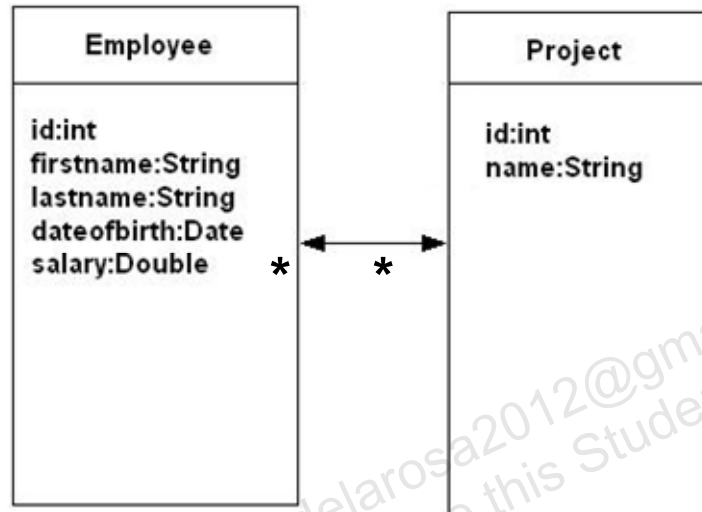
Copyright © 2012, Oracle and/or its affiliates. All rights reserved.

In the code snippet in the slide, the many-to-one relationship between Employee and Department entities are shown.

- The many side is always the owning side of the relationship. Here Employee is the owning side. The JoinColumn is specified at the owning side.
- The inverse side of a bidirectional relationship must refer to its owning side by using the mappedBy element. In the code snippet in the slide, the @OneToMany annotation provides the mappedBy element in the Department class. Department is the inverse side.
- The mappedBy element designates the property or field in the entity that is the owner of the relationship. In the code snippet, mappedBy refers to "dept," which is the Department field declared in the Employee class.
- The many side of many-to-one bidirectional relationships must **not** define the mappedBy element. In the Employee class, @ManyToOne annotation does not have the mappedBy element.
- A @JoiningColumn cannot be defined in this case. A separate join table has to be created in such a scenario.

Bidirectional Many-to-Many Relationships

When two entities are associated with each other by means of collections



ORACLE


Copyright © 2012, Oracle and/or its affiliates. All rights reserved.

When one or more entities are associated with a collection of their entities, we must model it as a many-to-many relationship. Each of the entities on each side of the relationship has a collection-valued association that contains entities of the target type. The figure in the slide shows a many-to-many relationship between Employee and Project.

Each employee can work on multiple projects, and each project can be worked on by multiple employees. This is a bidirectional many-to-many relationship.

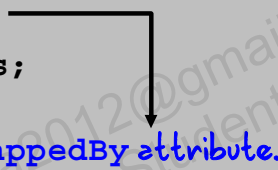
Employee and Project Entities

```
@Entity public class Employee {  
    @Id protected Integer id;  
    @ManyToOne protected Department dept;  
    @ManyToMany protected Collection<Project> projects;  
}
```



A horizontal arrow points from the `@ManyToMany` annotation in the `Employee` class to the text "Owning side".

```
@Entity public class Project {  
    @Id private Integer id;  
    @ManyToMany(mappedBy="projects")  
    private Collection<Employee> emps;  
}
```



A horizontal arrow points from the `@ManyToMany(mappedBy="projects")` annotation in the `Project` class to the text "Inverse side has a mappedBy attribute.".

Inverse side has a mappedBy attribute.

ORACLE

Copyright © 2012, Oracle and/or its affiliates. All rights reserved.

In the code snippet in the slide, two entities are considered: Employee and Projects.

Both side have `@ManyToMany` mappings. There is no `@JoinColumn` in this relationship. As it is a bidirectional relationship, you have to choose an owning side and an inverse side.

In this example, Employee is chosen as the owning side. Project is the inverse side and has the `mappedBy` element.

The only way to implement a many-to-many relationship is with a separate join table.

Quiz

State whether the following statements are true:

- a. For many-to-many bidirectional relationships, either side may be the owning side.
- b. The owning side of a bidirectional relationship controls the database write.
- c. For many-to-many bidirectional relationships, there is no need to have an owning side.
- d. The many side of a bidirectional relationship is always the owning side of the relationship.

ORACLE

Copyright © 2012, Oracle and/or its affiliates. All rights reserved.

Answer: a, b, d

Topics

- Entity relationships
- **The Criteria API**
- Applying the JPA in the HenleyApp two-tier application



ORACLE

Copyright © 2012, Oracle and/or its affiliates. All rights reserved.

The Criteria API

- The Criteria API, an alternative method for constructing queries, uses a Java programming language API instead of JP QL or native SQL.
- Criteria API allows generics and thus removes the need for casting.
- Criteria queries set the SELECT, FROM, and WHERE clauses by using Java programming language objects, so the query can be created in a typesafe manner.
- Criteria API :
 - Standardizes many of the programming features that exist in proprietary persistence products
 - Applies programming best practices of the proprietary models
 - Makes full use of the Java programming language features

ORACLE

Copyright © 2012, Oracle and/or its affiliates. All rights reserved.

The Criteria API and JPQL are closely related and are designed to allow similar operations in their queries. Developers familiar with JPQL syntax will find equivalent object-level operations in the Criteria API. The Criteria API was introduced in JPA 2.0.

In spite of JP QL, programming APIs are still used to enable features not yet supported by the standard query language. The Criteria API allows you to find, modify, and delete persistent entities by invoking Java Persistence API entity operations.

Steps to Create a Criteria Query

1. Use an `EntityManager` instance to create a `CriteriaBuilder` object.
2. Create a query object by creating an instance of the `CriteriaQuery` interface.
3. Set the query root by calling the `FROM` method on the `CriteriaQuery` object.
4. Specify what the type of the query result will be by calling the `SELECT` method of the `CriteriaQuery` object.
5. Prepare the query for execution by creating a `TypedQuery<T>` instance, specifying the type of the query result.
6. Execute the query by calling the `getResultList` method on the `TypedQuery<T>` object.

ORACLE

Copyright © 2012, Oracle and/or its affiliates. All rights reserved.

For a particular `CriteriaQuery` object, the root entity of the query, from which all navigation originates, is called the *query root*. It is similar to the `FROM` clause in a JPQL query.

A JP QL Query Versus a Criteria Query

JP QL	Query Using Criteria API
<pre>SELECT emp FROM Employee emp WHERE emp.firstname = 'John'</pre>	<pre>1 CriteriaBuilder crbl = em.getCriteriaBuilder(); 2 CriteriaQuery<Employee> cq = crbl.createQuery(Employee.class); 3 Root<Employee> emp = cq.from(Employee.class); 4 cq.select(emp) .where(crbl.equal(emp.get("firstname"), "John")); 5 TypedQuery<Employee> tq = em.createQuery(cq); 6 List<Employee> allemps = tq.getResultList();</pre>

ORACLE

Copyright © 2012, Oracle and/or its affiliates. All rights reserved.

The query in the slide, written using the Criteria API, can be explained as follows:

1. To create a CriteriaBuilder instance, call the `getCriteriaBuilder` method on the EntityManager instance: `CriteriaBuilder cb = em.getCriteriaBuilder();`
2. The query object is created by using the CriteriaBuilder instance:
`CriteriaQuery<emp> cq = cb.createQuery(emp.class);`
The query will return instances of the entity, so the type of the query is specified when the `CriteriaQuery` object is created to create a typesafe query.
3. The FROM clause of the query is set, and the root of the query specified, by calling the `from` method of the query object: `Root<emp> emp = cq.from(emp.class);`
4. The SELECT clause of the query is set by calling the `select` method of the query object and passing in the query root: `cq.select(emp);`
5. The query object is now used to create a `TypedQuery<T>` object that can be executed against the data source. The modifications to the query object are captured to create a ready-to-execute query: `TypedQuery<emp> tq = em.createQuery(cq);`
6. This typed query object is executed by calling its `getResultList` method, because this query will return multiple entity instances. The results are stored in a `List<emp>` collection-valued object: `List<emp> allemps = tq.getResultList();`

Using the Criteria API in the HenleyApp

```
public List<T> findAll() {  
    javax.persistence.criteria.CriteriaQuery cq =  
    getEntityManager().getCriteriaBuilder().createQuery();  
    cq.select(cq.from(entityClass));  
    return  
    getEntityManager().createQuery(cq).getResultList();  
}  
  
public List<T> findRange(int[] range) {  
    javax.persistence.criteria.CriteriaQuery cq =  
    getEntityManager().getCriteriaBuilder().createQuery();  
    cq.select(cq.from(entityClass));  
    javax.persistence.Query q =  
    getEntityManager().createQuery(cq);  
    q.setMaxResults(range[1] - range[0]);  
    q.setFirstResult(range[0]);  
    return q.getResultList();  
}
```

ORACLE

Copyright © 2012, Oracle and/or its affiliates. All rights reserved.

The slide shows some examples of using Criteria queries in the HenleyApp application.

Quiz

Which of the following statements are true about Criteria API?

- a. The Criteria API uses the Java programming language API.
- b. The Criteria API does not allow the use of generics.
- c. Criteria queries set the SELECT, FROM, and WHERE clauses by using Java programming language objects, so the query can be created in a typesafe manner.

ORACLE

Copyright © 2012, Oracle and/or its affiliates. All rights reserved.

Answer: a, c

Quiz

Identify the classes of the Criteria API from the following list:

- a. CriteriaBuilder
- b. CriteriaQuery
- c. EntityManager
- d. NamedQuery

ORACLE

Copyright © 2012, Oracle and/or its affiliates. All rights reserved.

Answer: a, b

Topics

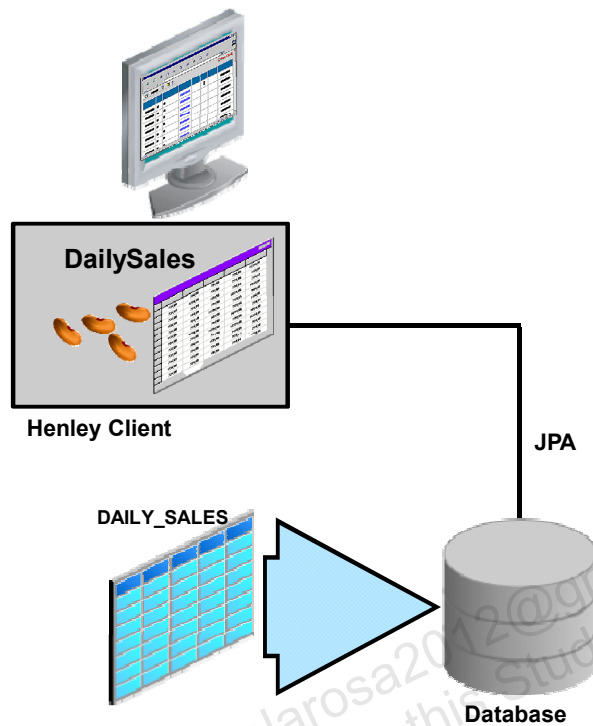
- Entity relationships
- The Criteria API
- Applying the JPA in the HenleyApp two-tier application



ORACLE

Copyright © 2012, Oracle and/or its affiliates. All rights reserved.

HenleyApp: Two-Tier Architecture



ORACLE

Copyright © 2012, Oracle and/or its affiliates. All rights reserved.

The Henley client contains the front-end components and data-access components to manage data that is stored in the back-end Java DB database server. The front-end components use Java and JavaFX APIs to design user interface screens. The data access components use JPA to perform the database operations. The database server used for this application is Java DB.

The development environment or IDE used is NetBeans IDE.

Using the JPA in the HenleyApp Two-Tier Application

1. Download the required persistent provider implementation.
2. Configure NetBeans IDE for using the JPA for development.
3. Figure out the entity relationships in the HenleyApp application.
4. Examine methodical usage of EntityManagerFactory and EntityManager instances in the HenleyApp application.
5. Examine the usage of named queries and the Criteria API in the HenleyApp application.
6. Package the HenleyApp JPA application.
7. Examine the information that goes to the persistence.xml file of HenleyApp.
8. Examine how DAO pattern can be applied while using JPA in HenleyApp.

ORACLE

Copyright © 2012, Oracle and/or its affiliates. All rights reserved.

This slide shows what will be covered in the subsequent slides. The subsequent slides show how JPA has been applied to the HenleyApp two-tier application.

Using a Persistent Provider Implementation

- Persistence provider implementations may be optimized for specific database vendors.
- One important reason for using the JPA is to decouple your application from the underlying database technology.
- After downloading and installing the reference implementation, you must configure your development environment to use these files in your project. The goals for configuration are to:
 - Include the reference implementation in the Java language compiler classpath
 - Include the reference implementation in the Java runtime environment (JRE) classpath

ORACLE

Copyright © 2012, Oracle and/or its affiliates. All rights reserved.

You can download the Java Persistence API reference implementation from the GlassFish project. Although originally part of the enterprise application server, the reference implementation works well in desktop applications too.

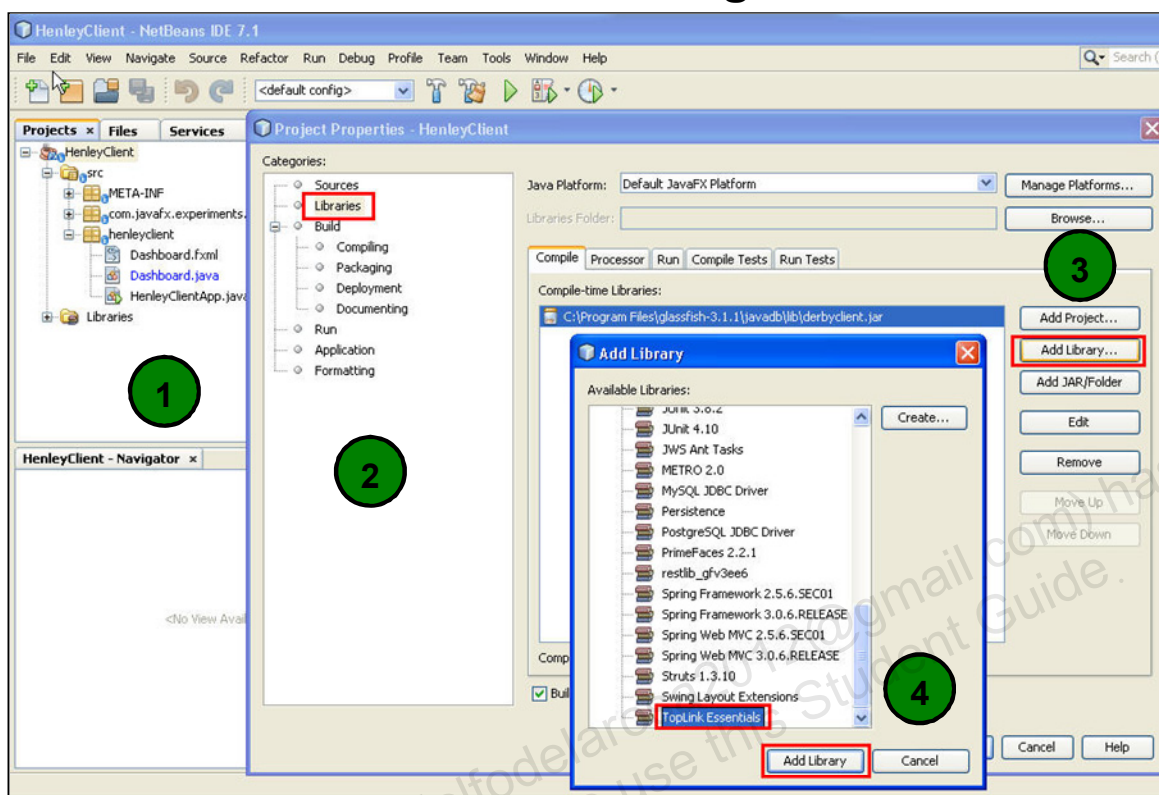
You can easily use and package the reference implementation in your desktop applications. Using the reference implementation, you can simplify your desktop applications to use simple POJOs for both application logic and persistence. Adding the API to your development environment usually involves just including the provider's JAR files in your compiler and runtime environment classpaths. An IDE can help you automate the process of packaging an API implementation with your application.

You should avoid using proprietary extensions. Avoiding proprietary extensions ensures that you can use the widest range of database technologies now and in the future. You should contact your provider vendor for information about what databases they support.

GlassFish is an industrial-strength, open-source application server. Being the reference implementation of the Java EE 6 standard, GlassFish also provides a Java Persistence API implementation. You do not have to download the entire GlassFish product to get the API implementation. The filename is `glassfish-persistence-installer-v2-b46.jar`, but you should expect slightly different filenames as the version changes.

[TopLink Essentials JPA implementation](#) is the reference implementation provided by GlassFish.

NetBeans IDE Configuration



ORACLE

Copyright © 2012, Oracle and/or its affiliates. All rights reserved.

The simplest way to install and configure a JPA reference implementation is to use an integrated development environment (IDE). NetBeans or Eclipse are both examples of popular IDEs that will help you include the provider implementation JAR files in your compiler and JRE classpaths.

In NetBeans:

- Add the persistent provider, which is EclipseLink (JPA 2.0) to your project from the properties window, as shown in the slide.
- In order to use the Java DB database client, you must add the file `derbyclient.jar` to the project. `derbyclient.jar` is available in your Java installation location (for example, `D:\Program Files\Java\jdk1.7.0\db\lib`). This JAR file can be added from the project properties window.

Entity Relationships in the HenleyApp

```
@Entity
@Table(name = "DAILY_SALES", catalog = "", schema
= "HENLEY")
.....
public class DailySales implements Serializable {
....
@JoinColumn(name = "REGION_ID",
referencedColumnName = "REGION_ID")
    @ManyToOne
    private Region regionId;

@JoinColumn(name = "PRODUCT_ID",
referencedColumnName = "PRODUCT_ID")
    @ManyToOne
    private Product productId;
.....
```

ORACLE

Copyright © 2012, Oracle and/or its affiliates. All rights reserved.

Having identified the entities of HenleyApp in the previous lesson, in this and the next two slides, we will identify the entity relationships.

Consider the following entities: DailySales, Product, Region.

In NetBeans, examine `DailySales.java`, `Product.java`, and `Region.java` of the `HenleyApp_2Tier` project.

Entity Relationships in the HenleyApp

```
@Entity
@Table(name = "PRODUCT", catalog = "", schema =
"HENLEY")

.....
public class Product implements Serializable {
    @OneToMany(mappedBy = "productId")
    private Collection<DailySales>
    dailySalesCollection;
    @JoinColumn(name = "PRODUCT_ID",
referencedColumnName = "PRODUCT_ID")
    @ManyToOne
    private Product productId;
    .....
}
```

ORACLE

Copyright © 2012, Oracle and/or its affiliates. All rights reserved.

The code snippet in the slide from `Product.java` shows the entity relationship between `Product` and `DailySales`.

Entity Relationships in the HenleyApp

```
@Entity
@Table(name = "REGION", catalog = "", schema =
"HENLEY")
.....
public class Region implements Serializable {
....
    @OneToMany(mappedBy = "regionId")
    private
    Collection<DailySales>dailySalesCollection;
.....
```

ORACLE

Copyright © 2012, Oracle and/or its affiliates. All rights reserved.

The code snippet in the slide from `Region.java` shows the entity relationship with `DailySales`.

Using the API

How to Use the API

- To work with entities, you must use the `javax.persistence` package.
- To work with an entity, you need an `EntityManager` instance.
- To create an `EntityManager` object, you need an `EntityManagerFactory` instance.
- To get the factory, you must use the `Persistence` class.
- To create queries and transactions in a desktop environment, you can use the `EntityManager` object.

Code Example

```
// declaring instances
EntityManagerFactory emf;
EntityManager em;

//creating objects
emf =
Persistence.createEntityManagerF
actory("henley");
em = emf.createEntityManager();

//creating transactions
em.getTransaction().begin();
em.persist (dailysls);
em.getTransaction().commit();
```

ORACLE

Copyright © 2012, Oracle and/or its affiliates. All rights reserved.

In this slide, you identify the packages and classes of JPA, which you will commonly use.

- The `javax.persistence` package is one of the important packages of the JPA .
- One of the first things you need is an `EntityManager` instance.
- An `EntityManager` provides methods to begin and end transactions, to persist and find entities in the persistence context, and to merge or even remove those entities.
- Additionally, an `EntityManager` instance can create and execute queries.
- The `Persistence` class is the bootstrap class used in Java SE environments.

Defining a Query in the HenleyApp

```
@Entity
@Table(name = "CUSTOMER", catalog = "", schema =
"HENLEY")
@XmlRootElement
@NamedQueries({
    @NamedQuery(name = "Customer.findAll", query =
"SELECT c FROM Customer c"),
    @NamedQuery(name = "Customer.findByCustomerId",
query = "SELECT c FROM Customer c WHERE c.customerId =
:customerId"),
    @NamedQuery(name = "Customer.findByFirstName", query
= "SELECT c FROM Customer c WHERE c.firstName =
:firstName"),
    @NamedQuery(name = "Customer.findByLastName", query
= "SELECT c FROM Customer c WHERE c.lastName =
:lastName") })
public class Customer implements Serializable {
```

ORACLE

Copyright © 2012, Oracle and/or its affiliates. All rights reserved.

The name of the query must be unique within the entire persistence unit. A common practice is to prefix the query name with the entity name. For example, the "findAll" query for the Customer entity would be named "Customer.findAll".

The Criteria API in the HenleyApp

```
public List<T> findAll() {
    javax.persistence.criteria.CriteriaQuery cq =
    getEntityManager().getCriteriaBuilder()
                        .createQuery();
    cq.select(cq.from(entityClass));
    return getEntityManager().createQuery(cq)
                        .getResultList();
}
```

ORACLE

Copyright © 2012, Oracle and/or its affiliates. All rights reserved.

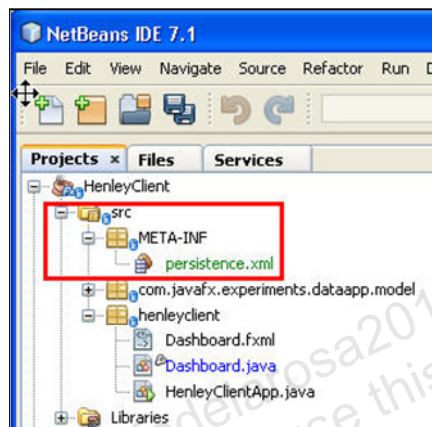
The simplest form of query execution is via the `getResultList()` method. It returns a collection containing the query results. If the query did not return any data, the collection is empty. The return type is specified as a `List` instead of a `Collection` in order to support queries that specify a sort order.

For queries that return values, the developer may choose to call either `getSingleResult()` if the query is expected to return a single result or `getResultList()` if more than one result may be returned. The `executeUpdate()` method is used to invoke bulk update and delete queries.

The code in the slide shows the use of the Criteria API.

Packaging and Deployment

- You must define your application's persistence unit in a configuration file called `persistence.xml`.
- This file should exist alongside your application in a `META-INF` directory.



ORACLE

Copyright © 2012, Oracle and/or its affiliates. All rights reserved.

This slide and the following few slides cover how to package and deploy a JPA application to a Java SE environment.

Packaging means integrating the parts of the application so that it can be interpreted correctly and the application works accurately when deployed to an application server or run in a stand-alone JVM.

Deployment is the process of getting the application into an execution environment and running it. There are some obvious differences between deploying in a Java EE server and deploying to a Java SE runtime environment; for example, some of the Java EE container services will not be present.

The set of entities in your application is called a persistence unit.

You can put the `META-INF` subdirectory within your project's source directory as shown in the diagram in the slide.

The `persistence.xml` file lets the persistence API implementation know about entities.

The persistence.xml File

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0" ..... >
1  <persistence-unit name="henley" transaction-
2  type="RESOURCE_LOCAL">
3  <provider>oracle.toplink.essentials.ejb.cmp3
    EntityManagerFactoryProvider</provider>
    <class>com.javafx.experiments.dataapp.model.DailySales
    </class>
    <class>com.javafx.experiments.dataapp.model.Address
    </class>
    <class>com.javafx.experiments.dataapp.model.Customer
    </class>
    .....
    </persistence-unit>
</persistence>
```

ORACLE

Copyright © 2012, Oracle and/or its affiliates. All rights reserved.

The persistence.xml file has many functions, but its most important task in the desktop environment is to list all the entities in your application and to name the persistence unit. Listing entity classes is required for portability in Java SE environments. The file may contain one or more persistence unit configurations that are separate and distinct from one another.

Tools like the NetBeans IDE can create the outline for you.

The important elements of the file number listed in the slide are as follows:

1. persistence-unit

The <persistence-unit> tag defines the name of the persistence unit used in the EntityManagerFactory method, and transaction-type defines the transaction resource type—either local (RESOURCE_LOCAL) or container (JTA). The default for Java SE applications is resource local transactions.

2. Provider

The <provider> tag identifies the persistence provider. As mentioned previously, the reference implementation for JPA 2.0 is EclipseLink.

3. Class

Use the class element to list the entity class names in your application.

4. Property (shown in the next slide)

The `<properties>` key in the persistence unit definition includes properties named with the prefix `javax.persistence.`, such as:

- `jdbc.driver` (the JDBC driver class)
- `jdbc.url` (the JDBC URL to load)
- `jdbc.user` and `jdbc.password` (the username and password for the database defined in the URL)

The JDBC properties are now standard in JPA 2.0.

The persistence.xml File

4
<properties>
 <property name="toplink.jdbc.user"
value="henley"/>
 <property name="toplink.jdbc.password"
value="henley"/>
 <property name="toplink.jdbc.url"
value="jdbc:derby://localhost:1527/henley"/>
 <property name="toplink.jdbc.driver"
value="org.apache.derby.jdbc.ClientDriver"/>
</properties>
 </persistence-unit>
</persistence>

database connection properties

ORACLE

Copyright © 2012, Oracle and/or its affiliates. All rights reserved.

In Java SE desktop environments, you should put database connection properties in the `persistence.xml` file if Java Naming and Directory Interface (JNDI) lookups are not possible.

Database connection properties include:

- The username and password for the database connection
- The database connection string
- Driver classname

Additionally, you can even include persistence provider properties like options to create or drop-create new tables.

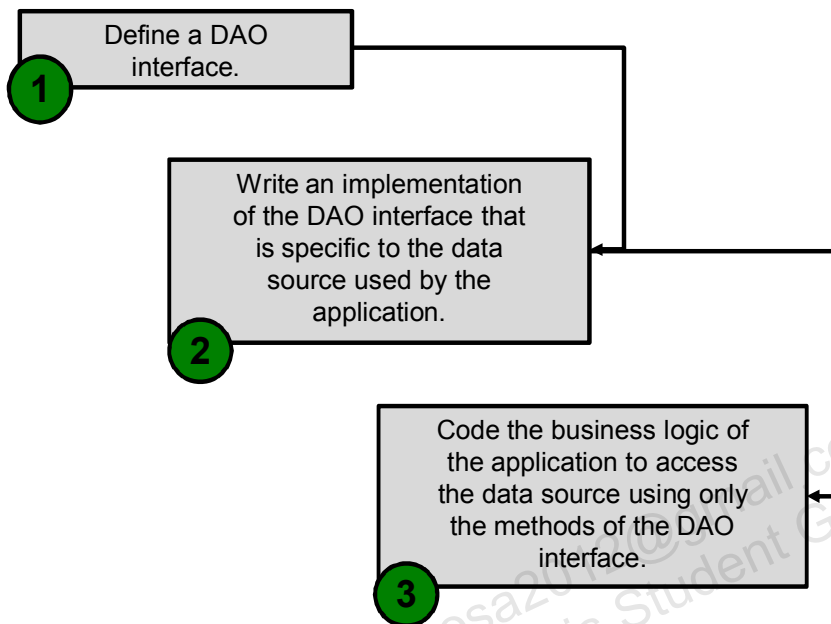
Data Access Objects

- The role of a data access object (DAO) is to segregate completely the persistence logic from business or presentation logic.
- This ensures that the user of DAOs is unaware of the database, its physical representation, and the relationships between the objects of the database.
- The primary advantage of using the DAO design pattern is isolating the code that needs to be changed whenever the application's data source changes.
- The DAO pattern is suitable for introducing JPA into an existing application.

ORACLE

Copyright © 2012, Oracle and/or its affiliates. All rights reserved.

Applying the DAO Design Pattern



ORACLE

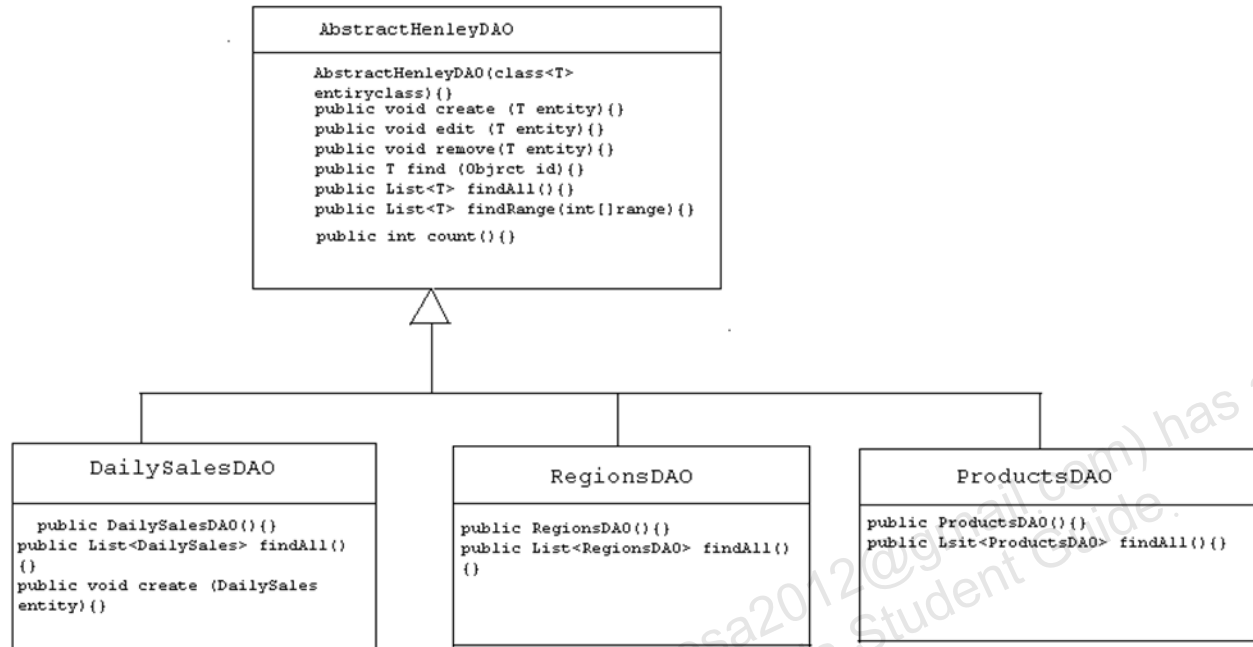
Copyright © 2012, Oracle and/or its affiliates. All rights reserved.

The DAO design pattern is implemented using the steps listed in the slide:

- The DAO interface contains methods for interacting (reading and writing) with a data source. These methods signatures must be generic. This would enable these methods not to contain any aspects that would bind them to a specific database.

Directly exposing persistence APIs to other application tiers is something to be avoided. Therefore, a well-designed data access object implements a simple persistence manager interface by delegating to a particular persistence mechanism.

Applying the DAO in the HenleyApp



ORACLE

Copyright © 2012, Oracle and/or its affiliates. All rights reserved.

AbstractHenleyDAO class is an abstract class and it provides all the data access methods. The methods use generics to create, edit, remove, and find data in the database.

DailySalesDAO, RegionsDAO, and ProductsDAO extend the AbstractHenleyDAO class and override the required methods of the abstract class.

The entity class instances (DailySales, Products, and Regions) are used to represent tables of the database.

HenleyApp Code Example

```
public abstract class AbstractHenleyDAO<T> {  
    private Class<T> entityClass;  
    protected HenleyEntityManager HenleyEnM;  
  
    public AbstractHenleyDAO(Class<T> entityClass) {  
        this.entityClass = entityClass;  
        HenleyEnM=new HenleyEntityManager();  
        HenleyEnM.create();  
    }  
  
    public void create(T entity) {  
        HenleyEnM.getEntityManager().getTransaction().begin();  
  
        HenleyEnM.getEntityManager().persist(entity);  
        HenleyEnM.getEntityManager().getTransaction().commit();  
        HenleyEnM.close();  
    }  
}
```

1
AbstractHenleyDAO
class

2
DailySalesDAO

```
public class DailySalesDAO extends AbstractHenleyDAO<DailySales> {  
  
    public DailySalesDAO() {  
        super(DailySales.class);  
    }  
  
    @Override public List<DailySales> findAll() {  
        return super.findAll();  
    }  
  
    @Override  
    public void create(DailySales entity) {  
        super.create(entity);  
    }  
}
```

ORACLE

Copyright © 2012, Oracle and/or its affiliates. All rights reserved.

In the first code snippet given in the slide, we can see that the create method of the AbstractHenleyDAO class provides the logic for creating a new record in a table. The method uses generics so that it can be used for creating or persisting any entity.

In the second code snippet, the overridden create method of DailySalesDAO is shown. It invokes its super classes' create method and provides the DailySales entity in the constructor.

Summary

In this lesson, you should have learned how to:

- Identify relationships in an application
- Build and deploy a JPA application in a Java SE environment
- Apply a two-tier design in the HenleyApp application



ORACLE

Copyright © 2012, Oracle and/or its affiliates. All rights reserved.

Practice 10: Overview

- 10-1: Identifying Entity Relationships in the BrokerTool Application
- 10-2: Implementing Database Connectivity in the BrokerTool Application by Using the JPA



ORACLE

Copyright © 2012, Oracle and/or its affiliates. All rights reserved.

Adolfo De+la+Rosa (adolfodelarosa2012@gmail.com) has a
non-transferable license to use this Student Guide.