CHAPTER 9
Persistence: EntityManager

Persistence is a key piece of the Java EE platform. In older versions of J2EE, the EJB 2.x specification was responsible for defining this layer. In Java EE 5, persistence was spun off into its own specification. Now, in EE6, we have a new revision called the Java Persistence API, Version 2.0, or more simply, JPA.

Persistence provides an ease-of-use abstraction on top of JDBC so that your code may be isolated from the database and vendor-specific peculiarities and optimizations. It can also be described as an object-to-relational mapping engine (ORM), which means that the Java Persistence API can automatically map your Java objects to and from a relational database. In addition to object mappings, this service also provides a query language that is very SQL-like but is tailored to work with Java objects rather than a relational schema.

In short, JPA handles the plumbing between Java and SQL. EJB provides convenient integration with JPA via the *entity bean*.

Entity beans, unlike session and message-driven types, are not server-side components. Instead, they are simple objects whose state can be synchronized with an underlying persistent storage provider. They are created just like any normal instance, typically using the `new` operator, and have no special APIs that must be implemented by the Entity class.

Much like EJB's server-side types, however, entity beans gain powerful services when used within the context of the container. In the case of persistence, Entity instances may become managed objects under the control of a service called the `javax.persistence.EntityManager`.

In the Java Persistence specification, the `EntityManager` is the central authority for all persistence actions. Because entities are plain Java objects, they do not become persistent until your code explicitly interacts with the `EntityManager` to make them persistent. The `EntityManager` manages the object-relational (O/R) mapping between a fixed set of entity classes and an underlying data source. It provides APIs for creating queries, finding objects, synchronizing, and inserting objects into the database. It also can

provide caching and manage the interaction between an entity and transactional services in a Java EE environment such as the Java Transaction API (JTA). The `EntityManager` is well integrated with Java EE and EJB but is not limited to this environment; it also can be used in plain Java programs.

You can use Java Persistence outside of an application server and in plain Java SE programs.

This chapter focuses on the details of the persistence service and how it can be accessed within Java EE.

Entities Are POJOs

Entities, in the Java Persistence specification, are plain old Java objects (POJOs). You allocate them with the `new()` operator just as you would any other plain Java object. Their state is not synchronized with persistent storage unless associated with an `EntityManager`. For instance, let's look at a simple example of an `Employee` entity:

```
import javax.persistence.Entity;
import javax.persistence.Id;

/**
 * Represents an Employee in the system.  Modeled as a simple
 * value object with some additional EJB and JPA annotations.
 *
 * @author <a href="mailto:andrew.rubinger@jboss.org">ALR</a>
 * @version $Revision: $
 */
@Entity
// Mark that we're an Entity Bean, EJB's integration point
// with Java Persistence
public class Employee
{

   /**
    * Primary key of this entity
    */
   @Id
   // Mark that this field is the primary key
   private Long id;

   /**
    * Name of the employee
    */
   private String name;
```

```
/**
 * Default constructor, required by JPA
 */
public Employee()
{

}

/**
 * Convenience constructor
 */
public Employee(final long id, final String name)
{
   // Set
   this.id = id;
   this.name = name;
}

/**
 * @return the id
 */
public Long getId()
{
   return id;
}

/**
 * @param id the id to set
 */
public void setId(final Long id)
{
   this.id = id;
}

/**
 * @return the name
 */
public String getName()
{
   return name;
}

/**
 * @param name the name to set
 */
public void setName(final String name)
{
   this.name = name;
}
```

```
    /**
     * {@inheritDoc}
     * @see java.lang.Object#toString()
     */
    @Override
    public String toString()
    {
       return "Employee [id=" + id + ", name=" + name + "]";
    }
}
```

If we allocate instances of this `Employee` class, no magic happens when `new()` is invoked. Calling the `new` operator does not magically interact with some underlying service to create the `Employee` in the database:

```
// This is just an object
Employee hero = new Employee(1L,"Trey Anastasio");
```

Allocated instances of the `Customer` class remain POJOs until you ask the `EntityManager` to create the entity in the database.

Managed Versus Unmanaged Entities

Before we can go any deeper into the entity manager service, we need to delve more deeply into the lifecycle of entity object instances. An entity bean instance is either managed (aka attached) by an entity manager or unmanaged (aka detached). When an entity is attached to an `EntityManager`, the manager tracks state changes to the entity and synchronizes those changes to the database whenever the entity manager decides to flush its state. When an entity is detached, it is unmanaged. Any state changes to an entity that is detached are not tracked by the entity manager.

Persistence Context

A *persistence context* is a set of managed entity object instances. Persistence contexts are managed by an entity manager. The entity manager tracks all entity objects within a persistence context for changes and updates made, and flushes these changes to the database using the flush mode rules discussed later in this chapter. Once a persistence context is closed, all managed entity object instances become detached and are no longer managed. Once an object is detached from a persistence context, it can no longer be managed by an entity manager, and any state changes to this object instance will not be synchronized with the database.

When a persistence context is closed, all managed entity objects become detached and are unmanaged.

There are two types of persistence contexts: transaction-scoped and extended persistence contexts.

Transaction-scoped persistence context

Though we'll discuss transactions in much greater detail in Chapter 17, it's important to be aware of a transaction in simple terms to discuss Entities. For the purposes of this discussion, we may think of a transaction as a set of beginning and end boundaries. Everything executed in between must either fully succeed or fully fail, and state changes made within a transaction are visible elsewhere in the system only when the end boundary completes successfully.

Persistence contexts may live as long as a transaction and be closed when a transaction completes. This is called a *transaction-scoped persistence context*. When the transaction completes, the transaction-scoped persistence context will be destroyed and all managed entity object instances will become detached. Only persistence contexts managed by an application server may be transaction-scoped. In other words, only `EntityManager` instances injected with the `@PersistenceContext` annotation or its XML equivalent may be transaction-scoped.

```
@PersistenceContext(unitName="nameOfThePc")
EntityManager entityManager;

// Assume this method invocation takes place within
// the context of a running Transaction
public Employee getTheWorldsBestGuitarist()
{
    // Trey has key 1 in the DB
    Employee trey = entityManager.find(Employee.class, 1);
    trey.setName("Ernest Joseph Anastasio, III");
    return cust;
}
```

When `getTheWorldsBestGuitarist()` is executed, the EJB container invokes it within the context of a JTA transaction. An `Employee` reference is pulled from the `EntityManager`, and the `setName()` method is used to change the name of the employee. The `Employee` instance that the `EntityManager` returns will remain managed for the duration of the JTA transaction. This means that the change made by calling the `setName()` method will be synchronized with the database when the JTA transaction completes and commits.

The `Employee` instance is also returned by `getTheWorldsBestGuitarist()`. After the JTA transaction completes, the transaction-scoped persistence context is destroyed, and this `Employee` instance is no longer managed. This means that if `setName()` is called after it becomes detached, no changes will be made to any database.

Extended persistence context

Persistence contexts may also be configured to live longer than a transaction. This is called an *extended persistence context*. Entity object instances that are attached to an extended context remain managed even after a transaction is complete. This feature is extremely useful in situations where you want to have a conversation with your database but not keep a long-running transaction, as transactions hold valuable resources such as JDBC connections and database locks. Here's some small pseudocode to illustrate this concept:

```
Employee pageMcConnell = null;

transaction.begin(); // Start Tx 1
// Page has key 2 in the DB
pageMcConnell = extendedEntityManager.find(Employee.class, 2L);
transaction.commit(); // Tx 1 Ends

transaction.begin(); // Start Tx 2
pageMcConnell.setName("Leo!"); // Change Page's name to his nickname
extendedEntityManager.flush(); // Flush changes to the DB manually
// pageMcConnell instance is to remain managed, and changes are flushed
transaction.commit(); // End Tx 2
```

In this example, a local variable, `pageMcConnell`, is initialized by calling the `EntityManager.find()` method in transaction 1. Unlike a transaction-scoped persistence context, the `Employee` instance pointed to by this local variable remains managed. This is because extended persistence context stays alive past the completion of transaction 1. In transaction 2, the employee is updated and the changes are flushed to the database.

Extended persistence contexts may be created and managed by application code, and we'll see examples of this later in this chapter. They can also be created and managed by stateful session beans.

Detached entities

Entity instances become unmanaged and detached when a transaction scope or extended persistence context ends. An interesting side effect is that detached entities can be serialized and sent across the network to a remote client. The client can make changes remotely to these serialized object instances and send them back to the server to be merged back and synchronized with the database.

This behavior is very different from the EJB 2.1 entity model, where entities are always managed by the container. In EJB 2.1, applications using entity beans always had a proxy to the entity bean; in EJB 3.x, we work with concrete instances of plain Java classes. For EJB 2.1 developers, this behavior will seem strange at first, since you are used to the container managing every aspect of the entity. You'll find that after you get used to the new EJB 3.x model, your application code actually shrinks and is easier to manage.

The reason we can eliminate code is very simple to illustrate. EJB 2.1 code often used the Value Object Pattern (often called Data Transfer Objects). The idea of this pattern was that the entity bean exposed a method that copied its entire state into an object that could be serialized to remote clients (like a Swing application) that needed access to the entity's state:

```
// EJB 2.1 Entity bean class
public class CustomerBean implements javax.ejb.EntityBean {

    CustomerValueObject getCustomerVO() {
       return new CustomerValueObject(getFirstName(), getLastName(),
                                      getStreet(), getCity(), getState, getZip());
    }
}
```

This is exactly the kind of plumbing we earlier deemed evil when discussing the benefits of using a Container and Application Server. Application code, armed with the right tools, should be free of this kind of error-prone and excessive noise.

Also, it is very expensive to make a remote method call to an entity bean from a client. If the client had to call `getFirstName()`, `getLastName()`, etc., to get information about a customer it was displaying, performance would suffer. This is where the Value Object Pattern came in. EJB 3.x eliminates the need for this pattern because persistent objects become value objects automatically when they are detached from a persistent context. One side effect we encounter in dealing with detached entities revolves around entity relationships, which we'll discuss later.

Packaging a Persistence Unit

An `EntityManager` maps a fixed set of classes to a particular database. This set of classes is called a *persistence unit*. Before you can even think about creating or querying entities with an entity manager, you must learn how to package a persistence unit for use within a Java SE (regular Java application) or Java EE (application server) environment. A persistence unit is defined in a *persistence.xml* file, which is described by the JPA2 specification in section 8.2.1. This file is a required deployment descriptor for the Java Persistence specification. A *persistence.xml* file can define one or more persistence units. The JAR file or directory that contains a *META-INF/persistence.xml* file is called the "root" of the persistence unit, and this may be:

- An EJB JAR file
- The *WEB-INF/classes* directory of a WAR file
- A JAR file in the *WEB-INF/lib* directory of a WAR file
- A JAR file in an EAR library directory
- An application client JAR file

The structure of one of these JAR files may look like the following:

```
/META-INF/
/META-INF/persistence.xml
/org/
/org/example/
/org/example/entity/
/org/example/entity/Employee.class
```

The *persistence.xml* deployment descriptor defines the identities and configuration properties of each persistence unit described within it. Each persistence unit must have an identity, although the empty string is a valid name.

The set of classes that belong to the persistence unit can be specified, or you can opt for the persistence provider to scan the JAR file automatically for the set of classes to deploy as entities. When scanning is used, the persistence provider will look at every class file within the JAR to determine whether it is annotated with the `@javax.persistence.Entity` annotation, and if it is, it will add it to the set of entities that must be mapped.

Each persistence unit is tied to one and only one data source. In Java SE environments, vendor-specific configuration must be used to define and configure these data sources. In Java EE environments, specific XML elements define this association.

The root of the *persistence.xml* XML schema is the `<persistence>` element, which contains one or more `<persistence-unit>` elements. Each `<persistence-unit>` has two attributes: `name` (required) and `transaction-type` (optional). The subelements of `<persistence-unit>` are `<description>` (optional), `<provider>` (optional), `<jta-data-source>` (optional), `<non-jta-data-source>` (optional), `<mapping-file>` (optional), `<jar-file>` (optional), `<class>` (optional), `<properties>` (optional), and `<exclude-unlisted-classes>` (optional).

Here's an example of a *persistence.xml* file:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
  version="2.0">
   <persistence-unit name="nameOfMyPu">
      <jta-data-source>java:/DataSourceNameInJndi</jta-data-source>
      <properties>
          <!-- Standard and Provider-specific config may go here -->
          <property name="myprovider.property.name" value="someValue"/>
      </properties>
   </persistence-unit>
</persistence>
```

The `name` attribute defines the name by which the unit will be referenced. This name is used by injection annotations and XML deployment descriptor elements to reference this unit. This attribute is required.

The `transaction-type` attribute defines whether you want your persistence unit to be managed by and integrated with Java EE transactions (JTA) or you want to use the resource local (`RESOURCE_LOCAL`) `javax.persistence.EntityTransaction` API to manage the integrity of your `EntityManager` instances. This attribute defaults to JTA in Java EE environments and to `RESOURCE_LOCAL` in SE environments.

The `<description>` element is really just a comment describing the given persistence unit and is not required.

The `<provider>` element is the fully qualified name of a class that implements the `javax.persistence.PersistenceProvider` interface. In Java EE and SE environments, the persistence implementation is pluggable: your vendor provides an appropriate implementation. Usually, you do not have to define this element and can rely on the default value.

If you are using JTA or `RESOURCE_LOCAL` persistence units, you will probably define a `<jta-data-source>` or `<non-jta-data-source>` element, respectively. These elements specify a vendor-specific identity of a particular data source. Usually, this string is the global JNDI name for referencing the data source. If neither is defined, then a vendor-provided default will be used.

The `<properties>` element defines the set of vendor-specific attributes passed to the persistence provider. They specify configuration that is specific to a vendor implementation. Since there is no registry or JNDI service within Java SE, this is usually how vendors configure data sources, instead of using the `<jta-data-source>` and `<non-jta-data-source>` elements.

The Persistence Unit Class Set

A persistence unit maps a fixed set of classes to a relational database. By default, if you specify no other metadata within your *persistence.xml* file, the JAR file that contains *persistence.xml* will be scanned from its root for any classes annotated with the `@javax.persistence.Entity` annotation. These classes are added to the set of classes the persistence unit will manage. You can specify additional JARs that you want to be scanned using the `<jar-file>` element. The value of this element is a path relative to the JAR file that contains *persistence.xml*:

```
<persistence>
   <persistence-unit name="nameOfMyPu">
      ...
      <jar-file>../lib/employee.jar</jar-file>
      ...
   </persistence-unit>
</persistence>
```

Scanning JAR files is guaranteed to work in Java EE environments but is not portable in Java SE applications. In theory, it may not be possible to determine the set of JAR

files that must be scanned. In practice, however, this is not the case. Whether you do or do not rely on a JAR scan, classes can be listed explicitly with the `<class>` element:

```
<persistence>
   <persistence-unit name="nameOfMyPu">
      ...
      <class>org.example.entity.Employee</class>
      <class>org.example.entity.AnotherEntity</class>
      ...
   </persistence-unit>
</persistence>
```

The `Employee` and `AnotherEntity` classes listed within the `<class>` elements are added to the persistence unit set along with any other classes scanned in the persistence unit's archive. If you do not want the *persistence.xml*'s JAR file to be scanned, then you can use the `<exclude-unlisted-classes>` element.

```
<persistence>
   <persistence-unit name="nameOfMyPu">
      ...
      <exclude-unlisted-classes/>
      ...
   </persistence-unit>
</persistence>
```

The final set of classes is determined by a union of all of the following metadata:

- Classes annotated with `@Entity` in the *persistence.xml* file's JAR file (unless `<exclude-unlisted-classes>` is specified)
- Classes annotated with `@Entity` that are contained within any JARs listed with any `<jar-file>` elements
- Classes mapped in the *META-INF/orm.xml* file if it exists
- Classes mapped in any XML files referenced with the `<mapping-file>` element
- Classes listed with any `<class>` elements

Usually, you will find that you do not need to use the `<class>`, `<jar-file>`, or `<mapping-file>` element. One case where you might need one of these elements is when the same class is being used and mapped within two or more persistence units.

Obtaining an EntityManager

Now that you have packaged and deployed your persistence units, you need to obtain access to an `EntityManager` so that you can persist, update, remove, and query your entity beans within your databases. In Java SE, entity managers are created using a `javax.persistence.EntityManagerFactory`. Although you can use the factory interface in Java EE, the platform provides some additional features that make it easier and less verbose to manage entity manager instances.

EntityManagerFactory

`EntityManager`s may be created or obtained from an `EntityManagerFactory`. In a Java SE application, you must use an `EntityManagerFactory` to create instances of an `EntityManager`. Using the factory isn't a requirement in Java EE, and we recommend that you don't use it directly from application code.

```
package javax.persistence;

import java.util.Map;
import javax.persistence.metamodel.Metamodel;
import javax.persistence.criteria.CriteriaBuilder;

public interface EntityManagerFactory
{
    public EntityManager createEntityManager();
    public EntityManager createEntityManager(Map map);
    public CriteriaBuilder getCriteriaBuilder();
    public Metamodel getMetamodel();
    public boolean isOpen();
    public void close();
    public Map<String, Object> getProperties();
    public Cache getCache();
    public PersistenceUnitUtil getPersistenceUnitUtil();
}
```

The `createEntityManager()` methods return `EntityManager` instances that manage a distinct extended persistence context. You can pass in a `java.util.Map` parameter to override or extend any provider-specific properties you did not declare in your *persistence.xml* file. When you are finished using the `EntityManagerFactory`, you should `close()` it (unless it is injected; we'll discuss this later). The `isOpen()` method allows you to check whether the `EntityManagerFactory` reference is still valid.

Getting an EntityManagerFactory in Java EE

In Java EE, it is easy to get an `EntityManagerFactory`. It can be injected directly into a field or *setter* method of your EJBs using the `@javax.persistence.PersistenceUnit` annotation:

```
package javax.persistence;

@Target({ TYPE, METHOD, FIELD })
@Retention(RUNTIME)
public @interface PersistenceUnit
{
    String name() default "";

    String unitName() default "";
}
```

The `unitName()` is the identity of the `PersistenceUnit`. When the `PersistenceUnit` is used, it not only injects the `EntityManagerFactory`, it also registers a reference to it within the JNDI ENC of the EJB. (The JNDI ENC is discussed more in Chapter 16.) The EJB container is responsible for noticing the `@PersistenceUnit annotation` and injecting the correct factory:

```
import javax.persistence.*;
import javax.ejb.*;

@Stateless
public MyBean implements MyBusinessInterface
{
   @PersistenceUnit(unitName="nameOfMyPu")
   private EntityManagerFactory factory1;
   private EntityManagerFactory factory2;

   @PersistenceUnit(unitName="nameOfAnotherPu")
   public void setFactory2(EntityManagerFactory f)
   {
      this.factory2 = f;
   }
}
```

When an instance of the stateless session bean is created, the EJB container sets the factory field to the persistence unit identified by `"nameOfMyPu"`. It also calls the `setFactory2()` method with the `"nameOfAnotherPu"` persistence unit.

In EJB, an injected `EntityManagerFactory` is automatically closed by the EJB container when the instance is discarded. In fact, if you call `close()` on an injected `EntityManagerFactory`, an `IllegalStateException` is thrown.

Obtaining a Persistence Context

A persistence context can be created by calling the `EntityManagerFactory.createEntityManager()` method. The returned `EntityManager` instance represents an extended persistence context. If the `EntityManagerFactory` is JTA-enabled, then you have to explicitly enlist the `EntityManager` instance within a transaction by calling the `EntityManager.joinTransaction()` method. If you do not enlist the `EntityManager` within the JTA transaction, the changes you make to your entities are not synchronized with the database.

`EntityManager.joinTransaction()` is required to be invoked only when an `EntityManager` is created explicitly using an `EntityManagerFactory`. If you are using EJB container-managed persistence contexts, you do not need to perform this extra step.

Using the `EntityManagerFactory` API is a bit verbose and can be awkward when you are making nested EJB calls, for instance. Fortunately, EJB and the Java Persistence

specification are nicely integrated. An `EntityManager` can be injected directly into an EJB using the `@javax.persistence.PersistenceContext` annotation.

```
package javax.persistence;

public enum PersistenceContextType
{
   TRANSACTION,
   EXTENDED
}

public @interface PersistenceProperty
{
   String name();
   String value();
}

@Target({TYPE, METHOD, FIELD})
@Retention(RUNTIME)
public @interface PersistenceContext
{
    String name() default "";
    String unitName() default "";
    PersistenceContextType type() default PersistenceContextType.TRANSACTION;
    PersistenceProperty[] properties() default {};
}
```

The `@PersistenceContext` annotation works in much the same way as `@PersistenceUnit`, except that an entity manager instance is injected instead of an `EntityManagerFactory`:

```
@Stateless
public class MyBean implements MyBusinessInterface
{
   @PersistenceContext(unitName="nameOfMyPu")
   private EntityManager entityManager;
}
```

The `unitName()` attribute identifies the persistence. By default, a transaction-scoped persistence context is injected when using this annotation. You can override this default with the `type()` attribute. When you access this transaction-scoped `EntityManager`, a persistence context becomes associated with the transaction until it finishes. This means that if you interact with any entity managers within the context of a transaction, even if they are different instances that are injected into different beans, the same persistence context will be used.

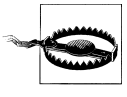
You must never call `close()` on an injected entity manager. Cleanup is handled by the application server. If you close an entity manager, an `IllegalStateException` is thrown.

An `EXTENDED` entity manager can only be injected into a stateful session bean; stateless session and message-driven beans are pooled, and there would be no way to close the persistence context and release any managed entity instances. In order to obtain an

extended context, a stateful session bean uses the `@javax.persistence.PersistenceContext` annotation with a type of `EXTENDED`:

```
@Stateful
public class MyBean implements MyBusinessInterface
{
   @PersistenceContext (unitName="nameOfMyPu", type=PersistenceContextType.EXTENDED)
   private EntityManager manager;
}
```

When this `MyBean` backing instance is created, a persistence context is also created for the injected manager field. The persistence context has the same lifespan as the bean. When the stateful session bean is removed, the persistence context is closed. This means that any entity object instances remain attached and managed as long as the stateful session bean is active.



It is strongly suggested that you use the `@PersistenceContext` annotation or the XML equivalent when using Java Persistence with EJBs. These features were defined to make it easier for developers to interact with entity beans. Entity managers created using `EntityManagerFactory` are more error-prone because the application developer has to worry about more things. For instance, the developer could forget to `close()` an entity manager and subsequently leak resources. Take advantage of the ease-of-use facilities of your EJB container!

Interacting with an EntityManager

Now that you have learned how to deploy and obtain a reference to an entity manager, you are ready to learn the semantics of interacting with it. The `EntityManager` API has methods to insert and remove entities from a database as well as merge updates from detached entity instances. There is also a rich query API that you can access by creating query objects from certain `EntityManager` methods:

```
package javax.persistence;

import java.util.Map;
import javax.persistence.metamodel.Metamodel;
import javax.persistence.criteria.CriteriaBuilder;
import javax.persistence.criteria.CriteriaQuery;

public interface EntityManager
{
    public void persist(Object entity);
    public <T> T merge(T entity);
    public void remove(Object entity);
    public <T> T find(Class<T> entityClass, Object primaryKey);
    public <T> T find(Class<T> entityClass, Object primaryKey,
                      Map<String, Object> properties);
    public <T> T find(Class<T> entityClass, Object primaryKey,
                      LockModeType lockMode);
    public <T> T find(Class<T> entityClass, Object primaryKey,
```

```
                    LockModeType lockMode,
                    Map<String, Object> properties);
    public <T> T getReference(Class<T> entityClass,
                              Object primaryKey);
    public void flush();
    public void setFlushMode(FlushModeType flushMode);
    public FlushModeType getFlushMode();
    public void lock(Object entity, LockModeType lockMode);
    public void lock(Object entity, LockModeType lockMode,
                     Map<String, Object> properties);
    public void refresh(Object entity);
    public void refresh(Object entity,
                        Map<String, Object> properties);
    public void refresh(Object entity, LockModeType lockMode);
    public void refresh(Object entity, LockModeType lockMode,
                      Map<String, Object> properties);
    public void clear();
    public void detach(Object entity);
    public boolean contains(Object entity);
    public LockModeType getLockMode(Object entity);
    public void setProperty(String propertyName, Object value);
    public Map<String, Object> getProperties();
    public Query createQuery(String qlString);
    public <T> TypedQuery<T> createQuery(CriteriaQuery<T> criteriaQuery);
    public <T> TypedQuery<T> createQuery(String qlString, Class<T> resultClass);
    public Query createNamedQuery(String name);
    public <T> TypedQuery<T> createNamedQuery(String name, Class<T> resultClass);
    public Query createNativeQuery(String sqlString);
    public Query createNativeQuery(String sqlString, Class resultClass);
    public Query createNativeQuery(String sqlString, String resultSetMapping);
    public void joinTransaction();
    public <T> T unwrap(Class<T> cls);
    public Object getDelegate();
    public void close();
    public boolean isOpen();
    public EntityTransaction getTransaction();
    public EntityManagerFactory getEntityManagerFactory();
    public CriteriaBuilder getCriteriaBuilder();
    public Metamodel getMetamodel();
}
```

Example: A Persistent Employee Registry

From our simple `Employee` entity defined earlier, we can use the `EntityManager` facilities to perform CRUD (Create, Read, Update, Delete) operations and build a simple persistent registry of employees. The full example is available in greater detail in Appendix F.

A Transactional Abstraction

Before we can take advantage of the `EntityManager` to flush and synchronize our changes with the database, we must set up a transactional context within which our code can

run. Because we're not going to delve into the full features of transactions until later, let's define a simple abstraction that marks the beginning and end of the transactional context.

```
public interface TxWrappingLocalBusiness
{
    **
    * Wraps the specified tasks in a new Transaction
    *
    * @param task
    * @throws IllegalArgumentException If no tasks are specified
    * @throws TaskExecutionException If an error occurred in invoking
    * {@link Callable#call()}
    */
   void wrapInTx(Callable<?>... tasks) throws IllegalArgumentException,
      TaskExecutionException;
}
```

From here we can construct simple `java.util.concurrent.Callable` implementations that encapsulate our JPA operations, and these will all run within a transaction that starts and ends with the invocation to `wrapInTx`. Let's assume we have an instance called `txWrapper` that implements `TxWrappingLocalBusiness` for us.

Persisting Entities

Persisting an entity is the act of inserting it within a database. We persist entities that have not yet been created in the database. To create an entity, we first allocate an instance of it, set its properties, and wire up any relationships it might have with other objects. In other words, we initialize an entity bean just as we would any other Java object. Once we've done this, we can then interact with the entity manager service by calling the `EntityManager.persist()` method:

```
// Execute the addition of the employees, and conditional checks,
// in the context of a Transaction
txWrapper.wrapInTx(new Callable<Void>()
{
   @Override
   public Void call() throws Exception
   {
      // Create a few plain instances
      final Employee dave = new Employee(ID_DAVE, NAME_DAVE);
      final Employee josh = new Employee(ID_JOSH, NAME_JOSH);
      final Employee rick = new Employee(ID_RICK, NAME_RICK);

      // Get the EntityManager from our test hook
      final EntityManager em = emHook.getEntityManager();

      // Now first check if any employees are found in the underlying persistent
      // storage (shouldn't be)
      Assert
        .assertNull("Employees should not have been added to the EM yet",
           em.find(Employee.class, ID_DAVE));
```

```
        // Check if the object is managed (shouldn't be)
        Assert.assertFalse("Employee should not be managed yet", em.contains(josh));

        // Now persist the employees
        em.persist(dave);
        em.persist(josh);
        em.persist(rick);
        log.info("Added: " + rick + dave + josh);

        // The employees should be managed now
        Assert.assertTrue(
          "Employee should be managed now, after call to persist",
          em.contains(josh));

        // Return
        return null;
      }
    });
```

When this method is called, the entity manager queues the `Employee` instances for insertion into the database, and the objects become managed. When the actual insertion happens depends on a few variables. If `persist()` is called within a transaction, the insert may happen immediately, or it may be queued until the end of the transaction, depending on the flush mode (described later in this chapter). You can always force the insertion manually within a transaction by calling the `EntityManager.flush()` method. You may call `persist()` outside of a transaction only if the entity manager is an `EXTENDED` persistence context. When you call `persist()` outside of a transaction with an `EXTENDED` persistence context, the insert is queued until the persistence context is associated with a transaction. An injected extended persistence context is automatically associated with a JTA transaction by the EJB container. For other extended contexts created manually with the `EntityManagerFactory` API, you must call `EntityManager.joinTransaction()` to perform the transaction association.

If the entity has any relationships with other entities, these entities may also be created within the database if you have the appropriate cascade policies set up. Cascading and relationships are discussed in detail in Chapter 11. Java Persistence can also be configured to automatically generate a primary key when the `persist()` method is invoked through the use of the `@GeneratedValue` annotation atop the primary key field or setter. So, in the previous example, if we had auto key generation enabled, we could view the generated key after the `persist()` method completed.

The `persist()` method throws an `IllegalArgumentException` if its parameter is not an entity type. `TransactionRequiredException` is thrown if this method is invoked on a transaction-scoped persistence context. However, if the entity manager is an extended persistence context, it is legal to call `persist()` outside of a transaction scope; the insert is queued until the persistence context interacts with a transaction.

Finding and Updating Entities

The entity manager provides two mechanisms for locating objects in your database. One way is with simple entity manager methods that locate an entity by its primary key. The other is by creating and executing queries.

find() and getReference()

The `EntityManager` has two different methods that allow you to find an entity by its primary key:

```
public interface EntityManager
{
   <T> T find(Class<T> entityClass, Object primaryKey);
   <T> T getReference(Class<T> entityClass, Object primaryKey);
}
```

Both methods take the entity's class as a parameter, as well as an instance of the entity's primary key. They use Java generics so that you don't have to do any casting. How do these methods differ? The `find()` method returns `null` if the entity is not found in the database. It also initializes the state based on the lazy-loading policies of each property (lazy loading is discussed in Chapter 10).

Once you have located an entity bean by calling `find()`, calling `getReference()`, or creating and executing a query, the entity bean instance remains managed by the persistence context until the context is closed. During this period, you can change the state of the entity bean instance as you would any other object, and the updates will be synchronized automatically (depending on the flush mode) or when you call the `flush()` method directly.

merge()

The Java Persistence specification allows you to merge state changes made to a detached entity back into persistence storage using the entity manager's `merge()` method.

If the entity manager isn't already managing an `Employee` instance with the same ID, a full copy of the parameter is made and returned from the `merge()` method. This copy is managed by the entity manager, and any additional *setter* methods called on this copy will be synchronized with the database when the `EntityManager` decides to flush. The original parameter remains detached and unmanaged.

The `merge()` method will throw an `IllegalArgumentException` if its parameter is not an entity type. The `TransactionRequiredException` is thrown if this method is invoked on a transaction-scoped persistence context. However, if the entity manager is an extended persistence context, it is legal to invoke this method outside of a transaction scope, and the update will be queued until the persistence context interacts with a transaction.

Now we can create a new `Employee` instance with new properties and synchronize this state with persistent storage:

```
// Now change Employee Dave's name in a Tx; we'll verify the changes were flushed
// to the DB later
txWrapper.wrapInTx(new Callable<Void>()
{
   @Override
   public Void call() throws Exception
   {
      // Get an EM
      final EntityManager em = emHook.getEntityManager();

      // Make a new "Dave" as a detached object with same primary key,
      // but a different name
      final Employee dave = new Employee(ID_DAVE, NAME_DAVE_NEW);

      // Merge these changes on the detached instance with the DB
      Employee managedDave = em.merge(dave);

      // Change Dave's name
      dave.setName(NAME_DAVE_NEW);
      log.info("Changing Dave's name: " + dave);

      // That's it - the new name should be flushed to the DB when the Tx completes
      return null;
   }
});
```

In this example, we are creating an `Employee` instance with a primary key ID of `ID_DAVE`. After we've performed this merge with the `EntityManager`, the `dave` instance's state is synced to the DB, and the object returned from the `merge` call is a managed object. Changing his name via a traditional call to the setter method for his name will change the state of this object, and the `EntityManager` will propagate these changes to the database when the transaction completes.

Alternatively, we could have used `EntityManager.find()` to look up `dave` from the DB, and then directly changed the name upon that reference.

`getReference()` differs from `find()` in that if the entity is not found in the database, this method throws a `javax.persistence.EntityNotFoundException` and there is no guarantee that the entity's state will be initialized.

Both `find()` and `getReference()` throw an `IllegalArgumentException` if their parameters are not an entity type. You are allowed to invoke them outside the scope of a transaction. In this case, any object returned is detached if the `EntityManager` is transaction-scoped but remains managed if it is an extended persistence context.

To prove that `dave`'s new name has been persisted, let's look him up again from a new transaction.

```
// Since we've changed Dave's name in the last transaction, ensure that we see the
// changes have been flushed and we can see them from a new Tx.
```

```
txWrapper.wrapInTx(new Callable<Void>()
{
   @Override
   public Void call() throws Exception
   {
      // Get an EM
      final EntityManager em = emHook.getEntityManager();

      // Look up "Dave" again
      final Employee dave = em.find(Employee.class, ID_DAVE);

      // Ensure we see the name change
      Assert.assertEquals("Employee Dave's name should have been changed",
       NAME_DAVE_NEW, dave.getName());

      // Now we'll detach Dave from the EM, this makes the object no longer managed
      em.detach(dave);

      // Change Dave's name again to some dummy value.  Because the object is
      // detached and no longer managed, we should not see this new value
      // synchronized with the DB
      dave.setName("A name we shouldn't see flushed to persistence");
      log.info("Changing Dave's name after detached: " + dave);

      // Return
     return null;
   }
});
```

To illustrate the difference between managed and unmanaged objects, here we manually detach `dave` from the `EntityManager`. We can still change his name just as we did before, but now these changes will not be synchronized with the database.

Queries

Persistent objects can also be located by using the JPA Query Language. Unlike EJB 2.1, there are no *finder* methods, and you must create a `Query` object by calling the `EntityManager`'s `createQuery()`, `createNamedQuery()`, or `createNativeQuery()` methods:

```
public interface EntityManager
{
    public Query createQuery(String qlString);
    public <T> TypedQuery<T> createQuery(CriteriaQuery<T> criteriaQuery);
    public <T> TypedQuery<T> createQuery(String qlString, Class<T> resultClass);
    public Query createNamedQuery(String name);
    public <T> TypedQuery<T> createNamedQuery(String name, Class<T> resultClass);
    public Query createNativeQuery(String sqlString);
    public Query createNativeQuery(String sqlString, Class resultClass);
    public Query createNativeQuery(String sqlString, String resultSetMapping);
}
```

Creating and executing a JPA QL query is analogous to creating and executing a JDBC `PreparedStatement`:

```
Query query = entityManager.createQuery("from Employee c where id=2");
Employee employee = (Employee)query.getSingleResult();
```

New to JPA2 is the addition of the Criteria API, a fluent expression to programmatically build queries. In many ways this can be more flexible than the string-based JPA QL.

Queries and JPA QL are discussed in Chapter 13.

All object instances returned by `find()`, `getResource()`, or a query remain managed as long as the persistence context in which you accessed them remains active. This means that further calls to `find()` (or whatever) will return the same entity object instance.

Removing Entities

An entity can be removed from the database by calling the `EntityManager.remove()` method. The `remove()` operation does not immediately delete the employee from the database. When the entity manager decides to flush, based on the flush rules described later in this chapter, an SQL `DELETE` is executed:

```
// Uh oh, Rick has decided to leave the company. Let's delete his record.
txWrapper.wrapInTx(new Callable<Void>()
{
   @Override
   public Void call() throws Exception
   {
      // Get an EM
      final EntityManager em = emHook.getEntityManager();

      // Look up Rick
      final Employee rick = em.find(Employee.class, ID_RICK);

      // Remove
      em.remove(rick);
      log.info("Deleted: " + rick);

      // Return
      return null;
   }
});
```

After `remove()` is invoked, the `rick` instance will no longer be managed and will become detached. Also, if the entity has any relationships to other entity objects, those can be removed as well, depending on the cascading rules discussed in Chapter 11. The `remove()` operation can be undone only by recreating the entity instance using the `persist()` method.

The `remove()` method throws an `IllegalArgumentException` if its parameter is not an entity type. The `TransactionRequiredException` is thrown if this method is invoked on a transaction-scoped persistence context. However, if the `EntityManager` is an extended

persistence context, it is legal to invoke this method outside of a transaction scope, and the `remove` will be queued until the persistence context interacts with a transaction.

refresh()

If you are concerned that a current managed entity is not up-to-date with the database, then you can use the `EntityManager.refresh()` method. The `refresh()` method refreshes the state of the entity from the database, overwriting any changes made to that entity. This effectively reverts any local changes.

If the entity bean has any related entities, those entities may also be refreshed, depending on the cascade policy set up in the metadata of the entity mapping.

The `refresh()` method throws an `IllegalArgumentException` if its parameter is not managed by the current entity manager instance. The `TransactionRequiredException` is thrown if this method is invoked on a transaction-scoped persistence context. However, if the entity manager is an extended persistence context, it is legal to invoke this method outside of a transaction scope. If the object is no longer in the database because another thread or process removed it, then this method will throw an `EntityNotFoundException`.

contains() and clear()

The `contains()` method takes an entity instance as a parameter. If this particular object instance is currently being managed by the persistence context, it returns `true`. It throws an `IllegalArgumentException` if the parameter is not an entity.

If you need to detach all managed entity instances from a persistence context, you can invoke the `clear()` method of the `EntityManager`. Be aware that when you call `clear()`, any changes you have made to managed entities are lost. It is wise to call `flush()` before `clear()` is invoked so you don't lose your changes.

flush() and FlushModeType

When you call `persist()`, `merge()`, or `remove()`, these changes are not synchronized with the database until the entity manager decides to flush. You can force synchronization at any time by calling `flush()`. By default, flushing automatically happens before a correlated query is executed (inefficient implementations may even flush before *any* query) and at transaction commit time. The exception to this default rule is `find()`. A flush does not need to happen when `find()` or `getReference()` is called, because finding by a primary key is not something that would be affected by any updates.

You can control and change this default behavior by using the `javax.persistence.FlushModeType` enumeration:

```
public enum FlushModeType
{
   AUTO,
   COMMIT
}
```

`AUTO` is the default behavior described in the preceding code snippet. `COMMIT` means that changes are flushed only when the transaction commits, not before any query. You can set the `FlushModeType` by calling the `setFlushMode()` method on the `EntityManager`.

Why would you ever want to change the `FlushModeType`? The default flush behavior makes a lot of sense. If you are doing a query on your database, you want to make sure that any updates you've made within your transaction are flushed so that your query will pick up these changes. If the entity manager didn't flush, then these changes might not be reflected in the query. Obviously, you want to flush changes when a transaction commits.

`FlushModeType.COMMIT` makes sense for performance reasons. The best way to tune a database application is to remove unnecessary calls to the database. Some vendor implementations will do all required updates with a batch JDBC call. Using `COMMIT` allows the entity manager to execute all updates in one huge batch. Also, an `UPDATE` usually ends up in the row being write-locked. Using `COMMIT` limits the amount of time the transaction holds on to this database lock by holding it only for the duration of the JTA commit.

Locking

The `EntityManager` API supports both read and write locks. Because locking behavior is closely related to the concept of transactions, using the `lock()` method is discussed in detail in Chapter 17.

unwrap() and getDelegate()

The `unwrap()` method allows you to obtain a reference to the underlying persistence provider object that implements the `EntityManager` interface. Most vendors will have API extensions to the `EntityManager` interface that can be executed by obtaining and typecasting this delegate object to a provider's proprietary interface. In theory, you should be able to write vendor-independent code, but in practice, most vendors provide a lot of extensions to Java Persistence that you may want to take advantage of in your applications. The `getDelegate()` method was provided in JPA1; for now, it's recommended that users call `unwrap()`.

CHAPTER 10
Mapping Persistent Objects

In this chapter, we take a thorough look at the process of developing entity beans—specifically, mapping them to a relational database. A good rule of thumb is that entity beans model business concepts that can be expressed as nouns. Although this is a guideline rather than a requirement, it helps determine when a business concept is a candidate for implementation as an entity bean. In grammar school, you learned that nouns are words that describe a person, place, or thing. The concepts of "person" and "place" are fairly obvious: a person entity might represent a customer or passenger, and a place entity might represent a city or port of call. Similarly, entity beans often represent "things": real-world objects, such as ships and credit cards, and abstractions, such as reservations. Entity beans describe both the state and behavior of real-world objects and allow developers to encapsulate the data and business rules associated with specific concepts; an `Employee` entity encapsulates the data and business rules associated with an employee, for example. This makes it possible for data associated with a concept to be manipulated consistently and safely.

Entities represent data in the database, so changes to an entity bean result in changes to the database. That's ultimately the purpose of an entity bean: to provide programmers with a simpler mechanism for accessing and changing data. It is much easier to change a customer's name by calling `Employee.setName()` than by executing an SQL command against the database. In addition, using entity beans provides opportunities for software reuse. Once an entity bean has been defined, its definition can be used throughout your application in a consistent manner. The concept of an employee, for example, is used in many areas of business, including booking, task assignment, and accounts payable. An `Employee` entity is also a unified model to information and thus ensures that access to its data is consistent and simple. Representing data as entity beans can make development easier and more cost-effective.

When a new entity is created and persisted into the entity manager service, a new record must be inserted into the database and a bean instance must be associated with that data. As the entity is used and its state changes, these changes must be synchronized with the data in the database: entries must be inserted, updated, and removed. The

process of coordinating the data represented by a bean instance with the database is called *persistence*.

The Java Persistence specification gave a complete overhaul to entity beans. CMP 2.1 had a huge weakness in that applications written to that specification were completely nonportable between vendors because there was no object-to-relational (O/R) mapping. O/R mapping was completely left to the vendor's discretion. These next chapters focus solely on Java Persistence's object mappings to a relational database. This chapter focuses on basic entity bean mappings to a relational database. Chapter 11 will discuss how entities can have complex relationships to one another and how Java Persistence can map these to your database. Chapter 13 will cover how we interact with entity beans through the Java Persistence Query Language (JPA QL) and Criteria APIs.

The Programming Model

Entities are plain Java classes in Java Persistence. You declare and allocate these bean classes just as you would any other plain Java object. You interact with the entity manager service to persist, update, remove, locate, and query for entity beans. The entity manager service is responsible for automatically managing the entity beans' state. This service takes care of enrolling the entity bean in transactions and persisting its state to the database. We've seen this power in the previous chapter.

The Employee Entity

The `Employee` class is a simple entity bean that models the concept of an employee within a company. Java Persistence is all about relational databases. This section introduces the `Employee` entity's design and implementation. This entity will be refactored in many different ways throughout this chapter so that we can show you the multiple ways in which you can map the entity to a relational database.

The Bean Class

The `Employee` bean class is a plain Java object that you map to your relational database. It has fields that hold state and, optionally, it has *getter* and *setter* methods to access this state. It must have, at minimum, a no-argument constructor (which may be the default, implicit constructor):

```
import javax.persistence.Entity;
import javax.persistence.Id;
/**
 * Represents an Employee in the system.  Modeled as a simple
 * value object with some additional EJB and JPA annotations.
 *
```

```
 * @author <a href="mailto:andrew.rubinger@jboss.org">ALR</a>
 * @version $Revision: $
 */
@Entity
// Mark that we're an Entity Bean, EJB's integration point
// with Java Persistence
public class Employee
{

   /**
    * Primary key of this entity
    */
   @Id
   // Mark that this field is the primary key
   private Long id;

   /**
    * Name of the employee
    */
   private String name;

   /**
    * Default constructor, required by JPA
    */
   public Employee() { }

   /**
    * Convenience constructor
    */
   public Employee(final long id, final String name)
   {
      // Set
      this.id = id;
      this.name = name;
   }

   public Long getId() { return id; }
   public void setId(final Long id) { this.id = id; }
   public String getName() { return name; }
   public void setName(final String name) { this.name = name; }

   @Override
   public String toString()
   {
      return Employee.class.getSimpleName() + " [id=" + id + ", name=" + name + "
]";
   }
}
```

Java Persistence requires only two pieces of metadata when you are creating a persistent class: the `@javax.persistence.Entity` annotation denotes that the class should be mapped to your database, and the `@javax.persistence.Id` annotation marks which property in your class will be used as the primary key. The persistence provider will assume that all other properties in your class will map to a column of the same name and of the

same type. The table name will default to the unqualified name of the bean class. Here is the table definition the persistence provider is assuming you are mapping to:

```
create table Employee(
   id long primary key not null,
   name VARCHAR(255)
);
```

The `@javax.persistence.Entity` annotation tells the persistence provider that your class can be persisted:

```
package javax.persistence;

@Target(TYPE) @Retention(RUNTIME)
public @interface Entity
{
   String name() default "";
}
```

The `@Entity` annotation has one `name()` attribute. This name is used to reference the entity within a JPA QL expression. If you do not provide a value for this attribute, the name defaults to the unqualified name of the bean class.

How you apply the `@javax.persistence.Id` annotation determines whether you will use the Java bean style for declaring your persistent properties or whether you will use Java fields. If you place the `@Id` annotation on a *getter* method, then you must apply any other mapping annotations on *getter* and *setter* methods in the class. The provider will also assume that any other *getter* and *setter* methods in your class represent persistent properties and will automatically map them based on their base name and type.

Earlier we placed the `@Id` annotation on a member field of the class. The persistence provider will also assume that any other member fields of the class are also persistent properties and will automatically map them based on their base name and type. Any mapping annotations must be placed on member fields in this example, not on *getter* or *setter* methods. Here, we are really defining the *access type*—that is, whether our relational mappings are defined on the fields or the methods of a class.

XML Mapping File

If you do not want to use annotations to identify and map your entity beans, you can alternatively use an XML mapping file to declare this metadata. By default, the persistence provider will look in the *META-INF* directory for a file named *orm.xml*, or you can declare the mapping file in the `<mapping-file>` element in the *persistence.xml* deployment descriptor.

The mapping file has a top element of `<entity-mappings>`. The `<entity>` element defines the entity class and access type: `PROPERTY` or `FIELD`. The `<id>` element is a subelement of the `<attributes>` element and defines what attribute your primary key is. Like annotated classes, the persistence provider will assume that any other property in your class is a persistent property, and you do not have to explicitly define them.

For brevity, we'll continue to use the annotation-based approach in our examples.

Basic Relational Mapping

A developer can take two directions when implementing entity beans. Some applications start from a Java object model and derive a database schema from this model. Other applications have an existing database schema from which they have to derive a Java object model.

The Java Persistence specification provides enough flexibility to start from either direction. If you are creating a database schema from a Java object model, most persistence vendors have tools that can autogenerate database schemas based on the annotations or XML metadata you provide in your code. In this scenario, prototyping your application is fast and easy, as you do not have to define much metadata in order for the persistence engine to generate a schema for you. When you want to fine-tune your mappings, the Java Persistence specification has the necessary annotations and XML mappings to do this.

If you have an existing database schema, many vendors have tools that can generate Java entity code directly from it. Sometimes, though, this generated code is not very object-oriented and doesn't map to your database very well. Luckily, the Java Persistence specification provides the necessary mapping capabilities to facilitate a solution to this problem.

You will find that your use of annotations and mapping XML will depend on the direction you are coming from. If you are autogenerating your schema from your entity classes, you probably will not need annotations such as `@Table` and `@Column` (covered in this chapter), as you will rely on well-defined specification defaults. If you have an existing schema or need to fine-tune the mapping, you may find that more metadata will need to be specified.

Elementary Schema Mappings

Let's assume we don't like the default table and column mappings of our original `Employee` entity class. Either we have an existing table we want to map to, or our DBA is forcing some naming conventions on us. Let's actually define the relational table we want to map our `Employee` entity to and use the `@javax.persistence.Table` and `@javax.persistence.Column` annotations to apply the mapping.

We want to change the table name and the column names of the `id` and `name` properties. We also want `name` to have a not-null constraint and want to set the `VARCHAR` length to 20. Let's modify our original `Employee` entity class and add the mapping annotations:

```
@Entity
@Table(name = "table_employees")
// Explicitly denote the name of the table in the DB
public class EmployeeWithCustomTableAndColumnMetadata
```

```
{
   @Id
   @Column(name="pk_employee_id")
   private Long id;

   @Column(name="employee_name",nullable=false, columnDefinition="integer")
   private String name;
   ...
}
```

@Table

The `@javax.persistence.Table` annotation tells the `EntityManager` service which relational table your bean class maps to. You do not have to specify this annotation if you do not want to, because, again, the table name defaults to the unqualified class name of the bean. Let's look at the full definition of this annotation:

```
package javax.persistence;

@Target({TYPE}) @Retention(RUNTIME)
public @interface Table
{
   String name() default "";
   String catalog() default "";
   String schema() default "";
   UniqueConstraint[] uniqueConstraints() default {};
}
```

The `catalog()` and `schema()` attributes are self-explanatory, as they identify the relational catalog and schema to which the table belongs.

```
public @interface UniqueConstraint
{
   /** Optional; chosen by provider if not specified **/
   String name default "";
   String[] columnNames();
}
```

The `@Table.uniqueConstraints()` attribute allows you to specify unique column constraints that should be included in a generated Data Definition Language (DDL). Some vendors have tools that can create DDLs from a set of entity classes or even provide automatic table generation when a bean is deployed. The `UniqueConstraint` annotation is useful for defining additional constraints when using these specific vendor features. If you are not using the schema generation tools provided by your vendor, then you will not need to define this piece of metadata.

@Column

Using the `@Column` annotation, we set the `name` property's column name to be `employee_name` and not nullable, and we set its database type to be an integer. We also set the `VARCHAR` length to 20. This is often important to save RAM in the database and

keep things moving efficiently; if you don't need the extra space, it's a good idea to restrict your field types to take up the least amount of data as possible.

The `@javax.persistence.Column` annotation describes how a particular field or property is mapped to a specific column in a table:

```
public @interface Column
{
   String name() default "";
   boolean unique() default false;
   boolean nullable() default true;
   boolean insertable() default true;
   boolean updatable() default true;
   String columnDefinition() default "";
   String table() default "";
   int length() default 255;
   int precision() default 0;
   int scale() default 0;
}
```

The `name()` attribute obviously specifies the column name. If it is unspecified, the column name defaults to the property or field name. The `table()` attribute is used for multitable mappings, which we cover later in this chapter. The rest of the attributes are used when you are autogenerating the schema from vendor-provided tools. If you are mapping to an existing schema, you do not need to define any of these attributes. The `unique()` and `nullable()` attributes define constraints you want placed on the column. You can specify whether you want this column to be included in `SQL INSERT` or `UPDATE` by using `insertable()` and `updatable()`, respectively. The `columnDefinition()` attribute allows you to define the exact DDL used to define the column type. The `length()` attribute determines the length of a `VARCHAR` when you have a `String` property. For numeric properties, you can define the `scale()` and `precision()` attributes.

Primary Keys

A *primary key* is the identity of a given entity bean. Every entity bean must have a primary key, and it must be unique. Primary keys can map to one or more properties and must map to one of the following types: any Java primitive type (including wrappers), `java.lang.String`, or a primary-key class composed of primitives and/or strings. Let's first focus on simple one-property primary keys.

@Id

The `@javax.persistence.Id` annotation identifies one or more properties that make up the primary key for your table:

```
package javax.persistence;

@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface Id
```

```
{
}
```

You can generate the primary key for your entity beans manually or have the persistence provider do it for you. When you want provider-generated keys, you have to use the `@javax.persistence.GeneratedValue` annotation:

```
package javax.persistence;

@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface GeneratedValue
{
   GenerationType strategy() default AUTO;
   String generator() default "";
}

public enum GenerationType
{
   TABLE, SEQUENCE, IDENTITY, AUTO
}
```

Persistence providers are required to provide key generation for primitive primary keys. You can define the type of primary generator you would like to have using the `strategy()` attribute. The `GeneratorType.AUTO` strategy is the most commonly used configuration, and it is the default:

```
/**
 * Primary key
 */
@Id
@GeneratedValue
private Long id;
```

Table Generators

The `TABLE` strategy designates a user-defined relational table from which the numeric keys will be generated. A relational table with the following logical structure is used:

```
create table GENERATOR_TABLE
(
    PRIMARY_KEY_COLUMN
    VARCHAR not null,
    VALUE_COLUMN long not null
);
```

The `PRIMARY_KEY_COLUMN` holds a value that is used to match the primary key you are generating for. The `VALUE_COLUMN` holds the value of the counter.

To use this strategy, you must have already defined a table generator using the `@javax.persistence.TableGenerator` annotation. This annotation can be applied to a class or to the method or field of the primary key:

```
@Target({ TYPE, METHOD, FIELD })
@Retention(RUNTIME)
```

```
public @interface TableGenerator {
    String name();
    String table() default "";
    String catalog() default "";
    String schema() default "";
    String pkColumnName() default "";
    String valueColumnName() default "";
    String pkColumnValue() default "";
    int initialValue() default 0;
    int allocationSize() default 50;
    UniqueConstraint[] uniqueConstraints() default { };
}
```

The `name()` attribute defines the name of the `@TableGenerator` and is the name referenced in the `@Id.generator()` attribute. The `table()`, `catalog()`, and `schema()` attributes describe the table definition of the generator table. The `pkColumnName()` attribute is the name of the column that identifies the specific entity primary key you are generating for. The `valueColumnName()` attribute specifies the name of the column that will hold the counter for the generated primary key. `pkColumnValue()` is the value used to match up with the primary key you are generating for. The `allocationSize()` attribute is how much the counter will be incremented when the persistence provider queries the table for a new value. This allows the provider to cache blocks so that it doesn't have to go to the database every time it needs a new ID. If you are autogenerating this table, then you can also define some constraints using the `uniqueConstraints()` attribute.

Let's look at how you would actually use this generator on the `Employee` entity:

```
@Entity
public class Employee implements java.io.Serializable {
   @TableGenerator(name="MY_GENERATOR"
                   table="GENERATOR_TABLE"
                   pkColumnName="PRIMARY_KEY_COLUMN"
                   valueColumnName="VALUE_COLUMN"
                   pkColumnValue="EMPLOYEE_ID"
                   allocationSize=10)
   @Id
 @GeneratedValue (strategy=GenerationType.TABLE, generator="MY_GENERATOR")
   private long id;
   ...
}
```

Now if you allocate and `persist()` an `Employee` entity, the `id` property will be autogenerated when the `persist()` operation is called.

Sequence Generators

Some RDBMs, specifically Oracle, have an efficient built-in structure to generate IDs sequentially. This is the `SEQUENCE` generator strategy. This generator type is declared via the `@javax.persistence.SequenceGenerator`:

```
package javax.persistence;
```

```
@Target({METHOD, TYPE, FIELD}) @Retention(RUNTIME)
public @interface SequenceGenerator {

    String name();
    String sequenceName() default "";
    String catalog() default "";
    String schema() default "";
    int initialValue() default 1;
    int allocationSize() default 50;
}
```

The `name()` attribute specifies how this `@SequenceGenerator` is referenced in `@Id` annotations. Use the `sequenceName()` attribute to define what sequence table will be used from the database. `initialValue()` is the first value that will be used for a primary key, and `allocationSize()` is how much it will be incremented when it is accessed. `schema()` and `catalog()`, like their counterparts in `@Table`, refer to the schema and catalog of the sequence generator, respectively. Let's again look at applying the `SEQUENCE` strategy on our `Employee` entity bean:

```
@Entity
@SequenceGenerator(name="EMPLOYEE_SEQUENCE",
     sequenceName="EMPLOYEE_SEQ")
public class Employee implements java.io.Serializable {
  @Id
 @GeneratedValue(strategy=GenerationType.SEQUENCE, generator="EMPLOYEE_SEQUENCE")
  private Long id;
}
```

This example is a little different from our `TABLE` strategy example in that the generator is declared on the bean's class instead of directly on the property. `TABLE` and `SEQUENCE` generators can be defined in either place. As with the `TABLE` generation type, the primary key is autogenerated when the `EntityManager.persist()` operation is performed.

Primary-Key Classes and Composite Keys

Sometimes relational mappings require a primary key to be composed of multiple persistent properties. For instance, let's say that our relational model specified that our `Employee` entity should be identified by both its last name and its Social Security number instead of an autogenerated numeric key. These are called *composite keys*. The Java Persistence specification provides multiple ways to map this type of model. One is through the `@javax.persistence.IdClass` annotation; the other is through the `@javax.persistence.EmbeddedId` annotation.

@IdClass

The first way to define a primary-key class (and, for that matter, composite keys) is to use the `@IdClass` annotation. Your bean class does not use this primary-key class internally, but it does use it to interact with the entity manager when finding a persisted

object through its primary key. `@IdClass` is a class-level annotation and specifies what primary-key class you should use when interacting with the entity manager.

```
@Target(TYPE)
@Retention(RUNTIME)
public @interface IdClass
{
   Class value();
}
```

In your bean class, you designate one or more properties that make up your primary key, using the `@Id` annotation. These properties must map exactly to properties in the `@IdClass`. Let's look at changing our `Employee` bean class to have a composite key made up of last name and Social Security number. First, let's define our primary-key class:

```
import java.io.Serializable;

public class ExternalEmployeePK implements Serializable
{
   private static final long serialVersionUID = 1L;
   private String lastName;
   private Long ssn;

   public String getLastName() { return lastName; }
   public void setLastName(String lastName) { this.lastName = lastName; }
   public Long getSsn() { return ssn; }
   public void setSsn(Long ssn) { this.ssn = ssn; }

   @Override public int hashCode(){ // Assume implemented }
   @Override public boolean equals(Object obj) { // Assume implemented  }

}
```

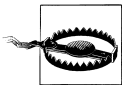
The primary-key class must meet these requirements:

- It must be Serializable.
- It must have a public no-arg constructor.
- It must implement the `equals()` and `hashCode()` methods.

Our Employee bean must have the same exact properties as the `ExternalEmployeePK` class, and these properties are annotated with multiple `@Id` annotations:

```
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.IdClass;

@Entity
@IdClass(ExternalEmployeePK.class)
// Use a composite primary key using a custom PK class
public class Employee
{
   @Id
   private String lastName;
```

```
    @Id
    private Long ssn;

    ...
}
```



Primary-key autogeneration is not supported for composite keys and primary-key classes. You will have to manually create the key values in code.

The primary-key class is used whenever you are querying for the `Employee`:

```
ExternalEmployeePK pk = new ExternalEmployeePK();
pk.setLastName("Rubinger");
pk.setSsn(100L);
Employee employee = em.find(Employee.class, pk);
```

Whenever you call an `EntityManager` method such as `find()` or `getReference()`, you must use the primary-key class to identify the entity.

@EmbeddedId

A different way to define primary-key classes and composite keys is to embed the primary-key class directly in your bean class. The `@javax.persistence.EmbeddedId` annotation is used for this purpose in conjunction with the `@javax.persistence.Embeddable` annotation:

```
package javax.persistence;

public @interface EmbeddedId
{
}

public @interface Embeddable
{
}
```

When we use an `@Embeddable` class as the type for our primary key, we mark the property as `@EmbeddedId`. Let's first see an example of our embeddable primary key type:

```
import java.io.Serializable;

import javax.persistence.Column;
import javax.persistence.Embeddable;
import javax.persistence.EmbeddedId;

@Embeddable
// Flag to JPA that we're intended to be embedded into an Entity
// class as a PK
public class EmbeddedEmployeePK implements Serializable
{
   private static final long serialVersionUID = 1L;
```

```
    @Column
    private String lastName;

    @Column
    private Long ssn;

    public String getLastName() { return lastName; }
    public void setLastName(String lastName) { this.lastName = lastName; }
    public Long getSsn() { return ssn; }
    public void setSsn(Long ssn) { this.ssn = ssn; }

    @Override
    public int hashCode() { // Assume this is implemented }
    @Override
    public boolean equals(Object obj) { // Assume this is implemented }

}
```

We then change our `Employee` bean class to use the `EmbeddedEmployeePK` directly, using the `@EmbeddedId` annotation:

```
@Entity
public class Employee
{
    @EmbeddedId
    private EmbeddedEmployeePK id;
    ...
}
```

The `EmbeddedEmployeePK` primary-key class is used whenever you are fetching the `Employee` using `EntityManager` APIs:

```
// Create the Embedded Primary Key, and use this for lookup
EmbeddedEmployeePK pk = new EmbeddedEmployeePK();
pk.setLastName("Rubinger");
pk.setSsn(100L);

// Now look up using our custom composite PK value class
Employee employee = em.find(Employee.class, pk);
```

Whenever you call an `EntityManager` method such as `find()` or `getReference()`, you must use the primary-key class to identify the entity.

If you do not want to have the `@Column` mappings with the primary-key class, or you just want to override them, you can use `@AttributeOverrides` to declare them directly in your bean class:

```
@Entity
public class Employee implements java.io.Serializable
{
    @EmbeddedId
    @AttributeOverrides({
        @AttributeOverride(name="lastName", column=@Column(name="LAST_NAME"),
        @AttributeOverride(name="ssn", column=@Column(name="SSN"))
    })
    private EmbeddedEmployeePK pk;
```

```
    ...
}
```

The `@AttributeOverrides` annotation is an array list of `@AttributeOverride` annotations. The `name()` attribute specifies the property name in the embedded class you are mapping to. The `column()` attribute allows you to describe the column the property maps to.

Property Mappings

So far, we have only shown how to specify column mappings for simple primitive types. There are still a few bits of metadata that you can use to fine-tune your mappings. In this section, you'll learn more annotations for more complex property mappings. Java Persistence has mappings for JDBC `Blob`s and `Clob`s, serializable objects, and embeddable objects, as well as optimistic concurrency with version properties. We discuss all of these.

@Transient

In our first example of our `Employee` bean class, we showed that the persistence manager would assume that every nontransient property (*getter/setter* or field, depending on your access type) in your bean class is persistent, even if the property does not have any mapping metadata associated with it. This is great for fast prototyping of your persistent objects, especially when your persistence vendor supports autotable generation. However, you may have properties that you don't want to be persistent, and therefore this default behavior is inappropriate. For instance, let's assume we want to express what an employee is currently doing without tying this information to persistent storage. We may very simply declare:

```
/**
 * Description of what the Employee's currently
 * working on.  We don't need to store this in the DB.
 */
@Transient
// Don't persist this
private String currentAssignment;
```

When you annotate a property with `@javax.persistence.Transient`, the persistence manager ignores it and doesn't treat it as persistent.

@Basic and FetchType

The `@Basic` annotation is the simplest form of mapping for a persistent property. This is the default mapping type for properties that are primitives, primitive wrapper types, `java.lang.String`, `byte[]`, `Byte[]`, `char[]`, `Character[]`, `java.math.BigInteger`, `java.math.BigDecimal`, `java.util.Date`, `java.util.Calendar`, `java.sql.Date`, `java.sql.Time`, and `java.sql.Timestamp`. You do not need to tell your persistence manager

explicitly that you're mapping a basic property, because it can usually figure out how to map it to JDBC using the property's type:

```
public @interface Basic
{
   FetchType fetch() default EAGER;
   boolean optional() default true;
}

public enum FetchType
{
   LAZY, EAGER
}
```

Usually, you would never annotate your properties with this annotation. However, at times you may need to specify the `fetch()` attribute, which allows you to specify whether a particular property is loaded lazily or eagerly when the persistent object is first fetched from the database. This attribute allows your persistence provider to optimize your access to the database by minimizing the amount of data you load with a query. So, if the `fetch()` attribute is `LAZY`, that particular property will not be initialized until you actually access this field. All other mapping annotations have this same attribute. The weird thing about the specification, though, is that the `fetch()` attribute is just a hint. Even if you mark the property as `LAZY` for a `@Basic` type, the persistence provider is still allowed to load the property eagerly. This is due to the fact that this feature requires class-level instrumentation. It should also be noted that lazy loading is neither really useful nor a significant performance optimization for standard, small objects, as loading these later requires the overhead of more SQL queries. It is best practice to eagerly load basic properties, and lazily load ones that may be large and infrequently accessed.

The `optional()` attribute is useful for when the persistence provider is generating the database schema for you. When this attribute is set to `true`, the property is treated as nullable.

Assuming employees may have a picture associated with their record, we probably don't need this picture all the time. We may lazily load it and allow the property to be nullable:

```
/**
 * Picture of the employee used in ID cards.
 */
@Lob
// Note that this is a binary large object
@Basic(fetch = FetchType.LAZY, optional = true)
// Don't load this by default; it's an expensive operation.
// Only load when requested.
private byte[] image;
```

Because we have a direct byte value here, we must also provide a mapping via the `@Lob` annotation.

@Lob

Sometimes your persistent properties require a lot of memory. One of your fields may represent an image or the text of a very large document. JDBC has special types for these very large objects. The `java.sql.Blob` type represents binary data, and `java.sql.Clob` represents character data. The `@javax.persistence.Lob` annotation is used to map these large object types. Java Persistence allows you to map some basic types to an `@Lob` and have the persistence manager handle them internally as either a `Blob` or a `Clob`, depending on the type of the property:

```
package javax.persistence;

public @interface Lob
{
}
```

Properties annotated with `@Lob` are persisted in a:

- `Blob` if the Java type is `byte[]`, `Byte[]`, or `java.io.Serializable`
- `Clob` if the Java type is `char[]`, `Character[]`, or `java.lang.String`

@Temporal

The `@Temporal` annotation provides additional information to the persistence provider about the mapping of a `java.util.Date` or `java.util.Calendar` property. This annotation allows you to map these object types to a date, a time, or a timestamp field in the database. By default, the persistence provider assumes that the temporal type is a timestamp:

```
package javax.persistence;

public enum TemporalType
{
   DATE,
   TIME,
   TIMESTAMP
}

public @interface Temporal
{
   TemporalType value() default TIMESTAMP;
}
```

For example, say we want to add the date of hire for an employee. We could add this support like so:

```
/**
 * Date the employee joined the company
 */
@Temporal(TemporalType.DATE)
// Note that we should map this as an SQL Date field;
```

```
// could also be SQL Time or Timestamp
private Date since;
```

The `since` property is stored in the database as a `DATE` SQL type.

@Enumerated

The `@Enumerated` annotation maps Java `enum` types to the database.

```
package javax.persistence;

public enum EnumType
{
   ORDINAL,
   STRING
}

public @interface Enumerated
{
    EnumTypes value() default ORDINAL;
}
```

A Java `enum` property can be mapped either to the string representation or to the numeric ordinal number of the `enum` value. For example, let's say we want an `Employee` entity property that designates the kind of employee: manager or peon. This could be represented in a Java `enum` like so:

```
public enum EmployeeType {
   MANAGER, PEON;
}

@Entity
public class Employee
{
   /**
    * Type of employee
    */
   @Enumerated(EnumType.STRING)
   // Show that this is an enumerated value, and the value to
   // be put in the DB is the value of the enumeration toString().
   private EmployeeType type;
}
```

You are not required to use the `@Enumerated` annotation to map a property. If you omit this annotation, the `ORDINAL EnumType` value is assumed.

@Embedded Objects

The Java Persistence specification allows you to embed nonentity Java objects within your entity beans and map the properties of this embedded value object to columns within the entity's table. These objects do not have any identity, and they are owned exclusively by their containing entity bean class. The rules are very similar to the

@EmbeddedId primary-key example given earlier in this chapter. We first start out by defining our embedded object:

```
@Embeddable
public class Address implements java.io.Serializable
{
   private String street;
   private String city;
   private String state;

   public String getStreet() { return street; }
   public void setStreet(String street) { this.street = street; }
   public String getCity() { return city; }
   public void setCity(String city) { this.city = city; }
   public String getState() { return state; }
   public void setState(String state) { this.state = state; }
}
```

The embedded Address class has the @Column mappings defined directly within it. Next, let's use the @javax.persistence.Embedded annotation within our Employee bean class to embed an instance of this Address class:

```
package javax.persistence;

public @interface Embedded {}
```

As with @EmbeddedId, the @Embedded annotation can be used in conjunction with the @AttributeOverrides annotation if you want to override the column mappings specified in the embedded class. The following example shows how this overriding is done. If you don't want to override, leave out the @AttributeOverrides:

```
@Entity
@Table(name="table_employees"
public class Customer implements java.io.Serializable
{
   ...
   private Address address;
   ...
   @Embedded
 @AttributeOverrides({
  @AttributeOverride (name="street", column=@Column(name="employee_street")),
  @AttributeOverride(name="city", column=@Column(name="employee_city")),
  @AttributeOverride(name="state", column=@Column(name="employee_state"))
 })
   public Address getAddress() {
      return address;
   }
...
}
```

In this example, we map the `Address` class properties to columns in `table_employees`. If you do not specify the `@Embedded` annotation and the `Address` class is serializable, then the persistence provider would assume that this was a `@Lob` type and serialize it as a byte stream to the column.

That's about it for basic property mappings. In the next chapter, we discuss how to map complex relationships between entity beans.

CHAPTER 11

Entity Relationships

Chapter 10 covered basic persistence mappings, including various ways to define primary keys as well as simple and complex property-type mappings. This chapter retools our employee registry a bit further by discussing the relationships between entities.

In order to model real-world business concepts, entity beans must be capable of forming relationships. For instance, an employee may have an address; we'd like to form an association between the two in our database model. The address could be queried and cached like any other entity, yet a close relationship would be forged with the `Employee` entity. Entity beans can also have one-to-many, many-to-one, and many-to-many relationships. For example, the `Employee` entity may have many phone numbers, but each phone number belongs to only one employee (a one-to-many relationship). Similarly, an employee may belong to many teams within his or her organization, and teams may have any number of employees (a many-to-many relationship).

The Seven Relationship Types

Seven types of relationships can exist between entity beans. There are four types of cardinality: *one-to-one*, *one-to-many*, *many-to-one*, and *many-to-many*. In addition, each relationship can be either *unidirectional* or *bidirectional*. These options seem to yield eight possibilities, but if you think about it, you'll realize that one-to-many and many-to-one bidirectional relationships are actually the same thing. Thus, there are only seven distinct relationship types. To understand relationships, it helps to think about some simple examples:

One-to-one unidirectional
: The relationship between an employee and an address. You clearly want to be able to look up an employee's address, but you probably don't care about looking up an address's employee.

One-to-one bidirectional
: The relationship between an employee and a computer. Given an employee, we'll need to be able to look up the computer ID for tracing purposes. Assuming the

computer is in the tech department for servicing, it's also helpful to locate the employee when all work is completed.

One-to-many unidirectional
: The relationship between an employee and a phone number. An employee can have many phone numbers (business, home, cell, etc.). You might need to look up an employee's phone number, but you probably wouldn't use one of those numbers to look up the employee.

One-to-many bidirectional
: The relationship between an employee (manager) and direct reports. Given a manager, we'd like to know who's working under him or her. Similarly, we'd like to be able to find the manager for a given employee. (Note that a many-to-one bidirectional relationship is just another perspective on the same concept.)

Many-to-one unidirectional
: The relationship between a customer and his or her primary employee contact. Given a customer, we'd like to know who's in charge of handling the account. It might be less useful to look up all the accounts a specific employee is fronting, although if you want this capability you can implement a many-to-one bidirectional relationship.

Many-to-many unidirectional
: The relationship between employees and tasks to be completed. Each task may be assigned to a number of employees, and employees may be responsible for many tasks. For now we'll assume that given a task we need to find its related employees, but not the other way around. (If you think you need to do so, implement it as a bidirectional relationship.)

Many-to-many bidirectional
: The relationship between an employee and the teams to which he or she belongs. Teams may also have many employees, and we'd like to do lookups in both directions.

Note that these relations represent the navigability of your domain model. Using JPA QL or the Criteria API (covered in Chapter 13), you'll be able to return even an unmapped association (for example, return the tasks for a given employee even if the association has been mapped as many-to-one unidirectional). Once again, the associations defined in the metadata represent the domain object navigation only.

In this chapter, we discuss how to specify relationships by applying annotations to your related entity beans. We also discuss several different common database schemas, and you will learn how to map them to your annotated relationships.

One-to-One Unidirectional Relationship

An example of a one-to-one unidirectional relationship is one between our `Employee` entity and an `Address`. In this instance, each employee has exactly one address, and each address has exactly one employee. Which bean references which determines the direction of navigation. While the `Employee` has a reference to the `Address`, the `Address` doesn't reference the `Employee`. The relationship is therefore unidirectional; you can only go from the employee to the address, not the other way around through object navigation. In other words, an `Address` entity has no idea who owns it. Figure 11-1 shows this relationship.

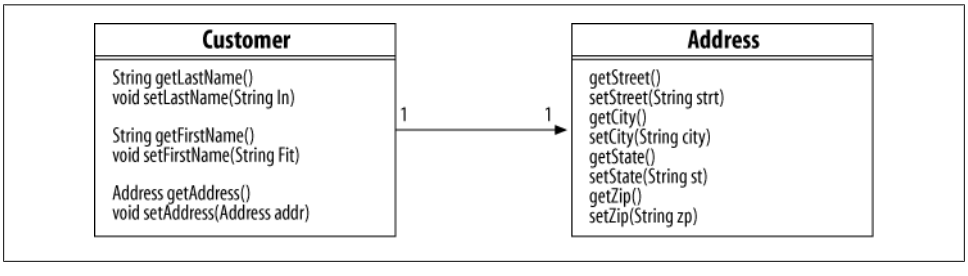


Figure 11-1. One-to-one unidirectional relationship

Relational database schema

One-to-one unidirectional relationships normally use a fairly typical relational database schema in which one table contains a foreign key (pointer) to another table. In this case, the `Employee` table contains a foreign key to the `Address` table, but the `Address` table doesn't contain a foreign key to the `Employee` table. This allows records in the `Address` table to be shared by other tables, a scenario explored in "Many-to-Many Unidirectional Relationship" on page 187.

Programming model

In unidirectional relationships (navigated only one way), one of the entity beans defines a property that lets it get or set the other bean in the relationship. Thus, inside the `Employee` class, you can call the `getAddress()`/`setAddress()` methods to access the Address entity, but there are no methods inside the `Address` class to access the `Employee`. Let's look at how we would mark up the `Employee` bean class to implement this one-to-one relationship to `Address`:

```
/**
 * The employee's address
 */
@OneToOne
@JoinColumn(name="ADDRESS_ID")
// Unidirectional relationship
private Address address;
```

A one-to-one relationship is specified using the `@javax.persistence.OneToOne` annotation and is mapped with the `@javax.persistence.JoinColumn` annotation. Let's first look at the `@JoinColumn` annotation:

```
package javax.persistence;

@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface JoinColumn
{
    String name() default "";
    String referencedColumnName() default "";
    boolean unique() default false;
    boolean nullable() default true;
    boolean insertable() default true;
    boolean updatable() default true;
    String columnDefinition() default "";
    String table() default "";
}
```

The `@JoinColumn` annotation is analogous to the `@Column` annotation. It defines the column in the `Employee`'s table that references the primary key of the `Address` table in the schema. If you are joining on something other than the primary-key column of the `Address` table, then you must use the `referencedColumnName()` attribute. This `referencedColumnName()` must be unique, since this is a one-to-one relationship.

If you need to map a one-to-one relationship in which the related entity has a composite primary key, use the `@JoinColumns` annotation to define multiple foreign-key columns:

```
public @interface @JoinColumns
{
   JoinColumn[] value();
}
```

Now let's learn about the `@OneToOne` annotation:

```
package javax.persistence;

@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface OneToOne
{
    Class targetEntity() default void.class;
    CascadeType[] cascade() default {};
    FetchType fetch() default EAGER;
    boolean optional() default true;
    String mappedBy() default "";
    boolean orphanRemoval() default false;
}
```

The `targetEntity()` attribute represents the entity class you have a relationship to. Usually, you do not have to initialize this attribute, as the persistence provider can figure out the relationship you are setting up from the property's type.

The `fetch()` attribute works the same as we described in Chapter 10. It allows you to specify whether you want the association to be lazily or eagerly loaded. In Chapter 13, we'll show you how you can eagerly fetch a relationship with JPA QL or the Criteria API, even when you have marked the `FetchType` as `LAZY`.

The `optional()` attribute specifies whether this relationship can be null. If this is set to `false`, then a non-null relationship must exist between the two entities.

The `cascade()` attribute is a bit complicated. We'll discuss it later in this chapter, as all relationship types have this attribute.

The `mappedBy()` attribute is for bidirectional relationships and is discussed in the next section.

The `orphanRemoval()` attribute is new to JPA 2.0, and defines whether removing the relationship should also result in a removal of the referred entity.

Primary-key join columns

Sometimes the primary keys of the two related entities are used instead of a specific join column. In this case, the primary keys of the related entities are identical, and there is no need for a specific join column.

In this mapping scenario, you are required to use an alternative annotation to describe the mapping—`@javax.persistence.PrimaryKeyJoinColumn`:

```
public @interface PrimaryKeyJoinColumn
{
   String name() default "";
   String referencedColumnName() default "";
   String columnDefinition() default "";
}
```

The `name()` attribute refers to the primary-key column name of the entity the annotation is applied to. Unless your entity has a composite primary key, you can leave this blank and the persistence provider will figure it out.

The `referencedColumnName()` is the column to join to on the related entity. If this is left blank, it is assumed that the related entity's primary key will be used.

The `columnDefinition()` is used when the persistence provider is generating schema, and its value will specify the SQL type of the `referencedColumnName()`.

If the primary-key join in question is of a composite nature, then the `@javax.persistence.PrimaryKeyJoinColumns` annotation is available to you:

```
public @interface PrimaryKeyJoinColumns
{
   PrimaryKeyJoinColumn[] value();
}
```

So, we could use this annotation to map the `Employee`/`Address` entities' one-to-one relationship:

```
@OneToOne(cascade={CascadeType.ALL})
@PrimaryKeyJoinColumn
private Address address;
```

Since we're joining on the primary keys of the `Employee` and `Address` entities and they are not composite keys, we can simply annotate the address property of `Employee` with the defaulted `@PrimaryKeyJoinColumn` annotation.

Default relationship mapping

If your persistence provider supports auto schema generation, you do not need to specify metadata such as `@JoinColumn` or `@PrimaryKeyJoinColumn`. Auto schema generation is great when you are doing fast prototypes:

```
@OneToOne
private Address address;
```

When you do not specify any database mapping for a unidirectional one-to-one relationship, the persistence provider will generate the necessary foreign-key mappings for you. In our employee/address relationship example, the following tables would be generated:

```
CREATE TABLE "PUBLIC"."EMPLOYEE"
(
   ID bigint PRIMARY KEY NOT NULL,
   ADDRESS_ID bigint
)
;

ALTER TABLE "PUBLIC"."EMPLOYEE"
ADD CONSTRAINT FK4AFD4ACEE5310533
FOREIGN KEY (ADDRESS_ID)
REFERENCES "PUBLIC"."ADDRESS"(ADDRESS_ID)
;
```

For unidirectional one-to-one relationships, the default mapping creates a foreign-key column named from a combination of the property you are mapping followed by an _ (underscore) character concatenated with the primary-key column name of the referenced table.

One-to-One Bidirectional Relationship

We can expand our `Employee` entity to include a reference to a `Computer` entity, which models the associate's company-provided computer. The employee will maintain a reference to his or her computer, and the computer will maintain a reference back to the employee. This makes good sense, since we may need to know the owner of a computer.

Relational database schema

The `Computer` has a corresponding `COMPUTER` table, which will contain a pointer to its `Employee` owner:

```
CREATE TABLE "PUBLIC"."COMPUTER"
(
   ID bigint PRIMARY KEY NOT NULL,
   MAKE varchar,
   MODEL varchar,
   OWNER_ID bigint
)
;
ALTER TABLE "PUBLIC"."COMPUTER"
ADD CONSTRAINT FKE023E33B5EAFBFC
FOREIGN KEY (OWNER_ID)
REFERENCES "PUBLIC"."EMPLOYEE"(OWNER_ID)
;
```

One-to-one bidirectional relationships may model relational database schemas in the same way as our one-to-one unidirectional relationship, in which one of the tables holds a foreign key that references the other. Remember that in a relational database model, there is no such notion of directionality, so the same database schema will be used for both unidirectional and bidirectional object relationships.

To model the relationship between the `Employee` and `Computer` entities, we need to declare a relationship property named `owner` in the `Computer` bean class:

```
@Entity
public class Computer
{
  ...
   @OneToOne
   // Bidirectional relationship, mappedBy
   // is declared on the non-owning side
   private Employee owner;
  ...
}
```

Similarly, the `Employee` class will have a reference to the `Computer`:

```
/**
 * The employee's computer
 */
@OneToOne(mappedBy = "owner")
// Bidirectional relationship
private Computer computer;
```

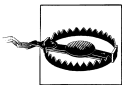
The `mappedBy()` attribute is new here. This attribute sets up the bidirectional relationship and tells the persistence manager that the information for mapping this relationship to our tables is specified in the `Computer` bean class, specifically to the `owner` property of `Computer`.

Here is an example for setting up a bidirectional relationship:

```
// Create a new Computer
final Computer computer = new Computer();
computer.setMake("Computicorp");
computer.setModel("ZoomFast 100");

// Create a new Employee
final Employee carloDeWolf = new Employee("Carlo de Wolf");

// Persist; now we have managed objects
EntityManager em = null;  // Assume we have this
em.persist(carloDeWolf);
em.persist(computer);

// Associate *both* sides of a bidirectional relationship
carloDeWolf.setComputer(computer);
computer.setOwner(carloDeWolf);
```

We have the `cascade()` attribute set to empty, so we must perform the association after each object is persisted and therefore managed. When we discuss cascading operations, you will see that there are ways to persist unmanaged objects as part of an association automatically.

There are some peculiarities with bidirectional relationships. With all bidirectional relationship types, including one-to-one, there is always the concept of an *owning* side of the relationship. Although a `setOwner()` method is available in the `Computer` bean class, it will not cause a change in the persistent relationship if we set it. When we marked the `@OneToOne` relationship in the `Employee` bean class with the `mappedBy()` attribute, this designated the `Employee` entity as the *inverse* side of the relationship. This means that the `Computer` entity is the *owning* side of the relationship.



Always wire both sides of a bidirectional relationship when modifying relationships. Entities are like any other Java object that has an association to another object. You have to set the values of both sides of the relationship in memory for the relationship to be updated.

If the employee broke his computer, you would have to set the relationship to null on both sides and then remove the `Computer` entity from the database.

One-to-Many Unidirectional Relationship

Entity beans can also maintain relationships with multiplicity. This means one entity bean can aggregate or contain many other entity beans. For example, an employee may have relationships with many phones, each of which represents a phone number. This is very different from simple one-to-one relationships—or, for that matter, from multiple one-to-one relationships with the same type of bean. One-to-many and many-to-many relationships require the developer to work with a collection of references instead of a single reference when accessing the relationship field.

Relational database schema

To illustrate a one-to-many unidirectional relationship, we will use a new entity bean, the Phone, for which we must define a table, the PHONE table:

```
CREATE TABLE "PUBLIC"."PHONE"
(
   ID bigint PRIMARY KEY NOT NULL,
   NUMBER varchar,
   TYPE varchar
)
;
```

This is modeled as a simple Phone class:

```
@Entity
public class Phone
{
   ...
   /**
    * Phone number
    */
   private String number;
   ...
}
```

One-to-many unidirectional relationships between the EMPLOYEE and PHONE tables could be implemented in a variety of ways. For this example, we have chosen to introduce a new table, a join table, to map phones to employees:

```
CREATE TABLE "PUBLIC"."EMPLOYEE_PHONE"
(
   EMPLOYEE_ID bigint NOT NULL,
   PHONES_ID bigint NOT NULL
)
;
ALTER TABLE "PUBLIC"."EMPLOYEE_PHONE"
ADD CONSTRAINT FK1E56289D581FE7C
FOREIGN KEY (PHONES_ID)
REFERENCES "PUBLIC"."PHONE"(PHONES_ID)
;
ALTER TABLE "PUBLIC"."EMPLOYEE_PHONE"
ADD CONSTRAINT FK1E56289DD9454221
FOREIGN KEY (EMPLOYEE_ID)
REFERENCES "PUBLIC"."EMPLOYEE"(EMPLOYEE_ID)
;
```

Likewise, we can introduce a link from Employee to obtain a collection of Phones:

```
/**
 * All {@link Phone}s for this {@link Employee}
 */
@OneToMany
// Unidirectional relationship
private Collection<Phone> phones;
```

Our database schema illustrates that the structure and relationships of the actual database can differ from the relationships as defined in the programming model. In this case, the relationship in the object model is inferred from the reference `Employee` has to a `Collection` of `Phones`, but the database contains a join table. When you are dealing with legacy databases (i.e., databases that were established before the EJB application), it's important to have the mapping options JPA affords so that the object model is not dictated by the schema.

Programming model

You declare one-to-many relationships using the `@javax.persistence.OneToMany` annotation:

```
package javax.persistence;

@Target({METHOD, FIELD})
@Retention(RUNTIME)
public @interface OneToMany
{
    Class targetEntity() default void.class;
    CascadeType[] cascade() default {};
    FetchType fetch() default LAZY;
    String mappedBy() default "";
    boolean orphanRemoval() default false;
}
```

The attribute definitions are pretty much the same as those for the `@OneToOne` annotation.

In the programming model, we represent multiplicity by defining a relationship property that can point to many entity beans and annotating it with `@OneToMany`. To hold this type of data, we'll employ some data structures from the `java.util` package: `Collection`, `List`, `Map`, and `Set`. The `Collection` maintains a homogeneous group of entity object references, which means that it contains many references to one kind of entity bean. The `Collection` type may contain duplicate references to the same entity bean, and the `Set` type may not.

To illustrate how an entity bean uses a collection-based relationship, let's look at some code that interacts with the `EntityManager`:

```
// Create an Employee
final Employee jaikiranPai = new Employee("Jaikiran Pai");

// Create a couple Phones
final Phone phone1 = new Phone("800-USE-EJB3");
final Phone phone2 = new Phone("8675309");

// Persist
final EntityManager em = null; // Assume we have this
em.persist(jaikiranPai);
em.persist(phone1);
em.persist(phone2);
```

```
// Associate
jaikiranPai.getPhones().add(phone1);
jaikiranPai.getPhones().add(phone2);
```

If you need to remove a `Phone` from the relationship, you need to remove the `Phone` from both the collection and the database:

```
jaikiranPai.getPhones().remove(phone1);
em.remove(phone1);
```

Removing the `Phone` from the `Employee`'s collection does not remove the `Phone` from the database. You have to delete the `Phone` explicitly; otherwise, it will be orphaned. The `orphanRemoval` attribute of `@OneToMany` may remove the orphaned phone automatically if set to `true`.

Many-to-One Unidirectional Relationship

Many-to-one unidirectional relationships result when many entity beans reference a single entity bean, but the referenced entity bean is unaware of the relationship. In our example company, we have a series of `Customer`s who are each assigned to an `Employee` who is the primary contact for the customer account. An `Employee` may be the point person for many `Customer`s, but the relationship will be one-way; customers will know who their primary contact is, but `Employee`s will not refer to all of their `Customer`s.

Relational database schema

Here we introduce a `CUSTOMER` table to model our customers:

```
CREATE TABLE "PUBLIC"."CUSTOMER"
(
   ID bigint PRIMARY KEY NOT NULL,
   NAME varchar,
   PRIMARYCONTACT_ID bigint
)
;
ALTER TABLE "PUBLIC"."CUSTOMER"
ADD CONSTRAINT FK27FBE3FE9BACAAF1
FOREIGN KEY (PRIMARYCONTACT_ID)
REFERENCES "PUBLIC"."EMPLOYEE"(PRIMARYCONTACT_ID)
;
```

Programming model

Many-to-one relationships are described with the `@javax.persistence.ManyToOne` annotation:

```
public @interface ManyToOne
{
   Class targetEntity() default void.class;
   CascadeType[] cascade() default {};
   FetchType fetch() default EAGER;
```

```
    boolean optional() default true;
}
```

The attribute definitions are pretty much the same as those for the `@OneToOne` annotation.

The programming model is quite simple for our relationship. We add a `primaryContact` property to our `Customer` entity bean class and annotate it with the `@ManyToOne` annotation:

```
@Entity
public class Customer
{
   ...
   /**
    * The primary {@link Employee} contact for this {@link Customer}
    */
   @ManyToOne
   // Unidirectional
   private Employee primaryContact;
   ...
}
```

The relationship between the `Employee` and `Customer` entities is unidirectional, so the `Employee` bean class doesn't define any relationship back to the `Customer`.

All of this should be mundane to you now. The impact of exchanging `Employee` references between `Customer` entities works just as we've seen with the previous relationship types.

One-to-Many Bidirectional Relationship

One-to-many and many-to-one bidirectional relationships sound like they're different, but they're not. A one-to-many bidirectional relationship occurs when one entity bean maintains a collection-based relationship property with another entity bean, and each entity bean referenced in the collection maintains a single reference back to its aggregating bean. For example, in our company an employee has a manager, and likewise a manager has many direct reports. The relationship is a one-to-many bidirectional relationship from the perspective of the manager (an `Employee`) and a many-to-one bidirectional relationship from the perspective of the report (also an `Employee`). For fun's sake, let's label "reports" in this sense as "peons." This becomes an interesting exercise as well, because it shows that relationships may exist within a single entity type.

Relational database schema

First, we need to equip our `EMPLOYEE` table with the support necessary to model the manager/peon relationship:

```
CREATE TABLE "PUBLIC"."EMPLOYEE"
(
   ID bigint PRIMARY KEY NOT NULL,
```

```
    NAME varchar,
    MANAGER_ID bigint
)
;
ALTER TABLE "PUBLIC"."EMPLOYEE"
ADD CONSTRAINT FK4AFD4ACE378204C2
FOREIGN KEY (MANAGER_ID)
REFERENCES "PUBLIC"."EMPLOYEE"(MANAGER_ID)
;
```

Now each `Employee` has a reference to his or her manager.

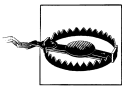
Programming model

Because this is a bidirectional relationship, the manager knows his or her reports, and also his or her own manager. The `Employee` class may therefore contain:

```
/**
 * Manager of the {@link Employee}
 */
@ManyToOne
private Employee manager;

/**
 * {@link Employee}s reporting to this {@link Employee}
 */
@OneToMany(mappedBy = "manager")
private Collection<Employee> peons;
```

The entire relationship here is contained within the `Employee` class. As with all bidirectional relationships, the inverse side specifies `mappedBy` to indicate the property to act as the owning side.

Java Persistence currently requires that the many-to-one side always be the owner. This may seem very confusing, but if you obey the cardinal rule of always wiring both sides of a relationship, then you will have no problems.



Always wire both sides of a bidirectional relationship in your Java code.

Usage

Client usage is as we've seen before:

```
// Create a few Employees
final Employee alrubinger = new Employee("Andrew Lee Rubinger");
final Employee carloDeWolf = new Employee("Carlo de Wolf - SuperLead!");
final Employee jaikiranPai = new Employee("Jaikiran Pai");
final Employee bigD = new Employee("Big D");

// Persist
em.persist(jaikiranPai);
```

```
em.persist(alrubinger);
em.persist(carloDeWolf);
em.persist(bigD);

// Associate *both* sides of the bidirectional relationship
final Collection<Employee> peonsOfD = bigD.getPeons();
peonsOfD.add(alrubinger);
peonsOfD.add(carloDeWolf);
peonsOfD.add(jaikiranPai);
alrubinger.setManager(bigD);
carloDeWolf.setManager(bigD);
jaikiranPai.setManager(bigD);
```

Many-to-Many Bidirectional Relationship

Many-to-many bidirectional relationships occur when many beans maintain a collection-based relationship property with another bean, and each bean referenced in the collection maintains a collection-based relationship property back to the aggregating beans. For example, in our example company every `Employee` may belong to many `Team`s, and each `Team` may be composed of many `Employees`.

Relational database schema

The `EMPLOYEE` and `TEAM` tables may be fairly simple, and neither will have a direct reference to the other:

```
CREATE TABLE "PUBLIC"."TEAM"
(
   ID bigint PRIMARY KEY NOT NULL,
   NAME varchar
)
;
CREATE TABLE "PUBLIC"."EMPLOYEE"
(
   ID bigint PRIMARY KEY NOT NULL,
   NAME varchar
)
;
```

Again, we use a join table to establish a many-to-many bidirectional relationship, and we'll call this the `TEAM_EMPLOYEE` table. Here we maintain two foreign key columns—one for the `EMPLOYEE` table and another for the `TEAM` table:

```
CREATE TABLE "PUBLIC"."TEAM_EMPLOYEE"
(
   TEAMS_ID bigint NOT NULL,
   MEMBERS_ID bigint NOT NULL
)
;
ALTER TABLE "PUBLIC"."TEAM_EMPLOYEE"
ADD CONSTRAINT FKA63C2502B25E948
FOREIGN KEY (TEAMS_ID)
REFERENCES "PUBLIC"."TEAM"(TEAMS_ID)
```

```
;
ALTER TABLE "PUBLIC"."TEAM_EMPLOYEE"
ADD CONSTRAINT FKA63C25052E6C3D6
FOREIGN KEY (MEMBERS_ID)
REFERENCES "PUBLIC"."EMPLOYEE"(MEMBERS_ID)
```

Programming model

Many-to-many relationships are logically defined using the `@javax.persistence.ManyToMany` annotation:

```
public @interface ManyToMany
{
   Class targetEntity( ) default void.class;
   CascadeType[] cascade( ) default {};
   FetchType fetch( ) default LAZY;
   String mappedBy( ) default "";
}
```

To model the many-to-many bidirectional relationship between the `Employee` and `Team` entities, we need to include collection-based relationship properties in both bean classes:

```
@Entity
public class Employee
{
    ...
   /**
    * The {@link Team}s to which this {@link Employee} belongs
    */
   @ManyToMany(mappedBy = "members")
   private Collection<Team> teams;
    ...
}

@Entity
public class Team
{
   ...
   /**
    * {@link Employee}s on this {@link Task}.
    */
   @ManyToMany
   private Collection<Employee> members;
   ...
}
```

The relationship is declared as a `java.util.Collection`. We could also use a `Set` type, which would contain only unique `Team`s and no duplicates. The effectiveness of the `Set` collection type depends largely on referential-integrity constraints established in the underlying database.

As with all bidirectional relationships, there has to be an owning side. In this case, it is the `Team` entity. Since the `Team` owns the relationship, its bean class may have the

`@JoinTable` mapping, though in our example we can accept the defaults. The `joinColumns()` attribute could identify the foreign-key column in the `TEAM_EMPLOYEE` table that references the `TEAM` table, whereas the `inverseJoinColumns()` attribute could identify the foreign key that references the `EMPLOYEE` table.

Like with `@OneToMany` relationships, if you are using your persistence provider's auto schema generation facilities, you do not need to specify a `@JoinTable` mapping. The Java Persistence specification has a default mapping for `@ManyToMany` relationships and will create the join table for you.

As with one-to-many bidirectional relationships, the `mappedBy()` attribute identifies the property on the `Team` bean class that defines the relationship. This also identifies the `Employee` entity as the inverse side of the relationship.

As far as modifying and interacting with the relationship properties, the same ownership rules apply as we saw in the one-to-many bidirectional example:

```
// Create a few employees
final Employee pmuir = new Employee("Pete Muir");
final Employee dallen = new Employee("Dan Allen");
final Employee aslak = new Employee("Aslak Knutsen");

// Create some teams
final Team seam = new Team("Seam");
final Team arquillian = new Team("Arquillian");

// Get EM
final EntityManager em = null; // Assume we have this

// Persist
em.persist(pmuir);
em.persist(dallen);
em.persist(aslak);
em.persist(seam);
em.persist(arquillian);

// Associate *both* directions
seam.getMembers().add(dallen);
seam.getMembers().add(pmuir);
seam.getMembers().add(aslak);
arquillian.getMembers().add(dallen);
arquillian.getMembers().add(pmuir);
arquillian.getMembers().add(aslak);
aslak.getTeams().add(seam);
aslak.getTeams().add(arquillian);
dallen.getTeams().add(seam);
dallen.getTeams().add(arquillian);
pmuir.getTeams().add(seam);
pmuir.getTeams().add(arquillian);
```

Many-to-Many Unidirectional Relationship

Many-to-many unidirectional relationships occur when many beans maintain a collection-based relationship with another bean, but the bean referenced in the `Collection` does not maintain a collection-based relationship back to the aggregating beans. In our example, we may assign any number of `Task`s to any number of `Employee`s, and `Employee`s may be assigned to any number of `Task`s. We'll maintain a reference from `Task` to `Employee`, but not the other way around.

Relational database schema

Our first order of business is to declare a `TASK` table:

```
CREATE TABLE "PUBLIC"."TASK"
(
   ID bigint PRIMARY KEY NOT NULL,
   DESCRIPTION varchar
)
;
```

Again, we'll make a join table to map `Task`s to `Employee`s:

```
CREATE TABLE "PUBLIC"."TASK_EMPLOYEE"
(
   TASK_ID bigint NOT NULL,
   OWNERS_ID bigint NOT NULL
)
;
ALTER TABLE "PUBLIC"."TASK_EMPLOYEE"
ADD CONSTRAINT FK7B1EDC2832EBEA41
FOREIGN KEY (TASK_ID)
REFERENCES "PUBLIC"."TASK"(TASK_ID)
;
ALTER TABLE "PUBLIC"."TASK_EMPLOYEE"
ADD CONSTRAINT FK7B1EDC28A3E4776F
FOREIGN KEY (OWNERS_ID)
REFERENCES "PUBLIC"."EMPLOYEE"(OWNERS_ID)
;
```

This many-to-many unidirectional relationship looks a lot like the join table mapping for the many-to-many bidirectional relationship discussed earlier. The big difference is that the object model will maintain a reference only in one direction.

Programming model

To model this relationship, we need to add a collection-based relationship field for `Employee` beans to the `Task`:

```
@Entity
public class Task
{
   ...
   /**
    * {@link Employee} in charge of this {@link Task}
```

```
     */
    @ManyToMany
    private Collection<Employee> owners;
    ...
}
```

Because the relationship is unidirectional, there are no owning or inverse sides, and we may omit the `mappedBy` attribute of `@ManyToMany`.

Usage is similar to what we've already seen:

```
// Create a couple of employees
final Employee smarlow = new Employee("Scott Marlow");
final Employee jpederse = new Employee("Jesper Pedersen");

// Create a couple of tasks
final Task task1 = new Task("Go to the Java User's Group in Boston");
final Task task2 = new Task("Help Shelly McGowan with testsuite setup");

// Persist
final EntityManager em = null; // Assume we have this
em.persist(smarlow);
em.persist(jpederse);
em.persist(task1);
em.persist(task2);

// Associate
task1.getOwners().add(smarlow);
task1.getOwners().add(jpederse);
task2.getOwners().add(smarlow);
task2.getOwners().add(jpederse);
```

Mapping Collection-Based Relationships

The one-to-many and many-to-many examples we've seen so far have used the `java.util.Collection` and `java.util.Set` types. The Java Persistence specification also allows you to represent a relationship with a `java.util.List` or a `java.util.Map`.

Ordered List-Based Relationship

The `java.util.List` interface can express collection-based relationships. You do not need any special metadata if you want to use a `List` rather than a `Set` or `Collection` type. (In this case, the `List` actually gives you a bag semantic, an unordered collection that allows duplicates). A `List` type can give you the additional ability to order the returned relationship based on a specific set of criteria. This requires the additional metadata provided by the `@javax.persistence.OrderBy` annotation:

```
package javax.persistence;

@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface OrderBy
{
```

```
    String value( ) default "";
}
```

The `value()` attribute allows you to declare partial JPA QL that specifies how you want the relationship to be ordered when it is fetched from the database. If the `value()` attribute is left empty, the `List` is sorted in ascending order based on the value of the primary key.

Let's take the `Employee`/`Team` relationship, which is a many-to-many bidirectional relationship, and have the `teams` attribute of `Employee` return a `List` that is sorted alphabetically by the `Team` entity's name:

```
@Entity
public class Employee
{
    ...
    @ManyToMany
    @OrderBy("name ASC")
    private List<Team> teams;
    ...
}
```

`"name ASC"` tells the persistence provider to sort the `Team`'s `name` in ascending order. You can use `ASC` for ascending order and `DESC` for descending order. You can also specify additional restrictions, such as `@OrderBy('name asc, otherattribute asc")`. In this case, the list will be ordered by `lastname`, and for duplicate names, it will be ordered by the `otherattribute`.

Map-Based Relationship

The `java.util.Map` interface can be used to express collection-based relationships. In this case, the persistence provider creates a map, with the key being a specific property of the related entity and the value being the entity itself. If you use a `java.util.Map`, you must use the `@javax.persistence.MapKey` annotation:

```
package javax.persistence;

@Target({METHOD, FIELD}) @Retention(RUNTIME)
public @interface MapKey
{
   String name( ) default "";
}
```

The `name()` attribute is the name of the persistent property that you want to represent the key field of the map object. If you leave this blank, it is assumed you are using the primary key of the related entity as the key of the map.

For an example, let's use a map to represent the one-to-many unidirectional `Employee`/`Phone` relationship discussed earlier in this chapter:

```
@Entity
public class Employee
```

```
{
    ...
    @OneToMany
    @MapKey(name="number")
    private Map<String,Phone> phones;
    ...
}
```

In this example, the `phones` property of `Employee` will return a `java.util.Map` where the key is the `number` property of the `Phone` entity and the value is, of course, the `Phone` entity itself. There is no extra column to keep the map key, since the map key is borrowed from the `Phone` entity.

Detached Entities and FetchType

In Chapter 9, we discussed how managed entity instances become detached from a persistence context when the persistence context ends. Since these entity instances are no longer managed by any persistence context, they may have uninitialized properties or relationships. If you are returning these detached entities to your clients and basically using them as data transfer objects between the client and server, you need to fully understand the effects of accessing any uninitialized relationships.

When an entity instance becomes detached, its state might not be fully initialized, because some of its persistent properties or relationships may be marked as lazily loaded in the mapping metadata. Each relationship annotation has a `fetch()` attribute that specifies whether the relationship property is loaded when the entity is queried. If the `fetch()` attribute is set to `FetchType.LAZY`, then the relationship is not initialized until it is traversed in your code:

```
Employee employee = entityManager.find(Employee.class, id);
employee.getPhones().size();
```

Invoking the `size()` method of the `phones` collection causes the relationship to be loaded from the database. It is important to note that this lazy initialization does not happen unless the entity bean is being managed by a persistence context. If the entity bean is detached, the specification is not clear on what actions the persistence provider should perform when accessing an unloaded relationship of a detached entity. Most persistence providers throw some kind of lazy instantiation exception when you call the accessor of the relationship or when you try to invoke an operation on the relationship of a detached entity:

```
Employee employee = entityManager.find(Employee.class, id);
entityManager.detach(employee);
try
{
   int numPhones = employee.getPhones().size();
}
catch (SomeVendorLazyInitializationException ex)
{
}
```

In this code, the application has received an instance of a detached `Employee` entity and attempts to access the `phones` relationship. If the `fetch()` attribute of this relationship is `LAZY`, most vendor implementations will throw a vendor-specific exception. This lazy initialization problem can be overcome in two ways. The obvious way is just to navigate the needed relationships while the entity instance is still managed by a persistence context. The second way is to perform the fetch eagerly when you query the entity. In Chapter 13, you will see that the JPA QL query language has a `FETCH JOIN` operation that allows you to preinitialize selected relationships when you invoke a query.

How is the persistence provider able to throw an exception when accessing the relationship when the `Employee` class is a plain Java class? Although not defined in the specification, the vendor has a few ways to implement this. One is through bytecode manipulation of the `Employee` class. In Java EE, the application server is required to provide hooks for bytecode manipulation for persistence providers. In Java SE, the persistence provider may require an additional post-compilation step on your code base. Another way for vendors to implement this is to create a proxy class that inherits from `Employee` and that reimplements all the accessor methods to add lazy initialization checking. For collection-based relationships, the persistence provider can just provide its own implementation of the collection and do the lazy check there. Whatever the implementation, make a note to discover what your persistence provider will do in the detached lazy initialization scenario so that your code can take appropriate measures to handle this exception.

Cascading

There is one annotation attribute that we have ignored so far: the `cascade()` attribute of the `@OneToOne`, `@OneToMany`, `@ManyToOne`, and `@ManyToMany` relationship annotations. This section discusses in detail the behavior that is applied when using the `cascade()` attribute.

When you perform an entity manager operation on an entity bean instance, you can automatically have the same operation performed on any relationship properties the entity may have. This is called *cascading*. For example, if you are persisting a new `Employee` entity with a new address and phone number, all you have to do is wire the object, and the entity manager can automatically create the employee and its related entities, all in one `persist()` method call:

```
Employee employee = new Employee();
customer.setAddress(new Address());
customer.getPhoneNumbers().add(new Phone());

// create them all in one entity manager invocation
entityManager.persist(employee);
```

With the Java Persistence specification, cascading can be applied to a variety of entity manager operations, including `persist()`, `merge()`, `remove()`, and `refresh()`. This

feature is enabled by setting the javax.persistence.CascadeType of the relationship annotation's cascade() attribute. The CascadeType is defined as a Java enumeration:

```
public enum CascadeType
{
   ALL, PERSIST,
   MERGE, REMOVE,
   REFRESH
}
```

The ALL value represents all of the cascade operations. The remaining values represent individual cascade operations. The cascade() attribute is an array of the cascade operations you want applied to your related entities.

PERSIST

PERSIST has to deal with the creation of entities within the database. If we had a CascadeType of PERSIST on the Employee side of our one-to-one relationship, you would not have to persist your created Address as well. It would be created for you. The persistence provider also will execute the appropriate SQL INSERT statements in the appropriate order for you.

If you did not have a cascade policy of PERSIST, then you would have to call EntityManager.persist() on the address object as well (as we do in the example).

MERGE

MERGE deals with entity synchronization, meaning inserts and, more importantly, updates. These aren't updates in the traditional sense. If you remember from previous chapters, we mentioned that objects could be detached from persistent management and serialized to a remote client, updates could be performed locally on that remote client, the object instance would be sent back to the server, and the changes would be merged back into the database. Merging is about synchronizing the state of a detached object instance back to persistent storage.

So, back to what MERGE means! MERGE is similar to PERSIST. If you have a cascade policy of MERGE, then you do not have to call EntityManager.merge() for the contained related entity:

```
employee.setName("William");
employee.getAddress().setCity("Boston");
entityManager.merge(employee);
```

In this example, when the employee variable is merged by the entity manager, the entity manager will cascade the merge to the contained address property and the city also will be updated in the database.

Another interesting thing about MERGE is that if you have added any new entities to a relationship that have not been created in the database, they will be persisted and created when the merge() happens:

```
Phone phone = new Phone();
phone.setNumber("617-666-6666");

employee.getPhoneNumbers().add(phone);
entityManager.merge(employee);
```

In this example, we allocate a Phone and add it to an Employee's list of phone numbers. We then call merge() with the employee, and since we have the MERGE CascadeType set on this relationship, the persistence provider will see that it is a new Phone entity and will create it within the database.

Remember that only the graph returned by the merge operation is in managed state, not the one passed as a parameter.

REMOVE

REMOVE is straightforward. In our Employee example, if you delete an Employee entity, its address will be deleted as well:

```
Employee employee = entityManager.find(Employee.class, id);
entityManager.remove(employee); // Also removes the address
```

REFRESH

REFRESH is similar to MERGE. Unlike merge, though, this CascadeType only pertains to when EntityManager.refresh() is called. Refreshing doesn't update the database with changes in the object instance. Instead, it refreshes the object instance's state from the database. Again, the contained related entities would also be refreshed:

```
Employee employee = entityManager.find(Employee.class, id);
entityManager.refresh(employee); // address would be refreshed too
```

So, if changes to the Employee's address were committed by a different transaction, the address property of the employee variable would be updated with these changes. This is useful in practice when an entity bean has some properties that are generated by the database (by triggers, for example). You can refresh the entity to read those generated properties. In this case, be sure to make those generated properties read-only from a persistence provider perspective, i.e., using the @Column (insertable=false, updatable=false).

ALL

ALL is a combination of all of the previous policies and is used for the purposes of simplicity.

When to Use Cascading

You don't always want to use cascading for every relationship you have. For instance, you would not want to remove the related `Computer` or `Task`s when removing an `Employee` entity from the database, because these entities have a life span that is usually longer than the employee. You might not want to cascade merges because you may have fetched stale data from the database or simply not filled the relationship in one particular business operation. For performance reasons, you may also not want to refresh all the relationships an entity has because this would cause more round trips to the database. Be aware how your entities will be used before deciding on the cascade type. If you are unsure of their use, then you should turn off cascading entirely and enable it on a per-case basis. Remember, cascading is simply a convenient tool for reducing the `EntityManager` API calls. It's very easy to trigger expensive database calls, which may result in unnecessary multitable joins.

CHAPTER 12
Entity Inheritance

In order to be complete, an object-to-relational mapping engine must support inheritance hierarchies. The Java Persistence specification supports entity inheritance, polymorphic relationships/associations, and polymorphic queries. These features were completely missing in the older EJB CMP 2.1 specification.

In this chapter, we'll modify the `Employee` entity that we defined in earlier chapters to make it fit into an inheritance hierarchy. We'll have it extend a base class called `Person` and redefine the `Employee` class to extend a `Customer` class. Figure 12-1 shows this class hierarchy.

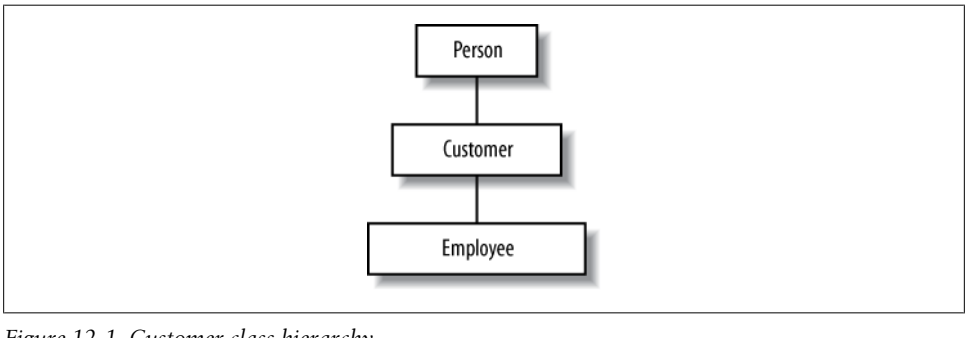


Figure 12-1. Customer class hierarchy

The Java Persistence specification provides three different ways to map an inheritance hierarchy to a relational database:

A single table per class hierarchy
: One table will have all properties of every class in the hierarchy.

A table per concrete class
: Each class will have a table dedicated to it, with all of its properties and the properties of its superclass mapped to this table.

A table per subclass
: Each class will have its own table. Each table will have only the properties that are defined in that particular class. These tables will not have properties of any superclass or subclass.

In this chapter, we use these three strategies to map the `Employee` class hierarchy defined in Figure 12-1.

Single Table per Class Hierarchy

In the single table per class hierarchy mapping strategy, one database table represents every class of a given hierarchy. In our example, the `Person`, `Customer`, and `Employee` entities are represented in the same table, as shown in the following code:

```
CREATE TABLE "PUBLIC"."SINGLECLASS_PERSON"
(
   DISCRIMINATOR varchar NOT NULL,
   ID bigint PRIMARY KEY NOT NULL,
   FIRSTNAME varchar,
   LASTNAME varchar,
   CITY varchar,
   STATE varchar,
   STREET varchar,
   ZIP varchar,
   EMPLOYEEID integer
)
;
```

As you can see, all the properties for the `Customer` class hierarchy are held in one table, `SINGLECLASS_PERSON`. The single table per class hierarchy mapping also requires an additional *discriminator column*. This column identifies the type of entity being stored in a particular row of `SINGLECLASS_PERSON`. Let's look at how the classes will use annotations to map this inheritance strategy:

```
@Entity(name = "SINGLECLASS_PERSON")
@DiscriminatorColumn(name = "DISCRIMINATOR", discriminatorType =
   DiscriminatorType.STRING)
@DiscriminatorValue("PERSON")
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
public class Person
{
   @Id
   @GeneratedValue
   private Long id;

   private String firstName;
   private String lastName;

   ...
}
```

This is our base class for both Customers and Employees. It defines the discriminator column we've just seen as "DISCRIMINATOR", where Person rows will have a value of "PERSON".

Since one table is representing the entire class hierarchy, the persistence provider needs some way to identify which class the row in the database maps to. It determines this by reading the value from the discriminator column. The @javax.persistence.DiscriminatorColumn annotation identifies which column in our table will store the discriminator's value. The name() attribute identifies the name of the column, and the discriminatorType() attribute specifies what type the discriminator column will be. It can be a STRING, CHAR, or INTEGER. For our Employee class hierarchy mapping, you do not have to specify the discriminatorType(), as it defaults to being a STRING. If you're OK with the default column name, you can remove the @DiscriminatorColumn entirely:

```
package javax.persistence;

@Target(TYPE) @Retention(RUNTIME)
public @interface DiscriminatorColumn
{
   String name( ) default "DTYPE";
   DiscriminatorType discriminatorType( ) default STRING;
   String columnDefinition( ) default "";
   int length( ) default 10;
}
```

The @javax.persistence.DiscriminatorValue annotation defines what value the discriminator column will take for rows that store an instance of a Person class. You can leave this attribute undefined if you want to. In that case, the persistence manager would generate a value for you automatically. This value will be vendor-specific if the DiscriminatorType is CHAR or INTEGER. The entity name is used by default when a type of STRING is specified. It is good practice to specify the value for CHAR and INTEGER values:

```
package javax.persistence;

@Target(TYPE) @Retention(RUNTIME)
public @interface DiscriminatorValue
{
   String value( )
}
```

The @javax.persistence.Inheritance annotation is used to define the persistence strategy for the inheritance relationship:

```
package javax.persistence;

@Target(TYPE) @Retention(RUNTIME)
public @interface Inheritance
{
   InheritanceType strategy( ) default SINGLE_TABLE;
}
```

```
public enum InheritanceType
{
   SINGLE_TABLE, JOINED, TABLE_PER_CLASS
}
```

The `strategy()` attribute defines the inheritance mapping that we're using. Since we're using the single table per class hierarchy, the `SINGLE_TABLE` enum is applied.

We can now extend this base into a more specialized `Customer`:

```
@Entity(name = "SINGLECLASS_CUSTOMER")
@DiscriminatorValue("CUSTOMER")
public class Customer extends Person
{
   private String street;
   private String city;
   private String state;
   private String zip;
   ...
}
```

`Customer` rows in the table will use a value of `"CUSTOMER"` in the discriminator column.

Finally, we have `Employee`:

```
@Entity (name="SINGLECLASS_EMPLOYEE")
@DiscriminatorValue("EMPLOYEE")
public class Employee extends Customer
{
   private Integer employeeId;
   ...
}
```

Advantages

The `SINGLE_TABLE` mapping strategy is the simplest to implement and performs better than all the inheritance strategies. There is only one table to administer and deal with. The persistence engine does not have to do any complex joins, unions, or subselects when loading the entity or when traversing a polymorphic relationship, because all data is stored in one table.

Disadvantages

One huge disadvantage of this approach is that all columns of subclass properties must be nullable. So, if you need or want to have any `NOT NULL` constraints defined on these columns, you cannot do so. Also, because subclass property columns may be unused, the `SINGLE_TABLE` strategy is not normalized.

Table per Concrete Class

In the table per concrete class strategy, a database table is defined for each concrete class in the hierarchy. Each table has columns representing its properties and all properties of any superclasses:

```
CREATE TABLE "PUBLIC"."TABLEPERCLASS_PERSON"
(
   ID bigint PRIMARY KEY NOT NULL,
   FIRSTNAME varchar,
   LASTNAME varchar
)
;

CREATE TABLE "PUBLIC"."TABLEPERCLASS_CUSTOMER"
(
   ID bigint PRIMARY KEY NOT NULL,
   FIRSTNAME varchar,
   LASTNAME varchar,
   CITY varchar,
   STATE varchar,
   STREET varchar,
   ZIP varchar
)
;

CREATE TABLE "PUBLIC"."TABLEPERCLASS_EMPLOYEE"
(
   ID bigint PRIMARY KEY NOT NULL,
   FIRSTNAME varchar,
   LASTNAME varchar,
   CITY varchar,
   STATE varchar,
   STREET varchar,
   ZIP varchar,
   EMPLOYEEID integer
)
;
```

One major difference between this strategy and the `SINGLE_TABLE` strategy is that no discriminator column is needed in the database schema. Also notice that each table contains every persistent property in the hierarchy. Let's now look at how we map this strategy with annotations:

```
@Entity(name = "TABLEPERCLASS_PERSON")
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public class Person
{
   @Id
   @GeneratedValue(strategy = GenerationType.TABLE)
   // Cannot accept default generation strategy for table-per-class
   private Long id;

   private String firstName;
```

```
    private String lastName;
    ...
}

@Entity(name = "TABLEPERCLASS_CUSTOMER")
public class Customer extends Person
{
    private String street;
    private String city;
    private String state;
    private String zip;
    ...
}

@Entity(name = "TABLEPERCLASS_EMPLOYEE")
public class Employee extends Customer
{
    private Integer employeeId;
    ...
}
```

Notice that the only inheritance metadata required is the `InheritanceType`, and this is needed on only the base `Person` class.

Advantages

The advantage to this approach over the `SINGLE_TABLE` strategy is that you can define constraints on subclass properties. Another plus is that it might be easier to map a preexisting legacy schema using this strategy.

Disadvantages

This strategy is not normalized, as it has redundant columns in each of its tables for each of the base class's properties. Also, to support this type of mapping, the persistence manager has to do some funky things. One way it could be implemented is for the container to use multiple queries when loading an entity or polymorphic relationship. This is a huge performance hit because the container has to do multiple round-trips to the database. Another way a container could implement this strategy is to use `SQL UNION`s. This still would not be as fast as the `SINGLE_TABLE` strategy, but it would perform much better than a multiselect implementation. The downside to an `SQL UNION` is that not all relational databases support this SQL feature. It is probably not wise to pick this strategy when developing your entity beans, unless you have good reason (e.g., an existing legacy schema).

Table per Subclass

In the table per subclass mapping, each subclass has its own table, but this table contains only the properties that are defined on that particular class. In other words, it is

similar to the TABLE_PER_CLASS strategy, except the schema is normalized. This is also called the JOINED strategy:

```
CREATE TABLE "PUBLIC"."JOINED_PERSON"
(
   ID bigint PRIMARY KEY NOT NULL,
   FIRSTNAME varchar,
   LASTNAME varchar
)
;

CREATE TABLE "PUBLIC"."JOINED_CUSTOMER"
(
   CITY varchar,
   STATE varchar,
   STREET varchar,
   ZIP varchar,
   ID bigint PRIMARY KEY NOT NULL
)
;
ALTER TABLE "PUBLIC"."JOINED_CUSTOMER"
ADD CONSTRAINT FK65AE08146E93989D
FOREIGN KEY (ID)
REFERENCES "PUBLIC"."JOINED_PERSON"(ID)
;

CREATE TABLE "PUBLIC"."JOINED_EMPLOYEE"
(
   EMPLOYEEID integer,
   EMP_PK bigint PRIMARY KEY NOT NULL
)
;
ALTER TABLE "PUBLIC"."JOINED_EMPLOYEE"
ADD CONSTRAINT FK88AF6EE4D423ED9D
FOREIGN KEY (EMP_PK)
REFERENCES "PUBLIC"."JOINED_CUSTOMER"(EMP_PK)
;
```

When the persistence manager loads an entity that is a subclass or traverses a polymorphic relationship, it does an SQL join on all the tables in the hierarchy. In this mapping, there must be a column in each table that can be used to join each table. In our example, the JOINED_EMPLOYEE, JOINED_CUSTOMER, and JOINED_PERSON tables share the same primary-key values. The annotation mapping is quite simple:

```
@Entity(name="JOINED_PERSON")
@Inheritance(strategy=InheritanceType.JOINED)
public class Person
{
   ...
}

@Entity(name="JOINED_CUSTOMER")
public class Customer extends Person
{
   ...
```

```
}

@Entity(name = "JOINED_EMPLOYEE")
@PrimaryKeyJoinColumn  (name="EMP_PK")
public class Employee extends Customer
{
   ...
}
```

The persistence manager needs to know which columns in each table will be used to perform a join when loading an entity with a `JOINED` inheritance strategy. The `@javax.persistence.PrimaryKeyJoinColumn` annotation can be used to describe this metadata:

```
package javax.persistence;

@Target({TYPE, METHOD, FIELD})
public @interface PrimaryKeyJoinColumn
{
   String name( ) default "";
   String referencedColumnName( ) default "";
   String columnDefinition( ) default "";
}
```

The `name()` attribute refers to the column contained in the current table on which you will perform a join. It defaults to the primary-key column of the superclass's table. The `referencedColumnName()` is the column that will be used to perform the join from the superclass's table. It can be any column in the superclass's table, but it defaults to its primary key. If the primary-key column names are identical between the base and subclasses, then this annotation is not needed. For instance, the `Customer` entity does not need the `@PrimaryKeyJoinColumn` annotation. The `Employee` class has a different primary-key column name than the tables of its superclasses, so the `@PrimaryKeyJoinColumn` annotation is required. If class hierarchy uses a composite key, there is a `@javax.persistence.PrimaryKeyJoinColumns` annotation that can describe multiple join columns:

```
package javax.persistence;

@Target({TYPE, METHOD, FIELD})
public @interface PrimaryKeyJoinColumns
{
   @PrimaryKeyJoinColumns[] value( );
}
```

Some persistence providers require a discriminator column for this mapping type. Most do not. Make sure to check your persistence provider implementation to see whether it is required.

Advantages

It is better to compare this mapping to other strategies to describe its advantages. Although it is not as fast as the `SINGLE_TABLE` strategy, you are able to define `NOT NULL` constraints on any column of any table, and your model is normalized.

This mapping is better than the `TABLE_PER_CLASS` strategy for two reasons. One, the relational database model is completely normalized. Two, it performs better than the `TABLE_PER_CLASS` strategy if `SQL UNION`s are not supported.

Disadvantages

It does not perform as well as the `SINGLE_TABLE` strategy.

Mixing Strategies

The persistence specification currently makes mixing inheritance strategies optional. The rules for mixing strategies in an inheritance hierarchy may be defined in future versions of the spec.

Nonentity Base Classes

The inheritance mappings we described so far in this chapter concerned a class hierarchy of entity beans. Sometimes, however, you need to inherit from a nonentity superclass. This superclass may be an existing class in your domain model that you do not want to make an entity. The `@javax.persistence.MappedSuperclass` annotation allows you to define this kind of mapping. Let's modify our example class hierarchy and change `Person` into a mapped superclass:

```
@MappedSuperclass
public class Person
{
   @Id @GeneratedValue
   public int getId( ) { return id; }
   public void setId(int id) { this.id = is; }
   ...
}

@Entity
@Table(name="CUSTOMER")
@Inheritance(strategy=InheritanceType.JOINED)
@AttributeOverride(name="lastname", column=@Column(name="SURNAME"))
public class Customer extends Person {
...
}

@Entity
@Table(name="EMPLOYEE")
```

```
@PrimaryKeyJoinColumn(name="EMP_PK")
public class Employee extends Customer {
...
}
```

Since it is not an entity, the mapped superclass does not have an associated table. Any subclass inherits the persistence properties of the base class. You can override any mapped property of the mapped class by using the `@javax.persistence.AttributeOverride` annotation.

You can have `@MappedSuperclass` annotated classes in between two `@Entity` annotated classes in a given hierarchy. Also, nonannotated classes (i.e., not annotated with `@Entity` or `@MappedSuperclass`) are completely ignored by the persistence provider.

CHAPTER 13

Queries, the Criteria API, and JPA QL

Querying is a fundamental feature of all relational databases. It allows you to pull complex reports, calculations, and information about intricately related objects from persistence storage. Queries in Java Persistence are done using the JPA QL query language, native Structured Query Language (SQL), and the new Criteria API.

JPA QL is a declarative query language similar to the SQL used in relational databases, but it is tailored to work with Java objects rather than a relational schema. To execute queries, you reference the properties and relationships of your entity beans rather than the underlying tables and columns these objects are mapped to. When a JPA QL query is executed, the entity manager uses the information you provided through the mapping metadata, discussed in the previous two chapters, and automatically translates it to an appropriate native SQL query. This generated native SQL is then executed through a JDBC driver directly on your database. Since JPA QL is a query language that represents Java objects, it is portable across vendor database implementations because the entity manager handles the conversion to raw SQL for you.

The JPA QL language is easy for developers to learn, yet precise enough to be interpreted into native database code. This rich and flexible query language empowers developers while executing in fast native code at runtime. Plus, because JPA QL is object-oriented, queries are usually much more compact and readable than their SQL equivalent. EJB QL existed in the EJB 2.1 specification, and it is really the only feature that survived in the new release. Since Persistence is completely spun off in its own specification, it's now known as JPA QL. Although it was well-formed, EJB QL in EJB 2.1 was incomplete, forcing developers to escape to JDBC or write really inefficient code. JPA QL has been greatly improved and expanded to be more parallel to SQL and should now meet most of your needs. Things such as projection, `GROUP BY`, and `HAVING` have been added, as well as bulk updates and deletes.

Like all query languages, JPA QL is modeled as a string, and therefore cannot be checked for structural correctness by the Java compiler. New to Java Persistence 2.0 is the Criteria API, a fluid interface for building queries using an object model.

Sometimes, though, JPA QL and the Criteria API are not enough. Because these are portable query language extensions, they cannot always take advantage of specific proprietary features of your database vendor. JPA QL does not allow you to execute stored procedures, for instance. The EJB 3.x Expert Group foresaw the need for this and has provided an API to map native SQL calls to your entity beans.

JPA QL and native SQL queries are executed through the `javax.persistence.Query` interface. The `Query` interface is analogous to the `java.sql.PreparedStatement` interface. This `Query` API gives you methods for paging your result set, as well as passing Java parameters to your query. Queries can be predeclared through annotations or XML or created dynamically at runtime through `EntityManager` APIs.

Similarly, the Criteria API has the `javax.persistence.criteria.CriteriaQuery` interface, through which we can construct queries via the object model and submit these via the `EntityManager`.

Query API

A query in Java Persistence is a full-blown Java interface that you obtain at runtime from the entity manager:

```
package javax.persistence;

public interface Query
{
    List getResultList();
    Object getSingleResult();
    int executeUpdate();
    Query setMaxResults(int maxResult);
    int getMaxResults();
    Query setFirstResult(int startPosition);
    int getFirstResult();
    Query setHint(String hintName, Object value);
    Map<String, Object> getHints();
    <T> Query setParameter(Parameter<T> param, T value);
    Query setParameter(Parameter<Calendar> param, Calendar value,
                       TemporalType temporalType);
    Query setParameter(Parameter<Date> param, Date value,
                       TemporalType temporalType);
    Query setParameter(String name, Object value);
    Query setParameter(String name, Calendar value,
                       TemporalType temporalType);
    Query setParameter(String name, Date value,
                       TemporalType temporalType);
    Query setParameter(int position, Object value);
    Query setParameter(int position, Calendar value,
                       TemporalType temporalType);
    Query setParameter(int position, Date value,
                       TemporalType temporalType);
    Set<Parameter<?>> getParameters();
    Parameter<?> getParameter(String name);
```

```
    <T> Parameter<T> getParameter(String name, Class<T> type);
    Parameter<?> getParameter(int position);
    <T> Parameter<T> getParameter(int position, Class<T> type);
    boolean isBound(Parameter<?> param);
    <T> T getParameterValue(Parameter<T> param);
    Object getParameterValue(String name);
    Object getParameterValue(int position);
    Query setFlushMode(FlushModeType flushMode);
    FlushModeType getFlushMode();
    Query setLockMode(LockModeType lockMode);
    LockModeType getLockMode();
    <T> T unwrap(Class<T> cls);
}
```

Queries are created using these `EntityManager` methods:

```
package javax.persistence;

public interface EntityManager
{
   public Query createQuery(String qlString);
   public <T> TypedQuery<T> createQuery(CriteriaQuery<T> criteriaQuery);
   public <T> TypedQuery<T> createQuery(String qlString, Class<T> resultClass);
   public Query createNamedQuery(String name);
   public <T> TypedQuery<T> createNamedQuery(String name, Class<T> resultClass);
   public Query createNativeQuery(String sqlString);
   public Query createNativeQuery(String sqlString, Class resultClass);
   public Query createNativeQuery(String sqlString, String resultSetMapping);
}
```

Let's first look at using `EntityManager.createQuery()` to create a query dynamically at runtime:

```
try
{
   // Define query String
   final String jpaQlQuery = "FROM " + Employee.class.getSimpleName() +
      " e WHERE e.name='Dave'";
   // Query and get result
   final Employee roundtrip = (Employee)em.createQuery(jpaQlQuery)
      .getSingleResult();
}
catch (EntityNotFoundException notFound) {}
catch (NonUniqueResultException nonUnique) {}
```

The previous query looks for a single, unique `Employee` entity named Dave. The query is executed when the `getSingleResult()` method is called. This method expects that the call will return only one result. If no result is returned, the method throws a `javax.persistence.EntityNotFoundException` runtime exception. If more than one result is found, a `javax.persistence.NonUniqueResultException` runtime exception is thrown. Since both of these exceptions are `RuntimeException`s, the example code is not required to have a full *try/catch* block.

There is a good chance that the NonUniqueResultException would be thrown by this example. Believe it or not, there are a lot of Daves in the world. You can change the query to use the getResultList() method to obtain a collection of results:

```
Query query = entityManager.creatQuery(
              "from Employee e where e.name='Dave'");
java.util.List<?> bills = query.getResultList( );
```

The getResultList() method does not throw an exception if there are no Daves; the returned list would just be empty.

Parameters

Much like a java.sql.PreparedStatement in JDBC, JPA QL allows you to specify parameters in query declarations so that you can reuse and execute the query multiple times on different sets of parameters. Two syntaxes are provided: named parameters and positional parameters. Let's modify our earlier Employee query to take both last name and first name as named parameters:

```
// Define query String
final String jpaQlQuery = "FROM " + Employee.class.getSimpleName() +
   " e WHERE e.name=:name";
// Set parameter
jpaQlQuery.setParameter("name", "Dave");
// Query and get result
final Employee roundtrip = (Employee)em.createQuery(jpaQlQuery).getSingleResult();
```

The : character followed by the parameter name is used in JPA QL statements to identify a named parameter. The setParameter() method in this example takes the name of the parameter first, and then the actual value. EJB QL also supports positional parameters. Let's modify the previous example to see this mode in action:

```
// Define query String
final String jpaQlQuery = "FROM " + Employee.class.getSimpleName() +
   " e WHERE e.name=?1";
// Set parameter
jpaQlQuery.setParameter(1, "Dave");
// Query and get result
final Employee roundtrip = (Employee)em.createQuery(jpaQlQuery).getSingleResult();
```

Instead of a string named parameter, setParameter() also takes a numeric parameter position. The ? character is used instead of the : character used with named parameters. Numeric parameters are indexed (start at) 1, not 0.

Using named parameters over positional parameters is recommended, as the JPA QL code becomes self-documenting. This is especially useful when working with predeclared queries.

Date Parameters

If you need to pass `java.util.Date` or `java.util.Calendar` parameters into a query, you need to use special `setParameter` methods:

```
package javax.persistence;

public enum TemporalType
{
     DATE, //java.sql.Date
     TIME, //java.sql.Time
     TIMESTAMP //java.sql.Timestamp
}

public interface Query
{
   Query setParameter(String name, java.util.Date value, TemporalType temporalTyp
e);
   Query setParameter(String name, Calendar value, TemporalType temporalType);
   Query setParameter(int position, Date value, TemporalType temporalType);
   Query setParameter(int position, Calendar value, TemporalType temporalType);
}
```

A `Date` or `Calendar` object can represent a real date, a time of day, or a numeric timestamp. Because these object types can represent different things at the same time, you need to tell your `Query` object how it should use these parameters. The `javax.persistence.TemporalType` passed in as a parameter to the `setParameter()` method tells the `Query` interface what database type to use when converting the `java.util.Date` or `java.util.Calendar` parameter to a native SQL type.

Paging Results

Sometimes an executed query returns too many results. For instance, maybe we're displaying a list of customers on a web page. The web page can display only so many customers, and maybe there are thousands or even millions of customers in the database. The `Query` API has two built-in functions to solve this type of scenario—`setMaxResults()` and `setFirstResult()`:

```
public List getEmployees(int max, int index) {
   Query query = entityManager.createQuery("from Employee e");
   return query.setMaxResults(max).
                setFirstResult(index).
                getResultList();
}
```

The `getEmployees()` method executes a query that obtains all employees from the database. We limit the number of employees it returns by using the `setMaxResults()` method, passing in the `max` method parameter. The method is also designed so that you can define an arbitrary set of results that you want returned by the execution of the query. The `setFirstResult()` method tells the query what position in the executed query's result set you want returned. So, if you had a max result of 3 and a first result

of 5, employees 5, 6, and 7 would be returned. We set this value in the `getEmployees()` method with the `index` parameter. Let's take this method and write a code fragment that lists all customers in the database:

```
List results;
int first = 0;
int max = 10;

do
{
   results = getEmployees(max, first);
   Iterator it = results.iterator();
   while (it.hasNext())
   {
      Employee employee = (Employee)it.next();
      System.out.println("Employee name: " + employee.getName());
   }
   entityManager.clear();
   first = first + results.getSize();
}
while(results.size() > 0);
```

In this example, we loop through all employees in the database and output their names to the system output stream. If we had thousands or even millions of employees in the database, we could quickly run out of memory, as each execution of the `getEmployees()` method would return customers that were still managed by the entity manager. So, after we are finished outputting a block of customers, we call `EntityManager.clear()` to detach these customers and let them be garbage-collected by the Java VM. Use this pattern when you need to deal with a lot of entity objects within the same transaction.

Hints

Some Java Persistence vendors will provide additional add-on features that you can take advantage of when executing a query. For instance, the JBoss EJB 3.x implementation allows you to define a timeout for the query. These types of add-on features can be specified as hints using the `setHint()` method on the query. Here's an example of defining a JBoss query timeout using hints:

```
Query query = entityManager.createQuery("from Employee e");
query.setHint("org.hibernate.timeout", 1000);
```

The `setHint()` method takes a string name and an arbitrary object parameter.

FlushMode

In Chapter 9, we talked about flushing and flush modes. Sometimes you would like a different flush mode to be enforced for the duration of a query. For instance, maybe a query wants to make sure that the entity manager does not flush before the query is

executed (since the default value implies that the entity manager can). The `Query` interface provides a `setFlushMode()` method for this particular purpose:

```
Query query = manager.createQuery("from Employee e");
query.setFlushMode(FlushModeType.COMMIT);
```

In this example, we're telling the persistence provider that we do not want the query to do any automatic flushing before this particular query is executed. Using this commit mode can be dangerous if some correlated dirty entities are in the persistence context. You might return wrong entities from your query. Therefore, it is recommended that you use the `FlushModeType.AUTO`.

JPA QL

Now that you have a basic understanding of how to work with `Query` objects, you can learn what features are available to you for creating your own JPA QL queries. JPA QL is expressed in terms of the abstract persistence schema of an entity: its abstract schema name, basic properties, and relationship properties. JPA QL uses the abstract schema names to identify beans, the basic properties to specify values, and the relationship properties to navigate across relationships.

To discuss JPA QL, we will use the relationships defined in Chapter 11.

Abstract Schema Names

The abstract schema name can be defined by metadata, or it can default to a specific value. It defaults to the unqualified name of the entity bean class if the `name()` attribute is not specified when declaring the `@Entity` annotation.

In the following example, the `@Entity.name()` attribute is not specified on the `Employee` bean class, so `Employee` is used to reference the entity within JPA QL calls:

```
@Entity
public class Employee {...}

entityManager.createQuery("SELECT e FROM Employee AS e");
```

In the following example, since the `@Entity.name()` attribute is defined, you would reference `Employee` entities in JPA QL as `Emp`:

```
@Entity(name="Emp")
public class Employee {...}

entityManager.createQuery("SELECT e FROM Emp AS e");
```

Simple Queries

The simplest JPA QL statement has no `WHERE` clause and only one abstract schema type. For example, you could define a query method to select all `Employee` beans:

```
SELECT OBJECT( e ) FROM Employee AS e
```

The `FROM` clause determines which entity bean types will be included in the `SELECT` statement (i.e., it provides the *scope* of the select). In this case, the `FROM` clause declares the type to be `Employee`, which is the abstract schema name of the `Employee` entity. The `AS e` part of the clause assigns `e` as the identifier of the `Employee` entity. This is similar to SQL, which allows an identifier to be associated with a table. Identifiers can be any length and follow the same rules that are applied to field names in the Java programming language. However, identifiers cannot be the same as existing abstract schema name values. In addition, identification variable names are *not* case-sensitive, so an identifier of `employee` would be in conflict with an abstract schema name of `Employee`. For example, the following statement is illegal because `Employee` is the abstract schema name of the `Employee` EJB:

```
SELECT OBJECT ( employee ) FROM Employee AS employee
```

The `AS` operator is optional, but it is used in this book to help make the JPA QL statements clearer. The following two statements are equivalent:

```
SELECT OBJECT(e) FROM Employee AS e

SELECT e FROM Employee e
```

The `SELECT` clause determines the type of any values that are returned. In this case, the statement returns the `Employee` entity bean, as indicated by the `e` identifier.

The `OBJECT()` operator is optional and is a relic requirement of the EJB 2.1 specification. It is there for backward compatibility.

Identifiers cannot be EJB QL reserved words. In Java Persistence, the following words are reserved:

SELECT	FROM	WHERE	UPDATE	DELETE
JOIN	OUTER	INNER	GROUP	BY
HAVING	FETCH	DISTINCT	OBJECT	NULL
TRUE	FALSE	NOT	AND	OR
BETWEEN	LIKE	IN	AS	UNKNOWN
EMPTY	MEMBER	OF	IS	AVG
MAX	MIN	SUM	COUNT	ORDER
ASC	DESC	MOD	UPPER	LOWER
TRIM	POSITION	CHARACTER_LENGTH	CHAR_LENGTH	BIT_LENGTH
CURRENT_TIME	CURRENT_DATE	CURRENT_TIMESTAMP	NEW	

It's a good practice to avoid all SQL reserved words, because you never know which ones will be used by future versions of EJB QL. You can find more information in the appendix of *SQL in a Nutshell* (O'Reilly).

Selecting Entity and Relationship Properties

JPA QL allows `SELECT` clauses to return any number of basic or relationship properties. For example, we can define a simple `SELECT` statement to return the name of all of employees in the registry:

```
final String jpaQlQuery = "SELECT e.name FROM Employee AS e"
@SuppressWarnings("unchecked")
final List<String> names = em.createQuery(jpaQlQuery).getResultList();
```

The `SELECT` clause uses a simple path to select the `Employee` entity's `name` property as the return type. The persistence property names are identified by the access type of your entity bean class, regardless of whether you've applied your mapping annotations on a `get` or `set` method or on the member fields of the class.

If you use `get` or `set` methods to specify your persistent properties, then the property name is extracted from the method name. The `get` part of the method name is removed, and the first character of the remaining string is lowercase.

When a query returns more than one item, you must use the `Query.getResultList()` method. The return value of the earlier query is dependent upon the type of the field being accessed. In our case, we obtain a `List` of `String`s because the `name` property is a `String`. If the `SELECT` clause queries more than one column or entity, the results are aggregated in an object array (`Object[]`) in the `java.util.List` returned by `getResultList()`. The following code shows how to access the returned results from a multi-property select:

```
// Create and persist
final Employee josh = new Employee(ID_JOSH, NAME_JOSH);
em.persist(josh);

// Lookup
final String jpaQlQuery = "SELECT e.name, e.id FROM " +
    Employee.class.getSimpleName() + " AS e";
@SuppressWarnings("unchecked")
final List<Object[]> properties = em.createQuery(jpaQlQuery).getResultList();

// Test
Assert.assertEquals(1, properties.size());
Assert.assertEquals(NAME_JOSH, properties.get(0)[0]);
Assert.assertEquals(ID_JOSH, properties.get(0)[1]);
```

Paths can be as long as required. It's common to use paths that navigate over one or more relationship fields to end at either a basic or a relationship property. For example, the following JPA QL statement selects the name of the manager of the employees:

```
@Entity
public class Employee
{
   /**
    * Manager of the {@link Employee}
    */
   @ManyToOne
   private Employee manager;
   ...
}
```

```
SELECT e.manager.name FROM Employee AS e
```

Using these relationships, we can specify more complex paths. Paths cannot navigate beyond persistent properties. For example, imagine that `Address` uses a `ZipCode` class as its `zip` property and this property is stored as a byte stream in the database:

```
public class ZipCode implements java.io.Serializable
{
    public int mainCode;
    public int codeSuffix;
    ...
}

@Entity
public class Address
{
   private ZipCode zip;
}
```

You can't navigate to one of the `ZipCode` class's instance fields:

```
// This is illegal
SELECT a.zip.mainCode FROM Address AS a
```

Of course, you could make the `ZipCode` class `@Embeddable`. If you did this, then you could obtain properties of the `ZipCode` class:

```
@Entity
public class Address {
   @Embedded private ZipCode zip;
}
```

This JPA QL would now be legal:

```
// @Embedded makes this legal now
SELECT a.zip.mainCode FROM Address AS a
```

It's illegal to navigate across a collection-based relationship field. The following JPA QL statement is illegal, even though the path ends in a single-type relationship field:

```
// This is illegal
SELECT e.phones.number FROM Employee AS e
```

If you think about it, this limitation makes sense. You can't use a navigation operator (.) in Java to access elements of a `java.util.Collection` object. For example, if `getPhones()` returns a `java.util.Collection` type, this statement is illegal:

```
// this is illegal in the Java programming language
employee.getPhones().getNumber();
```

Referencing the elements of a collection-based relationship field is possible, but it requires the use of an `IN` or `JOIN` operator and an identification assignment in the `FROM` clause.

Constructor Expressions

One of the most powerful features of JPA QL is the ability to specify a constructor within the `SELECT` clause that can allocate plain Java objects (nonentities) and pass columns you select into that constructor. For example, let's say we want to aggregate IDs and names from our `Employee` entity into a plain Java object called `Name`:

```
package org.mypackage;
public class Name
{
   private Long id;
   private String name;

   public Name(final Long id, final String name)
   {
      this.setId(id);
      this.setName(name);
   }

   public Long getId() { return id; }
   public void setId(Long id) { this.id = id; }
   public String getName() { return name; }
   public void setName(String name) { this.name = name; }
}
```

We can actually have our query return a list of `Name` classes instead of a plain list of strings. We do this by calling the constructor of `Name` directly within our query:

```
SELECT new org.mypackage.Name(e.id,e.name) FROM Employee e
```

The `Query` object will automatically allocate an instance of `Name` for each row returned, passing in the `id` and `name` columns as arguments to the `Name`'s constructor. This feature is incredibly useful for generating typed reports and can save you a lot of typing.

The IN Operator and INNER JOIN

Many relationships between entity beans are collection-based, and being able to access and select beans from these relationships is important. We've seen that it is illegal to select elements directly from a collection-based relationship. To overcome this

limitation, JPA QL introduces the `IN` operator, which allows an identifier to represent individual elements in a collection-based relationship field.

The following query uses the `IN` operator to select the elements from a collection-based relationship. It returns all the `Phone`s for an `Employee`:

```
SELECT p FROM Employee AS e, IN( e.phones ) p
```

The `IN` operator assigns the individual elements in the `phones` property to the `p` identifier. Once we have an identifier to represent the individual elements of the collection, we can reference them directly and even select them in the JPA QL statement. We can also use the element identifier in path expressions. For example, the following statement selects all phone numbers (a property of `Phone`) for an `Employee`:

```
SELECT p.number FROM Employee AS e, IN( e.phones ) p
```

The identifiers assigned in the `FROM` clause are evaluated from left to right. Once you declare an identifier, you can use it in subsequent declarations in the `FROM` clause. The `e` identifier, which was declared first, was subsequently used in the `IN` operator to define the `p` identifier.

This query can also be expressed as an `INNER JOIN`:

```
SELECT p.number FROM Employee e INNER JOIN e.phones p
```

The `INNER JOIN` syntax parallels the SQL language much better and is more intuitive for developers coming from the relational world.

LEFT JOIN

The `LEFT JOIN` syntax enables retrieval of a set of entities where matching values in the join statement may not exist. For values that do not exist, a null value is placed in the result set.

For example, let's say we want to generate a report with an employee's name and all the employee's phone numbers. Some employees may not have specified a phone number, but we still want to list their names. We would use a `LEFT JOIN` to acquire all of this information, including employees with no phone numbers:

```
SELECT e.name, p.number From Employee e LEFT JOIN e.phones p
```

If there were three employees in our system, and Bill Burke did not provide any phone numbers, the return values might look like this:

```
David Ortiz 617-555-0900
David Ortiz 617-555-9999
Trot Nixon 781-555-2323
Bill Burke null
```

The previous query can also be expressed as a `LEFT OUTER JOIN`. This is just syntax sugar to parallel SQL-92:

```
SELECT e.name, p.number From Employee e LEFT OUTER JOIN e.phones p
```

Fetch Joins

The `JOIN FETCH` syntax allows you to preload a returned entity's relationships, even if the relationship property has a `FetchType` of `LAZY`. For example, let's say we have defined our employee's one-to-many relationship to `Phone` as follows:

```
/**
 * All {@link Phone}s for this {@link Employee}
 */
@OneToMany (fetch=FetchType.LAZY)
private Collection<Phone> phones;
```

If we want to print out all employee information, including their phone numbers, usually we would just query for all employees and then traverse the `getPhones()` method inside a `for` loop:

```
Query query = manager.createQuery("SELECT e FROM Employee e"); ❶
List results = query.getResultList( );
Iterator it = results.iterator( );
while (it.hasNext( )) {
  Employee e = (Employee)it.next( );
  System.out.print(e.getName( ));
  for (Phone p : e.getPhones( )) { ❷
  System.out.print(p.getNumber( ));
  }
   System.out.println("");
 }
```

❶ There are performance problems with the preceding code. Because the `Phone` relationship is annotated as being lazily loaded in the `Employee` bean class, the `Phone` collection will not be initialized when we do the initial query.

❷ When `getPhones()` is executed, the persistence engine has to do an additional query to get the `Phone` entities associated with the `Employee`. This is called the *N + 1 problem*, as we have to do *N* extra queries beyond our initial query. When tuning database applications, it is always important to reduce the number of round-trips made to the database as much as possible. This is where the `JOIN FETCH` syntax comes into play. Let's modify our query to preload the `Phone` association:

```
SELECT e FROM Employee e LEFT JOIN FETCH e.phones
```

Using `LEFT JOIN FETCH` will additionally preload the `Phone` association. This can have a dramatic effect on performance because instead of *N* + 1 queries, only one query is made to the database.

Using DISTINCT

The `DISTINCT` keyword ensures that the query does not return duplicates. For example, the following query finds all employees on a team. This query will return duplicates:

```
SELECT emp FROM Team t INNER JOIN t.members emp
```

If an `Employee` belongs to more than one `Team`, there will be duplicate references to that employee in the result. Using the `DISTINCT` keyword ensures that each `Employee` is represented only once in the result:

```
SELECT DISTINCT emp FROM Team t INNER JOIN t.members emp
```

The WHERE Clause and Literals

You can use literal values to narrow the scope of the elements selected. This is accomplished through the `WHERE` clause, which behaves in much the same way as the `WHERE` clause in SQL.

For example, you can define a JPA QL statement that selects all the `Employee` entities that have a specific name. The literal in this case is a `String` literal. Literal strings are enclosed by single quotes. Literal values that include a single quote, such as the restaurant name *Wendy's*, use two single quotes to escape the quote: `Wendy''s`. The following statement returns employees who have the name "Natalie Glass":

```
SELECT e FROM Employee AS e WHERE e.name = 'Natalie Glass'
```

Path expressions in the `WHERE` clause may be used in the same way as in the `SELECT` clause. When making comparisons with a literal, the path expression must evaluate to a basic property; you can't compare a relationship field with a literal.

In addition to literal strings, literals can be exact numeric values (`long` types) and approximate numeric values (`double` types). Exact numeric literal values are expressed using the Java integer literal syntax (`321`, `-8932`, `+22`). Approximate numeric literal values are expressed using Java floating-point literal syntax in scientific (`5E3`, `-8.932E5`) or decimal (`5.234`, `38282.2`) notation. Boolean literal values use `TRUE` and `FALSE`.

The WHERE Clause and Operator Precedence

The `WHERE` clause is composed of conditional expressions that reduce the scope of the query and limit the number of items selected. Several conditional and logical operators can be used in expressions; they are listed here in order of precedence:

- Navigation operator (`.`)
- Arithmetic operators: `+`, `-` (unary); `*`, `/` (multiplication and division); `+`, `-` (addition and subtraction)
- Comparison operators: `=`, `>`, `>=`, `<`, `<=`, `<>` (not equal), `LIKE`, `BETWEEN`, `IN`, `IS NULL`, `IS EMPTY`, `MEMBER OF`
- Logical operators: `NOT`, `AND`, `OR`

The WHERE Clause and Arithmetic Operators

The arithmetic operators allow a query to perform arithmetic in the process of doing a comparison. Arithmetic operators can be used only in the `WHERE` clause, not in the `SELECT` clause.

The following JPA QL statement returns references to all the `Employee`s who have an ID less than 100:

```
SELECT e FROM Employee AS e WHERE e.id < (1 * 10)
```

The rules applied to arithmetic operations are the same as those used in the Java programming language, where numbers are *widened*, or *promoted*, in the process of performing a calculation. For example, multiplying a `double` and an `int` value requires that the `int` first be promoted to a `double` value. (The result will always be that of the widest type used in the calculation, so multiplying an `int` and a `double` results in a `double` value.)

`String`, `boolean`, and entity object types cannot be used in arithmetic operations. For example, using the addition operator with two `String` values is considered an illegal operation. There is a special function for concatenating `String` values, covered in "Functional expressions in the WHERE clause" on page 225.

The WHERE Clause and Logical Operators

Logical operators such as `AND`, `OR`, and `NOT` operate the same way in JPA QL as their corresponding logical operators in SQL.

Logical operators evaluate only Boolean expressions, so each operand (i.e., each side of the expression) must evaluate to `true`, `false`, or `NULL`. Logical operators have the lowest precedence so that all the expressions can be evaluated before they are applied.

The `AND` and `OR` operators don't behave like their Java language counterparts, `&&` and `||`. JPA QL does not specify whether the righthand operands are evaluated conditionally. For example, the `&&` operator in Java evaluates its righthand operand *only* if the lefthand operand is `true`. Similarly, the `||` logical operator evaluates the righthand operand *only* if the lefthand operand is `false`. We can't make the same assumption for the `AND` and `OR` operators in JPA QL. Whether these operators evaluate righthand operands depends on the native query language into which the statements are translated. It's best to assume that both operands are evaluated on all logical operators.

`NOT` simply reverses the Boolean result of its operand; expressions that evaluate to the Boolean value of `true` become `false`, and vice versa

The WHERE Clause and Comparison Symbols

Comparison operators, which use the symbols `=`, `>`, `>=`, `<`, `<=`, and `<>` should be familiar to you. Only the `=` and `<>` (equals and not equals) operators may be used on `boolean` and entity object identifiers. The greater-than and less-than symbols (`>`, `>=`, `<`, `<=`) can

be used on numeric values as well as strings. However, the semantics of these operations are not defined by the Java Persistence specification. Is character case (upper or lower) important? Does leading and trailing whitespace matter? Issues like these affect the ordering of string values. In order for JPA QL to maintain its status as an abstraction of native query languages, it cannot dictate `String` ordering, because native query languages may have very different ordering rules. In fact, even different relational database vendors vary on the question of `String` ordering, which makes it all but impossible to standardize ordering, even for SQL "compliant" databases.

Of course, this is all academic if you plan on using the same database well into the future. In such a case, the best thing to do is to examine the documentation for the database you are using to find out how it orders strings in comparisons. This tells you exactly how your JPA QL comparisons will work.

The WHERE Clause and Equality Semantics

Although it is legal to compare an exact numeric value (`short`, `int`, `long`) to an approximate numeric value (`double`, `float`), all other equality comparisons must compare the same types. You cannot, for example, compare a `String` value of `123` to the `Integer` literal `123`. However, you can compare two `String` types for equality.

You can compare numeric values for which the rules of numeric promotion apply. For example, a `short` may be compared to an `int`, an `int` to a `long`, etc. Java Persistence also states that primitives may be compared to primitive wrapper types—the rules of numeric promotion apply.

In older versions of the spec, `String` type comparisons had to match exactly, character for character. EJB 2.1 dropped this requirement, making the evaluation of equality between `String` types more ambiguous. This continued in Java Persistence. Again, this ambiguity arises from the differences between kinds of databases (relational versus object-oriented versus file), as well as differences between vendors of relational databases. Consult your vendor's documentation to determine exactly how `String` equality comparisons are evaluated.

You can also compare entity objects for equality, but these too must be of the same type. To be more specific, they must both be entity object references to beans from the same deployment. Once it's determined that the bean is the correct type, the actual comparison is performed on the beans' primary keys. If they have the same primary key, they are considered equal.

You may use `java.util.Date` objects in equality comparisons. See "Date Parameters" on page 209.

The WHERE Clause and BETWEEN

The BETWEEN clause is an inclusive operator specifying a range of values. In this example, we use it to select all Employees with an ID between 100 and 200:

```
SELECT e FROM Employee AS e WHERE e.id BETWEEN 100 AND 200
```

The BETWEEN clause may be used only on numeric primitives (byte, short, int, long, double, float) and their corresponding java.lang.Number types (Byte, Short, Integer, etc.). It cannot be used on String, boolean, or entity object references.

Using the NOT logical operator in conjunction with BETWEEN excludes the range specified. For example, the following JPA QL statement selects all the Employees that have an ID less than 100 or greater than 200:

```
SELECT e FROM Employee AS e WHERE e.id NOT BETWEEN 100 AND 200
```

The net effect of this query is the same as if it had been executed with comparison symbols:

```
SELECT e FROM Employee AS e
WHERE e.id < 100 OR e.id > 200
```

The WHERE Clause and IN

The IN conditional operator used in the WHERE clause is not the same as the IN operator used in the FROM clause (that's why the JOIN keyword in the FROM clause should be preferred over the IN keyword for collection navigation). In the WHERE clause, IN tests for membership in a list of literal values. For example, the following JPA QL statement uses the IN operator to select all the customers who reside in a specific set of states:

```
SELECT e FROM Employee AS e WHERE e.address.state IN ('FL', 'TX', 'MI', 'WI', 'MN')
```

Applying the NOT operator to this expression reverses the selection, excluding all customers who reside in the list of states:

```
SELECT e FROM Employee AS e WHERE e.address.state NOT IN ('FL', 'TX', 'MI', 'WI',
'MN')
```

If the field tested is null, the value of the expression is "unknown," which means it cannot be predicted.

The IN operator can be used with operands that evaluate to either string or numeric values. For example, the following JPA QL fragment uses the IN operator to select values 1, 3, 5, and 7:

```
WHERE e.property IN (1,3,5,7)
```

The IN operator can also be used with input parameters. For example, the following query selects all the parameters:

```
WHERE e.property IN (?1,?2,?3)
```

In this case, the input parameters (?1, ?2, and ?3) are combined with parameters sent to the query during runtime execution.

The WHERE Clause and IS NULL

The IS NULL comparison operator allows you to test whether a path expression is null. For example, the following EJB QL statement selects all the Employees who do not have a manager (those who are self-managing):

```
SELECT e FROM Employee AS e WHERE e.manager IS NULL
```

Using the NOT logical operator, we can reverse the results of this query, selecting all Employees who report to a manager:

```
SELECT e FROM Employee AS e WHERE e.manager IS NOT NULL
```

Path expressions are composed using "inner join" semantics. If an entity has a null relationship field, any query that uses that field as part of a path expression eliminates that entity from consideration. For example, if the Employee entity representing "John Smith" has a null value for its address relationship field, then the "John Smith" Employee entity won't be included in the result set for the following query:

```
SELECT e FROM Employee AS e
WHERE e.address.state = 'TX'
AND c.name = 'John Smith'
```

This seems obvious at first, but stating it explicitly helps eliminate much of the ambiguity associated with null relationship fields.

The NULL comparison operator can also be used to test input parameters. In this case, NULL is usually combined with the NOT operator to ensure that an input parameter is not a null value. For example, this query can be used to test for null input parameters. The JPA QL statement first checks that the city and state input parameters are not null and then uses them in comparison operations:

```
SELECT e FROM Employee AS e
WHERE :city IS NOT NULL AND :state IS NOT NULL
AND e.address.state = :state
AND e.address.city = :city
```

In this case, if either of the input parameters is a null value, the query returns an empty List, avoiding the possibility of UNKNOWN results from null input parameters. Your Java code should do these null checks (input assertions) up front to avoid an unnecessary database round-trip.

If the results of a query include a null relationship or a basic field, the results must include null values. For example, the following query selects the addresses of customers whose name is "John Smith":

```
SELECT e.address FROM Employee AS e WHERE e.name = 'John Smith'
```

If the Employee entity representing "John Smith" has a null value for its address relationship field, the previous query returns a List that includes a null value—the null represents the address relationship field of "John Smith"—in addition to a bunch of Address entity references. You can eliminate null values by including the NOT NULL operator in the query, as shown here:

```
SELECT e.address.city FROM Employee AS e
WHERE e.address.city NOT NULL AND e.address.state = 'FL'
```

The WHERE Clause and IS EMPTY

The IS EMPTY operator allows the query to test whether a collection-based relationship is empty. Remember from Chapter 11 that a collection-based relationship will never be null. If a collection-based relationship field has no elements, it returns an empty Collection or Set.

Testing whether a collection-based relationship is empty has the same purpose as testing whether a single relationship field or basic field is null: it can be used to limit the scope of the query and items selected. For example, the following query selects all the Employees who have no Phones:

```
SELECT e FROM Employee AS e
WHERE e.phones IS EMPTY
```

The NOT operator reverses the result of IS EMPTY. The following query selects all the Employees who have at least one Phone:

```
SELECT e FROM Employee AS e
WHERE e.phones IS NOT EMPTY
```

The WHERE Clause and MEMBER OF

The MEMBER OF operator is a powerful tool for determining whether an entity is a member of a specific collection-based relationship. The following query determines whether a particular Employee entity (specified by the input parameter) is a member of any of the Team/Employee relationships:

```
SELECT t
FROM Team AS t, IN (t.members) AS m, Employee AS e
WHERE
e = :myCustomer
  AND
e MEMBER OF m.members
```

Applying the NOT operator to MEMBER OF has the reverse effect, selecting all the customers on which the specified customer does not have a team:

```
SELECT t
FROM Team AS t, IN (t.members) AS m, Employee AS e
WHERE
e = :myCustomer
```

```
    AND
e NOT MEMBER OF m.members
```

Checking whether an entity is a member of an empty collection always returns `false`.

The WHERE Clause and LIKE

The `LIKE` comparison operator allows the query to select `String` type fields that match a specified pattern. For example, the following JPA QL statement selects all the customers with hyphenated names, like "Monson-Haefel" and "Berners-Lee":

```
SELECT OBJECT( e ) FROM Employee AS e WHERE e.name LIKE '%-%'
```

You can use two special characters when establishing a comparison pattern: `%` (percent) stands for any sequence of characters, and `_` (underscore) stands for any single character. You can use these characters at any location within a string pattern. If a `%` or `_` actually occurs in the string, you can escape it with the `\` (backslash) character. The `NOT` logical operator reverses the evaluation so that matching patterns are excluded. The following examples show how the `LIKE` clause evaluates `String` type fields:

`phone.number LIKE '617%'`
: True for "617-322-4151"

: False for "415-222-3523"

`cabin.name LIKE 'Suite _100'`
: True for "Suite A100"

: False for "Suite A233"

`phone.number NOT LIKE '608%'`
: True for "415-222-3523"

: False for "608-233-8484"

`someField.underscored LIKE '\_%'`
: True for "_xyz"

: False for "abc"

`someField.percentage LIKE '\%%'`
: True for "% XYZ"

: False for "ABC"

The `LIKE` operator can also be used with input parameters:

```
SELECT e FROM Employee AS e WHERE e.name LIKE :param
```

Functional Expressions

JPA QL has numerous functions that you can use to process strings and numeric values.

Functional expressions in the WHERE clause

JPA QL has seven functional expressions that allow for simple `String` manipulation and three functional expressions for basic numeric operations. The `String` functions are:

`LOWER(String)`
: Converts a string to lowercase.

`UPPER(String)`
: Converts a string to uppercase.

`TRIM([[LEADING | TRAILING | BOTH] [trim_char] FROM] String)`
: Allows you to trim a specified character from the beginning (`LEADING`), end (`TRAILING`), or both (`BOTH`). If you do not specify a trim character, the space character will be assumed.

`CONCAT(String1, String2)`
: Returns the `String` that results from concatenating `String1` and `String2`.

`LENGTH(String)`
: Returns an `int` indicating the length of the string.

`LOCATE(String1, String2 [, start])`
: Returns an `int` indicating the position at which `String1` is found within `String2`. If it's present, `start` indicates the character position in `String2` at which the search should start. Support for the `start` parameter is optional; some containers will support it and others will not. Don't use it if you want to ensure that the query is portable.

`SUBSTRING(String1, start, length)`
: Returns the `String` consisting of `length` characters taken from `String1`, starting at the position given by `start`.

The `start` and `length` parameters indicate positions in a `String` as integer values. You can use these expressions in the `WHERE` clause to refine the scope of the items selected. Here's how the `LOCATE` and `LENGTH` functions might be used:

```
SELECT e
FROM Employee AS e
WHERE
LENGTH(e.name) > 6
  AND
LOCATE(e.name, 'Monson') > -1
```

This statement selects all the employees with `Monson` somewhere in their last names but specifies that the names must be longer than six characters. Therefore, "Monson-Haefel" and "Monson-Ares" would evaluate to `true`, but "Monson" would return `false` because it has only six characters.

The arithmetic functions in JPA QL may be applied to primitive as well as corresponding primitive wrapper types:

`ABS(number)`
: Returns the absolute value of a number (`int`, `float`, or `double`)

`SQRT(double)`
: Returns the square root of a `double`

`MOD(int, int)`
: Returns the remainder for the first parameter divided by the second (e.g., `MOD(7, 5)` is equal to 2)

Functions returning dates and times

JPA QL has three functions that can return you the current date, time, and timestamp: `CURRENT_DATE`, `CURRENT_TIME`, and `CURRENT_TIMESTAMP`. Here's an example of searching for employees who joined the company on the current date:

```
SELECT e FROM Employee e WHERE e.startDate = CURRENT_DATE
```

Aggregate functions in the SELECT clause

Aggregate functions are used with queries that return a collection of values. They are fairly simple to understand and can be handy, especially the `COUNT()` function.

COUNT (identifier or path expression). This function returns the number of items in the query's final result set. The return type is a `java.lang.Long`, depending on whether it is the return type of the query method. For example, the following query provides a count of all the customers who live in Wisconsin:

```
SELECT COUNT( c )
FROM Customers AS c
WHERE c.address.state = 'WI'
```

The `COUNT()` function can be used with an identifier, in which case it always counts entities, or with path expressions, in which case it counts fields. For example, the following statement provides a count of all the zip codes that start with the numbers 554:

```
SELECT COUNT(c.address.zip)
FROM Customers AS c
WHERE c.address.zip LIKE '554%'
```

In some cases, queries that count a path expression have a corresponding query that can be used to count an identifier. For example, the result of the following query, which counts `Customers` instead of the `zip` field, is equivalent to the previous query:

```
SELECT COUNT( c )
FROM Customers AS c
WHERE c.address.zip LIKE '554%'
```

MAX(path expression), MIN(path expression). These functions can be used to find the largest or smallest value from a collection of any type of field. They cannot be used with identifiers or paths that terminate in a relationship field. The result type will be the type of field that is being evaluated. For example, the following query returns the highest price paid for an order:

```
SELECT MAX( o.totalAmount )
FROM ProductOrder AS o
```

The MAX() and MIN() functions can be applied to any valid value, including primitive types, strings, and even serializable objects. The result of applying the MAX() and MIN() functions to serializable objects is not specified, because there is no standard way to determine which serializable object is greater than or less than another one.

The result of applying the MAX() and MIN() functions to a String field depends on the underlying data store. This has to do with the inherent problems associated with String type comparisons.

AVG(numeric), SUM(numeric). The AVG() and SUM() functions can be applied only to path expressions that terminate in a numeric primitive field (byte, long, float, etc.) or in one of their corresponding numeric wrappers (Byte, Long, Float, etc.). The result of a query that uses the SUM() function has the same type as the numeric type it's evaluating. The result type of the AVG() function is a java.lang.Double, depending on whether it is used in the return type of the SELECT method.

For example, the following query uses the SUM() function to get the total amount for all orders:

```
SELECT SUM( o.totalAmount )
FROM ProductOrder o
```

DISTINCT, nulls, and empty arguments. The DISTINCT operator can be used with any of the aggregate functions to eliminate duplicate values. The following query uses the DISTINCT operator to count the number of *different* zip codes that match the pattern specified:

```
SELECT DISTINCT COUNT(c.address.zip)
FROM Customers AS c
WHERE c.address.zip LIKE '554%'
```

The DISTINCT operator first eliminates duplicate zip codes. If 100 customers live in the same area with the same zip code, their zip code is counted only once. After the duplicates have been eliminated, the COUNT() function counts the number of items left.

Any field with a null value is automatically eliminated from the result set operated on by the aggregate functions. The COUNT() function also ignores values with null values. The aggregate functions AVG(), SUM(), MAX(), and MIN() return null when evaluating an empty collection.

The COUNT() function returns 0 (zero) when the argument it evaluates is an empty collection. If the following query is evaluated on an order with no line items, the result is 0 (zero) because the argument is an empty collection:

```
SELECT COUNT( li )
FROM ProductOrder AS o, IN( o.lineItems ) AS li
WHERE o = ?1
```

The ORDER BY Clause

The `ORDER BY` clause allows you to specify the order of the entities in the collection returned by a query. The semantics of the `ORDER BY` clause are basically the same as in SQL. For example, we can construct a simple query that uses the `ORDER BY` clause to return an alphabetical list of all employees:

```
SELECT e
FROM Employee AS e
ORDER BY e.name
```

This will return a `Collection` of `Employee` entities in alphabetical order by name.

You can use the `ORDER BY` clause with or without the `WHERE` clause. For example, we can refine the previous query by listing only those U.S. employees who reside in Boston:

```
SELECT e
FROM Employee AS e
WHERE e.address.city = 'Boston' AND e.address.state = 'MA'
ORDER BY e.name
```

The default order of an item listed in the `ORDER BY` clause is always ascending, which means that the lesser values are listed first and the greater values last. You can explicitly specify the order as *ascending* or *descending* by using the keywords `ASC` and `DESC`. The default is `ASC`. Null elements will be placed either on top or at the bottom of the query result, depending on the underlying database. Here's a statement that lists customers in reverse (descending) order:

```
SELECT e
FROM Employee AS e
ORDER BY e.name DESC
```

You can specify multiple order-by items. For example, you can sort customers by `lastName` in ascending order and `firstName` in descending order:

```
SELECT c
FROM Customers AS c
ORDER BY c.lastName ASC, c.firstName DESC
```

If you have five `Customer` entities with the `lastName` equal to `Brooks`, this query sorts the results as follows:

```
Brooks, William
Brooks, Henry
Brooks, Hank
Brooks, Ben
Brooks, Andy
```

Although the fields used in the `ORDER BY` clause must be basic fields, the value selected can be an entity identifier, a relationship field, or a basic field. For example, the following query returns an ordered list of all zip codes:

```
SELECT addr.zip
FROM Address AS addr
ORDER BY addr.zip
```

The following query returns all the `Address` entities for customers named Smith, ordered by their zip code:

```
SELECT c.address
FOR Customer AS c
WHERE c.lastName = 'Smith'
ORDER BY c.address.zip
```

You must be careful which basic fields you use in the `ORDER BY` clause. If the query selects a collection of entities, then the `ORDER BY` clause can be used with only basic fields of the entity type that is selected. The following query is illegal, because the basic field used in the `ORDER BY` clause is not a field of the entity type selected:

```
// Illegal JPA QL
SELECT e
FROM Employee AS e
ORDER BY e.address.city
```

Because the `city` field is not a direct field of the `Employee` entity, you cannot use it in the `ORDER BY` clause.

A similar restriction applies to results. The field used in the `ORDER BY` clause must also be in the `SELECT` clause. The following query is illegal because the field identified in the `SELECT` clause is not the same as the one used in the `ORDER BY` clause:

```
// Illegal JPA QL
SELECT e.address.city
FROM Employee AS e
ORDER BY e.address.state
```

In the previous query, we wanted a list of all the cities ordered by their state. Unfortunately, this is illegal. You can't order by the `state` field if you are not selecting the `state` field.

Bulk UPDATE and DELETE

Java Persistence has the ability to perform bulk `UPDATE` and `DELETE` operations. This can save you a lot of typing. For example, let's say we want to give a $10 credit across the board to any customer named Bill Burke. We can do the following bulk `UPDATE`:

```
UPDATE ProductOrder o SET o.amountPaid = (o.amountPaid + 10)
WHERE EXISTS (
   SELECT c FROM o.customer c
   WHERE c.name = 'Bill Burke'
)
```

As an example of `DELETE`, say that we want to remove all orders placed by Bill Burke:

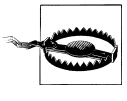
```
DELETE FROM ProductOrder o
WHERE EXISTS (
   SELECT c FROM o.customer c
   WHERE c.name = 'Bill Burke'
)
```

Be very careful how you use bulk `UPDATE` and `DELETE`. It is possible, depending on the vendor implementation, to create inconsistencies between the database and entities that are already being managed by the current persistence context. Vendor implementations are required only to execute the update or delete directly on the database. They do not have to modify the state of any currently managed entity. For this reason, it is recommended that you do these operations within their own transaction or at the beginning of a transaction (before any entities are accessed that might be affected by these bulk operations). Alternatively, executing `EntityManager.flush()` and `EntityManager.clear()` before executing a bulk operation will keep you safe.

Native Queries

JPA QL is a very rich syntax and should meet most of your querying needs. Sometimes, though, you want to take advantage of certain proprietary capabilities that are available only on a specific vendor's database.

The entity manager service provides a way to create native SQL queries and map them to your objects. Native queries can return entities, column values, or a combination of the two. The `EntityManager` interface has three methods for creating native queries: one for returning scalar values, one for returning one entity type, and one for defining a complex result set that can map to a mix of multiple entities and scalar values.



You can always get the underlying JDBC connection through a `javax.sql.DataSource` injected by the `@Resource` and execute any SQL statement you need. Be aware that your changes will not be reflected in the current persistence context.

Scalar Native Queries

```
Query createNativeQuery(String sql)
```

This creates a native query that returns scalar results. It takes one parameter: your native SQL. It executes as is and returns the result set in the same form as JPA QL returns scalar values.

Simple Entity Native Queries

```
Query createNativeQuery(String sql, Class entityClass)
```

A simple entity native query takes an SQL statement and implicitly maps it to one entity based on the mapping metadata you declared for that entity. It expects that the columns returned in the result set of the native query will match perfectly with the entity's O/R mapping. The entity that the native SQL query maps to is determined by the `entityClass` parameter:

```
Query query = manager.createNativeQuery(
    "SELECT p.phone_PK, p.phone_number, p.type
```

```
        FROM PHONE AS p", Phone.class
);
```

All the properties of the entities must be pulled.

Complex Native Queries

```
Query createNativeQuery(String sql, String mappingName)
```

This entity manager method allows you to have complex mappings for your native SQL. You can return multiple entities and scalar column values at the same time. The `mappingName` parameter references a declared `@javax.persistence.SqlResultSetMapping`. This annotation is used to define how the native SQL results hook back into your O/R model. If your returned column names don't match the parallel annotated property mapping, you can provide a field-to-column mapping for them using `@javax.persistence.FieldResult`:

```
package javax.persistence;

public @interface SqlResultSetMapping {
   String name( );
   EntityResult[] entities( ) default {};
   ColumnResult[] columns( ) default {};
}

public @interface EntityResult {
   Class entityClass( );
   FieldResult[] fields( ) default {};
   String discriminatorColumn( ) default "";
}

public @interface FieldResult {
   String name( );
   String column( );
}

public @interface ColumnResult {
   String name( );
}
```

Let's create an example to show how this would work.

Native queries with multiple entities

First, let's create a native query that returns multiple entity types—a `Customer` and its `CreditCard`:

```
@Entity
@SqlResultSetMapping(name="customerAndCreditCardMapping",
      entities={@EntityResult(entityClass=Customer.class),
                @EntityResult(entityClass=CreditCard.class,
                              fields={@FieldResult(name="id",
                                                   column="CC_ID"),
                                      @FieldResult(name="number",
```

```
                                                    column="number")}
                                )})
public class Customer {...}

// execution code
{
   Query query = manager.createNativeQuery(
        "SELECT c.id, c.firstName, cc.id As CC_ID,
 cc.number" +
                "FROM CUST_TABLE c, CREDIT_CARD_TABLE cc" +
                "WHERE c.credit_card_id = cc.id",
        "customerAndCreditCardMapping");
}
```

Because the result set returns multiple entity types, we must define an `@SqlResultSetMapping`. This annotation can be placed on an entity class or method. The `entities()` attribute is set to an array of `@EntityResult` annotations. Each `@EntityResult` annotation specifies the entities that will be returned by the native SQL query.

The `@javax.persistence.FieldResult` annotation is used to explicitly map columns in the query with properties of an entity. The `name()` attribute of `@FieldResult` identifies the entity bean's property, and the `column()` attribute identifies the result set column returned by the native query.

In this example, we do not need to specify any `@FieldResult`s for `Customer`, as the native query pulls in each column for this entity. However, since we are querying only the ID and number columns of the `CreditCard` entity, an `@FieldResult` annotation should be specified. In the `@EntityResult` annotation for `CreditCard`, the `fields()` attribute defines what `CreditCard` properties each queried column maps to. Because the `Customer` and `CreditCard` primary-key columns have the same name, the SQL query needs to distinguish that they are different. The `cc.id As CC_ID` SQL fragment performs this identification.

Named Queries

Java Persistence provides a mechanism so that you can predefine JPA QL or native SQL queries and reference them by name when creating a query. You would want to predeclare queries for the same reason you create `String` constant variables in Java: to reuse them in multiple situations. If you predefine your queries in one place, you have an easy way to fine-tune or modify them as time goes on. The `@javax.persistence.NamedQuery` annotation is used for predefining JPA QL queries:

```
package javax.persistence;

public @interface NamedQuery {
   String name( );
   String query( );
   QueryHint[] hints( ) default {};
}
```

```
public @interface QueryHint {
   String name( );
   String value( );
}

public @interface NamedQueries {
   NamedQuery[] value( );
}
```

You use the `@javax.persistence.NamedQueries` annotation when you are declaring more than one query on a class or package. The `@javax.persistence.QueryHint` annotation declares vendor-specific hints. These hints work in the same way as the `Query.setHint()` method described earlier in this chapter. Here's an example:

```
@NamedQueries({
   @NamedQuery
(name="getAverageProductOrderAmount",
               query=
                 "SELECT AVG( o.amountPaid )
                  FROM ProductOrder as o")
})
@Entity
public class ProductOrder {...}
```

This example declares a JPA QL query on the `ProductOrder` entity bean class. You can then reference these declarations in the `EntityManager.createNamedQuery()` method:

```
Query query = em.createNamedQuery("getAverageProductOrderAmount");
```

Named Native Queries

The `@javax.persistence.NamedNativeQuery` annotation is used for predefining native SQL queries:

```
package javax.persistence;

public @interface NamedNativeQuery {
   String name( );
   String query( );
   Class resultClass( ) default void.class;
   String resultSetMapping( ) default "";
}

public @interface NamedNativeQueries {
   NamedNativeQuery[] value( );
}
```

The `resultClass()` attribute is for when you have a native query that returns only one entity type (see "Native Queries" on page 230). The `resultSetMapping()` attribute must resolve to a predeclared `@SqlResultSetMapping`. Both attributes are optional, but you must declare at least one of them. Here is an example of predeclaring an `@NamedNativeQuery`:

```
@NamedNativeQuery(
   name="findCustAndCCNum",
   query="SELECT c.id, c.firstName, c.lastName, cc.number AS CC_NUM
               FROM CUST_TABLE c, CREDIT_CARD_TABLE cc
               WHERE c.credit_card_id = cc.id",
   resultSetMapping="customerAndCCNumMapping")
@SqlResultSetMapping(name="customerAndCCNumMapping",
      entities={@EntityResult(entityClass=Customer.class)},
      columns={@ColumnResult(name="CC_NUM")}
)
@Entity
public class Customer {...}
```

You can then reference this declaration in the `EntityManager.createNamedQuery()` method:

```
Query query = em.createNamedQuery("findCustAndCCNum");
```

CHAPTER 14

Entity Callbacks and Listeners

When you execute `EntityManager` methods such as `persist()`, `merge()`, `remove()`, and `find()`, or when you execute JPA QL queries or use the `Criteria` API, a predefined set of lifecycle events are triggered. For instance, the `persist()` method triggers database inserts. Merging triggers updates to the database. The `remove()` method triggers database deletes. Querying entities triggers a load from the database. Sometimes it is very useful to have your entity bean class be notified as these events happen. For instance, maybe you want to create an audit log of every interaction done on each row in your database. The Java Persistence specification allows you to set up callback methods on your entity classes so that your entity instances are notified when these events occur. You can also register separate listener classes that can intercept these same events. These are called *entity listeners*. This chapter discusses how you register your entity bean classes for lifecycle callbacks as well as how to write entity listeners that can intercept lifecycle events on your entities.

Callback Events

An annotation may represent each phase of an entity's lifecycle:

```
@javax.persistence.PrePersist
@javax.persistence.PostPersist
@javax.persistence.PostLoad
@javax.persistence.PreUpdate
@javax.persistence.PostUpdate
@javax.persistence.PreRemove
@javax.persistence.PostRemove
```

The `@PrePersist` and `@PostPersist` events have to do with the insertion of an entity instance into the database. The `@PrePersist` event occurs immediately when the `EntityManager.persist()` call is invoked or whenever an entity instance is scheduled to be inserted into the database (as with a cascaded merge). The `@PostPersist` event is not triggered until the actual database insert.

The `@PreUpdate` event is triggered just before the state of the entity is synchronized with the database, and the `@PostUpdate` event happens after. This synchronization could occur at transaction commit time, when `EntityManager.flush()` is executed, or whenever the persistence context deems it necessary to update the database.

The `@PreRemove` and `@PostRemove` events have to do with the removal of an entity bean from the database. `@PreRemove` is triggered whenever `EntityManager.remove()` is invoked on the entity bean, directly or because of a cascade. The `@PostRemove` event happens immediately after the actual database delete occurs.

The `@PostLoad` event is triggered after an entity instance has been loaded by a `find()` or `getReference()` method call on the `EntityManager` interface, or when a JPA QL or `Criteria` query is executed. It is also called after the `refresh()` method is invoked.

Callbacks on Entity Classes

You can have an entity bean instance register for a callback on any of these lifecycle events by annotating a public, private, protected, or package-protected method on the bean class. This method must return `void`, throw no checked exceptions, and have no arguments:

```
import javax.persistence.Entity;
import javax.persistence.PostPersist;
import javax.persistence.PrePersist;

/**
 * Represents an Employee which is able to receive JPA
 * events.
 */
@Entity
public class EntityListenerEmployee
{

   private String name;

   public String getName() { return name; }
   public void setName(final String name) { this.name = name; }

   /*
    * Event Listeners; fired by JPA and track state in the EventTracker
    */

   @PrePersist
   @SuppressWarnings("unused")
   private void prePersist(){ ... }

   @PostPersist
   @SuppressWarnings("unused")
   private void postPersist(){ ... }
```

```
    ...
}
```

When an event is triggered on a particular managed entity instance, the entity manager will invoke the appropriate annotated method on the entity bean class.

Entity Listeners

Entity listeners are classes that can generically intercept entity callback events. They are not entity classes themselves, but they can be attached to an entity class through a binding annotation or XML. You can assign methods on an entity listener class to intercept a particular lifecycle event. These methods return `void` and take one `Object` parameter that is the entity instance on which the event is being triggered. The method is annotated with the callback in which it is interested:

```
public class Auditor {

   @PostPersist void postInsert(final Object entity)
   {
      System.out.println("Inserted entity: " + entity.getClass().getName( ));
   }

   @PostLoad void postLoad(final Object entity)
   {
      System.out.println("Loaded entity: " + entity.getClass().getName( ));
   }

}
```

The entity listener class must have a public no-arg constructor. It can be applied to an entity class by using the `@javax.persistence.EntityListeners` annotation:

```
package javax.persistence;

@Target(TYPE) @Retention(RUNTIME)
public @interface EntityListeners
{
   Class[] value();
}
```

You may specify one or more entity listeners that intercept the callback events of an entity class:

```
@Entity
@EntityListeners  ({Auditor.class})
public class EntityListenerEmployee
{
  ...
}
```

By using the `@EntityListeners` annotation on the `EntityListenerEmployee` entity class, any callback methods within those entity listener classes will be invoked whenever `EntityListenerEmployee` entity instances interact with a persistence context.

Default Entity Listeners

You can specify a set of default entity listeners that are applied to every entity class in the persistence unit by using the `<entity-listeners>` element under the top-level `<entity-mappings>` element in the ORM mapping file. For instance, if you wanted to apply the `Auditor` listener to every entity class in a particular persistence unit, you would do the following:

```
<entity-mappings>
   <entity-listeners>
      <entity-listener class="org.package.Auditor">
         <post-persist name="postInsert"/>
         <post-load name="postLoad"/>
      </entity-listener>
   </entity-listeners>
</entity-mappings
```

If you want to turn off default entity listeners to a particular entity class, you can use the `@javax.persistence.ExcludeDefaultListeners` annotation:

```
@Entity
@ExcludeDefaultListeners
public class NoEntityListenersEmployee {
...
}
```

If either the `@ExcludeDefaultListeners` annotation or its XML equivalent is applied to the `EntityListenerEmployee` entity, the `Auditor` is turned off for that entity.

Inheritance and Listeners

If you have an inheritance entity hierarchy in which the base class has entity listeners applied to it, any subclass will inherit these entity listeners. If the subclass also has entity listeners applied to it, then both the base and the subclass's listeners will be attached:

```
@Entity
@EntityListeners(Auditor.class)
public class SingleEntityListenerEmployee
{
   ...
}

@Entity
@EntityListeners(AnotherListener.class)
public class DoubleEntityListenerEmployee extends SingleEntityListenerEmployee
{
   ...
}
```

In this example, the `Auditor` entity listener and `AnotherListener` will be attached to the `DoubleEntityListenerEmployee` entity. If all of these listeners have an `@PostPersist` callback, the order of callback execution will be as follows:

1. `Auditor`'s `@PostPersist` method
2. `AnotherListener`'s `@PostPersist` method

Entity listeners applied to a base class happen before any listeners attached to a subclass. Callback methods defined directly in an entity class happen last.

You can turn off inherited entity listeners by using `@javax.persistence.ExcludeSuperclassListeners`:

```
@Entity
@EntityListeners(Auditor.class)
public class EntityListenerEmployee {

}

@Entity
@ExcludeSuperclassListeners

public class NoEntityListenersEmployee extends EntityListenerEmployee {
...
}
```

In this example, no listeners would be executed for `NoEntityListenerEmployee` entity instances. `@ExcludeSuperclassListeners` has an XML equivalent in the `<exclude-superclass-listeners/>` element.