JPA

*Course Material for Java Persistence API*

Wriiten by : Mahendra Shinde

Java Persistence API (JPA)

### Introduction to the Java Persistence API

Working with object-oriented software and a relational database can be cumbersome and time consuming in today's enterprise environments. JPA is an Object/Relational Mapping tool for Java environments. The term Object/Relational Mapping (ORM) refers to the technique of mapping a data representation from an object model to a relational data model with a SQL-based schema.

## Object / Relational Mapping

**Object-relational mapping** (**ORM**, **O/RM**, and **O/R mapping**) in computer software is a programming technique for converting data between incompatible type systems in object-oriented programming languages. This creates, in effect, a "virtual object database" that can be used from within the programming language. There are both free and commercial packages available that perform object-relational mapping, although some programmers opt to create their own ORM tools.

Data management tasks in object-oriented (OO) programming are typically implemented by manipulating objects that are almost always non-scalar values. For example, consider an address book entry that represents a single person along with zero or more phone numbers and zero or more addresses. This could be modeled in an object-oriented implementation by a "Person object" with attributes/fields to hold each data item that the entry comprises: the person's name, a list of phone numbers, and a list of addresses. The list of phone numbers would itself contain "PhoneNumber objects" and so on. The address book entry is treated as a single object by the programming language (it can be referenced by a single variable containing a pointer to the object, for instance). Various methods can be associated with the object, such as a method to return the preferred phone number, the home address, and so on.

However, many popular database products such as structured query language database management systems (SQL DBMS) can only store and manipulate scalar values such as integers and strings organized within tables. The programmer must either convert the object values into groups of simpler values for storage in the database (or convert them back upon retrieval), or only use simple scalar values within the program. Object-relational mapping is used to implement the first approach.

The heart of the problem is translating the logical representation of the objects into an atomized form that is capable of being stored on the database, while somehow preserving the properties of the objects and their relationships so that they can be reloaded as an object when needed. If this storage and retrieval functionality is implemented, the objects are then said to be persistent.

Compared to traditional techniques of exchange between an object-oriented language and a relational database, ORM often reduces the amount of code that needs to be written

Disadvantages of O/R mapping tools are in areas where proprietary, or database-specific techniques have been highly optimized. Most O/R mapping tools do not perform well during bulk deletions of data or joins. Stored procedures may have better performance, but are not portable.

In addition, heavy reliance on ORM software has been pointed to as a major factor in producing poorly designed databases.

### Entities

An entity is a lightweight persistence domain object. Typically an entity represents a table in a relational database, and each entity instance corresponds to a row in that table. The primary programming artifact of an entity is the entity class, although entities can use helper classes.

The persistent state of an entity is represented either through persistent fields or persistent properties. These fields or properties use object/relational mapping annotations to map the entities and entity relationships to the relational data in the underlying data store.

### Requirements for Entity Classes

An entity class must follow these requirements:

* The class must be annotated with the javax.persistence.Entity annotation.
* The class must have a public or protected, no-argument constructor. The class may have other constructors.
* The class must not be declared final. No methods or persistent instance variables must be declared final.
* If an entity instance be passed by value as a detached object, such as through a session bean’s remote business interface, the class must implement the Serializable interface.
* Entities may extend both entity and non-entity classes, and non-entity classes may extend entity classes.
* Persistent instance variables must be declared private, protected, or package-private, and can only be accessed directly by the entity class’s methods. Clients must access the entity’s state through accessor or business methods.

### Persistent Fields and Properties in Entity Classes

The persistent state of an entity can be accessed either through the entity’s instance variables or through JavaBeans-style properties. The fields or properties must be of the following Java language types:

* Java primitive types
* java.lang.String
* Other serializable types including:
  + Wrappers of Java primitive types
  + java.math.BigInteger
  + java.math.BigDecimal
  + java.util.Date
  + java.util.Calendar
  + java.sql.Date
  + java.sql.Time
  + java.sql.TimeStamp
  + User-defined serializable types
  + byte[]
  + Byte[]
  + char[]
  + Character[]
* Enumerated types
* Other entities and/or collections of entities
* Embeddable classes

Entities may either use persistent fields or persistent properties. If the mapping annotations are applied to the entity’s instance variables, the entity uses persistent fields. If the mapping annotations are applied to the entity’s getter methods for JavaBeans-style properties, the entity uses persistent properties. You cannot apply mapping annotations to both fields and properties in a single entity.

### Persistent Fields

If the entity class uses persistent fields, the Persistence runtime accesses entity class instance variables directly. All fields not annotated javax.persistence.Transient or not marked as Java transient will be persisted to the data store. The object/relational mapping annotations must be applied to the instance variables.

### Persistent Properties

If the entity uses persistent properties, the entity must follow the method conventions of JavaBeans components. JavaBeans-style properties use getter and setter methods that are typically named after the entity class’s instance variable names. For every persistent property *property* of type *Type* of the entity, there is a getter method get*Property* and setter method set*Property*. If the property is a boolean, you may use is*Property* instead of get*Property*. For example, if a Customer entity uses persistent properties, and has a private instance variable called firstName, the class defines a getFirstName and setFirstName method for retrieving and setting the state of the firstName instance variable.

The method signature for single-valued persistent properties are as follows:

Type getProperty()

void setProperty(Type type)

Collection-valued persistent fields and properties must use the supported Java collection interfaces regardless of whether the entity uses persistent fields or properties. The following collection interfaces may be used:

* java.util.Collection
* java.util.Set
* java.util.List
* java.util.Map

If the entity class uses persistent fields, the type in the above method signatures must be one of these collection types. Generic variants of these collection types may also be used. For example, if the Customer entity has a persistent property that contains a set of phone numbers, it would have the following methods:

Set<PhoneNumber> getPhoneNumbers() {}

void setPhoneNumbers(Set<PhoneNumber>) {}

The object/relational mapping annotations for must be applied to the getter methods. Mapping annotations cannot be applied to fields or properties annotated @Transient or marked transient.

### Primary Keys in Entities

Each entity has a unique object identifier. A customer entity, for example, might be identified by a customer number. The unique identifier, or **primary key**, enables clients to locate a particular entity instance. Every entity must have a primary key. An entity may have either a simple or a composite primary key.

Simple primary keys use the javax.persistence.Id annotation to denote the primary key property or field.

Composite primary keys must correspond to either a single persistent property or field, or to a set of single persistent properties or fields. Composite primary keys must be defined in a primary key class. Composite primary keys are denoted using the javax.persistence.EmbeddedId and javax.persistence.IdClass annotations.

The primary key, or the property or field of a composite primary key, must be one of the following Java language types:

* Java primitive types
* Java primitive wrapper types
* java.lang.String
* java.util.Date (the temporal type should be DATE)
* java.sql.Date

Floating point types should never be used in primary keys. If you use a generated primary key, only integral types will be portable.

### Primary Key Classes

A primary key class must meet these requirements:

* The access control modifier of the class must be public.
* The properties of the primary key class must be public or protected if property-based access is used.
* The class must have a public default constructor.
* The class must implement the hashCode() and equals(Object other) methods.
* The class must be serializable.
* A composite primary key must be represented and mapped to multiple fields or properties of the entity class, or must be represented and mapped as an embeddable class.
* If the class is mapped to multiple fields or properties of the entity class, the names and types of the primary key fields or properties in the primary key class must match those of the entity class.

The following primary key class is a composite key, the orderId and itemId fields together uniquely identify an entity.

public final class LineItemKey implements Serializable {

public Integer orderId;

public int itemId;

public LineItemKey() {}

public LineItemKey(Integer orderId, int itemId) {

this.orderId = orderId;

this.itemId = itemId;

}

public boolean equals(Object otherOb) {

if (this == otherOb) {

return true;

}

if (!(otherOb instanceof LineItemKey)) {

return false;

}

LineItemKey other = (LineItemKey) otherOb;

return (

(orderId==null?other.orderId==null:orderId.equals

(other.orderId)

)

&&

(itemId == other.itemId)

);

}

public int hashCode() {

return (

(orderId==null?0:orderId.hashCode())

^

((int) itemId)

);

}

public String toString() {

return "" + orderId + "-" + itemId;

}

}

## Associations

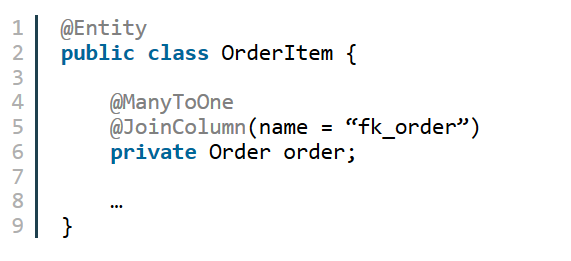
Association mappings are often the most difficult thing to implement correctly. In this section we examine some canonical cases one by one, starting with unidirectional mappings and then bidirectional cases. We will use Person and Address in all the examples

Associations will be classified by multiplicity and whether or not they map to an intervening join table. Nullable foreign keys are not considered to be good practice in traditional data modelling, so our examples do not use nullable foreign keys. This is not a requirement of Hibernate, and the mappings will work if you drop the nullability constraints.

1. Unidirectional associations
   1. Many to One

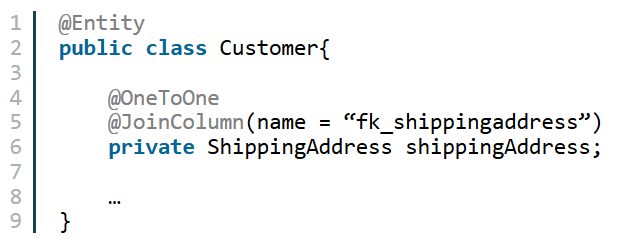
A *unidirectional many-to-one association* is the most common kind of unidirectional association.

Example:



* 1. One to One

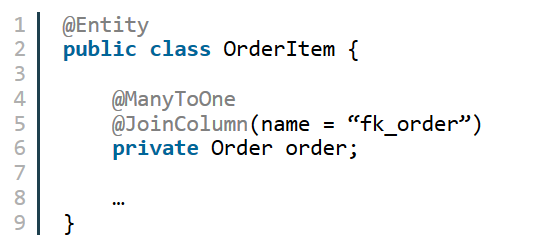
A unidirectional one-to-one association on a foreign key is almost identical. The only difference is the column unique constraint.

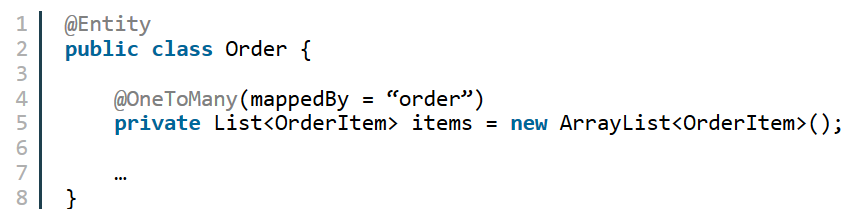


1. Bi-directional Associations
   1. one to many/many to one

A *bidirectional many-to-one association* is the most common kind of association. The following

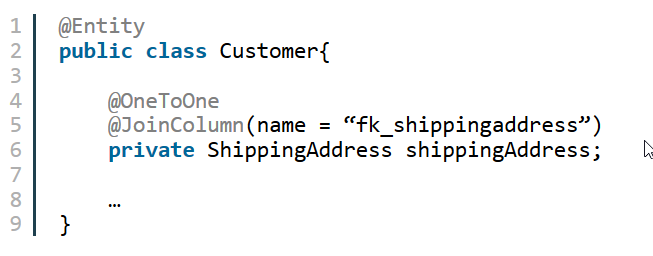
example illustrates the standard parent/child relationship.

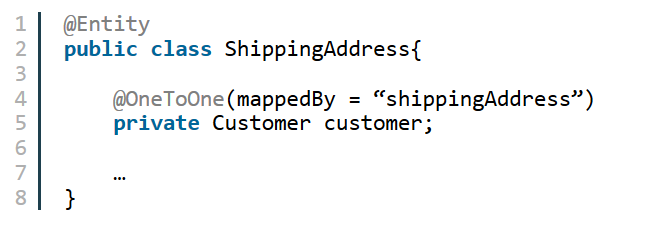




* 1. One to One

A bidirectional one-to-one association on a foreign key is common





### Queries and Relationship Direction

Java Persistence query language queries often navigate across relationships. The direction of a relationship determines whether a query can navigate from one entity to another. For example, a query can navigate from LineItem to Product but cannot navigate in the opposite direction. For Order and LineItem, a query could navigate in both directions, because these two entities have a bidirectional relationship.

### Cascade Deletes and Relationships

Entities that use relationships often have dependencies on the existence of the other entity in the relationship. For example, a line item is part of an order, and if the order is deleted, then the line item should also be deleted. This is called a cascade delete relationship.

Cascade delete relationships are specified using the cascade=REMOVE element specification for @OneToOne and @OneToMany relationships. For example:

@OneToMany(cascade=REMOVE, mappedBy="customer")

public Set<Order> getOrders() { return orders; }

### Entity Inheritance

Entities support class inheritance, polymorphic associations, and polymorphic queries. They can extend non-entity classes, and non-entity classes can extend entity classes. Entity classes can be both abstract and concrete.

### Abstract Entities

An abstract class may be declared an entity by decorating the class with @Entity. Abstract entities differ from concrete entities only in that they cannot be instantiated.

Abstract entities can be queried just like concrete queries. If an abstract entity is the target of a query, the query operates on all the concrete subclasses of the abstract entity.

@Entity

public abstract class Employee {

@Id

protected Integer employeeId;

...

}

@Entity

public class FullTimeEmployee extends Employee {

protected Integer salary;

...

}

@Entity

public class PartTimeEmployee extends Employee {

protected Float hourlyWage;

}

### Mapped Superclasses

Entities may inherit from superclasses that contain persistent state and mapping information, but are not entities. That is, the superclass is not decorated with the @Entity annotation, and is not mapped as an entity by the Java Persistence provider. These superclasses are most often used when you have state and mapping information common to multiple entity classes.

Mapped superclasses are specified by decorating the class with the javax.persistence.MappedSuperclass annotation.

@MappedSuperclass

public class Employee {

@Id

protected Integer employeeId;

...

}

@Entity

public class FullTimeEmployee extends Employee {

protected Integer salary;

...

}

@Entity

public class PartTimeEmployee extends Employee {

protected Float hourlyWage;

...

}

Mapped superclasses are not queryable, and can’t be used in EntityManager or Query operations. You must use entity subclasses of the mapped superclass in EntityManager or Query operations. Mapped superclasses can’t be targets of entity relationships. Mapped superclasses can be abstract or concrete.

Mapped superclasses do not have any corresponding tables in the underlying datastore. Entities that inherit from the mapped superclass define the table mappings. For instance, in the code sample above the underlying tables would be FULLTIMEEMPLOYEE and PARTTIMEEMPLOYEE, but there is no EMPLOYEE table.

### Non-Entity Superclasses

Entities may have non-entity superclasses, and these superclasses can be either abstract or concrete. The state of non-entity superclasses is non-persistent, and any state inherited from the non-entity superclass by an entity class is non-persistent. Non-entity superclasses may not be used in EntityManager or Query operations. Any mapping or relationship annotations in non-entity superclasses are ignored.

### Entity Inheritance Mapping Strategies

You can configure how the Java Persistence provider maps inherited entities to the underlying datastore by decorating the root class of the hierarchy with the javax.persistence.Inheritance annotation. There are three mapping strategies that are used to map the entity data to the underlying database:

* A single table per class hierarchy
* A table per concrete entity class
* A “join” strategy, where fields or properties that are specific to a subclass are mapped to a different table than the fields or properties that are common to the parent class

The strategy is configured by setting the strategy element of @Inheritance to one of the options defined in the javax.persistence.InheritanceType enumerated type:

public enum InheritanceType {

SINGLE\_TABLE,

JOINED,

TABLE\_PER\_CLASS

};

The default strategy is InheritanceType.SINGLE\_TABLE, and is used if the @Inheritance annotation is not specified on the root class of the entity hierarchy.

### The Single Table per Class Hierarchy Strategy

With this strategy, which corresponds to the default InheritanceType.SINGLE\_TABLE, all classes in the hierarchy are mapped to a single table in the database. This table has a **discriminator column**, a column that contains a value that identifies the subclass to which the instance represented by the row belongs.

The discriminator column can be specified by using the javax.persistence.DiscriminatorColumn annotation on the root of the entity class hierarchy.

###### Table 24-1 @DiscriminatorColumn Elements

|  |  |  |
| --- | --- | --- |
| **Type** | **Name** | **Description** |
| String | name | The name of the column in the table to be used as the discriminator column. The default is DTYPE. This element is optional. |
| DiscriminatorType | discriminatorType | The type of the column to be used as a discriminator column. The default is DiscriminatorType.STRING. This element is optional. |
| String | columnDefinition | The SQL fragment to use when creating the discriminator column. The default is generated by the Persistence provider, and is implementation-specific. This element is optional. |
| String | length | The column length for String-based discriminator types. This element is ignored for non-String discriminator types. The default is 31. This element is optional. |

The javax.persistence.DiscriminatorType enumerated type is used to set the type of the discriminator column in the database by setting the discriminatorType element of @DiscriminatorColumn to one of the defined types. DiscriminatorType is defined as:

public enum DiscriminatorType {

STRING,

CHAR,

INTEGER

};

If @DiscriminatorColumn is not specified on the root of the entity hierarchy and a discriminator column is required, the Persistence provider assumes a default column name of DTYPE, and column type of DiscriminatorType.STRING.

The javax.persistence.DiscriminatorValue annotation may be used to set the value entered into the discriminator column for each entity in a class hierarchy. You may only decorate concrete entity classes with @DiscriminatorValue.

If @DiscriminatorValue is not specified on an entity in a class hierarchy that uses a discriminator column, the Persistence provider will provide a default, implementation-specific value. If the discriminatorType element of @DiscriminatorColumn is DiscriminatorType.STRING, the default value is the name of the entity.

This strategy provides good support for polymorphic relationships between entities and queries that cover the entire entity class hierarchy. However, it requires the columns that contain the state of subclasses to be nullable.

### The Table per Concrete Class Strategy

In this strategy, which corresponds to InheritanceType.TABLE\_PER\_CLASS, each concrete class is mapped to a separate table in the database. All fields or properties in the class, including inherited fields or properties, are mapped to columns in the class’s table in the database.

This strategy provides poor support for polymorphic relationships, and usually requires either SQL UNION queries or separate SQL queries for each subclass for queries that cover the entire entity class hierarchy.

Support for this strategy is optional, and may not be supported by all Java Persistence API providers. The default Java Persistence API provider in the Application Server does not support this strategy.

### The Joined Subclass Strategy

In this strategy, which corresponds to InheritanceType.JOINED, the root of the class hierarchy is represented by a single table, and each subclass has a separate table that only contains those fields specific to that subclass. That is, the subclass table does not contain columns for inherited fields or properties. The subclass table also has a column or columns that represent its primary key, which is a foreign key to the primary key of the superclass table.

This strategy provides good support for polymorphic relationships, but requires one or more join operations to be performed when instantiating entity subclasses. This may result in poor performance for extensive class hierarchies. Similarly, queries that cover the entire class hierarchy require join operations between the subclass tables, resulting in decreased performance.

Some Java Persistence API providers, including the default provider in the Application Server, require a discriminator column in the table that corresponds to the root entity when using the joined subclass strategy. If you are not using automatic table creation in your application, make sure the database table is set up correctly for the discriminator column defaults, or use the @DiscriminatorColumn annotation to match your database schema

### Managing Entities

Entities are managed by the entity manager. The entity manager is represented by javax.persistence.EntityManager instances. Each EntityManager instance is associated with a persistence context. A persistence context defines the scope under which particular entity instances are created, persisted, and removed.

### The Persistence Context

A persistence context is a set of managed entity instances that exist in a particular data store. The EntityManager interface defines the methods that are used to interact with the persistence context.

### The EntityManager Interface

The EntityManager API creates and removes persistent entity instances, finds entities by the entity’s primary key, and allows queries to be run on entities.

### Container-Managed Entity Managers

With a **container-managed entity manager**, an EntityManager instance’s persistence context is automatically propagated by the container to all application components that use the EntityManager instance within a single Java Transaction Architecture (JTA) transaction.

JTA transactions usually involve calls across application components. To complete a JTA transaction, these components usually need access to a single persistence context. This occurs when an EntityManager is injected into the application components by means of the javax.persistence.PersistenceContext annotation. The persistence context is automatically propagated with the current JTA transaction, and EntityManager references that are mapped to the same persistence unit provide access to the persistence context within that transaction. By automatically propagating the persistence context, application components don’t need to pass references to EntityManager instances to each other in order to make changes within a single transaction. The Java EE container manages the life cycle of container-managed entity managers.

To obtain an EntityManager instance, inject the entity manager into the application component:

@PersistenceContext

EntityManager em;

### Application-Managed Entity Managers

With **application-managed entity managers**, on the other hand, the persistence context is not propagated to application components, and the life cycle of EntityManager instances is managed by the application.

Application-managed entity managers are used when applications need to access a persistence context that is not propagated with the JTA transaction across EntityManager instances in a particular persistence unit. In this case, each EntityManager creates a new, isolated persistence context. The EntityManager, and its associated persistence context, is created and destroyed explicitly by the application.

Applications create EntityManager instances in this case by using the createEntityManager method of javax.persistence.EntityManagerFactory.

To obtain an EntityManager instance, you first must obtain an EntityManagerFactory instance by injecting it into the application component by means of the javax.persistence.PersistenceUnit annotation:

@PersistenceUnit

EntityManagerFactory emf;

Then, obtain an EntityManager from the EntityManagerFactory instance:

EntityManager em = emf.createEntityManager();

### Finding Entities Using the EntityManager

The EntityManager.find method is used to look up entities in the data store by the entity’s primary key.

@PersistenceContext

EntityManager em;

public void enterOrder(int custID, Order newOrder) {

Customer cust = em.find(Customer.class, custID);

cust.getOrders().add(newOrder);

newOrder.setCustomer(cust);

}

### Managing an Entity Instance’s Life Cycle

You manage entity instances by invoking operations on the entity by means of an EntityManager instance. Entity instances are in one of four states: new, managed, detached, or removed. New entity instances have no persistent identity and are not yet associated with a persistence context. Managed entity instances have a persistent identity and are associated with a persistence context.

Detached entity instances have a persistent identify and are not currently associated with a persistence context. Removed entity instances have a persistent identity, are associated with a persistent context, and are scheduled for removal from the data store.

### Persisting Entity Instances

New entity instances become managed and persistent either by invoking the persist method, or by a cascading persist operation invoked from related entities that have the cascade=PERSIST or cascade=ALL elements set in the relationship annotation. This means the entity’s data is stored to the database when the transaction associated with the persist operation is completed. If the entity is already managed, the persist operation is ignored, although the persist operation will cascade to related entities that have the cascade element set to PERSIST or ALL in the relationship annotation. If persist is called on a removed entity instance, it becomes managed. If the entity is detached, persist will throw an IllegalArgumentException, or the transaction commit will fail.

@PersistenceContext

EntityManager em;

...

public LineItem createLineItem(Order order, Product product,

int quantity) {

LineItem li = new LineItem(order, product, quantity);

order.getLineItems().add(li);

em.persist(li);

return li;

}

The persist operation is propagated to all entities related to the calling entity that have the cascade element set to ALL or PERSIST in the relationship annotation.

@OneToMany(cascade=ALL, mappedBy="order")

public Collection<LineItem> getLineItems() {

return lineItems;

}

### Removing Entity Instances

Managed entity instances are removed by invoking the remove method, or by a cascading remove operation invoked from related entities that have the cascade=REMOVE or cascade=ALL elements set in the relationship annotation. If the remove method is invoked on a new entity, the remove operation is ignored, although remove will cascade to related entities that have the cascade element set to REMOVE or ALL in the relationship annotation. If remove is invoked on a detached entity it will throw an IllegalArgumentException, or the transaction commit will fail. If remove is invoked on an already removed entity, it will be ignored. The entity’s data will be removed from the data store when the transaction is completed, or as a result of the flush operation.

public void removeOrder(Integer orderId) {

try {

Order order = em.find(Order.class, orderId);

em.remove(order);

}...

In this example, all LineItem entities associated with the order are also removed, as Order.getLineItems has cascade=ALL set in the relationship annotation.

### Synchronizing Entity Data to the Database

The state of persistent entities is synchronized to the database when the transaction with which the entity is associated commits. If a managed entity is in a bidirectional relationship with another managed entity, the data will be persisted based on the owning side of the relationship.

To force synchronization of the managed entity to the data store, invoke the flush method of the entity. If the entity is related to another entity, and the relationship annotation has the cascade element set to PERSIST or ALL, the related entity’s data will be synchronized with the data store when flush is called.

If the entity is removed, calling flush will remove the entity data from the data store.

### Creating Queries

The EntityManager.createQuery and EntityManager.createNamedQuery methods are used to query the datastore using Java Persistence query language queries.

The createQuery method is used to create **dynamic queries**, queries that are defined directly within an application’s business logic.

public List findWithName(String name) {

return em.createQuery(

"SELECT c FROM Customer c WHERE c.name LIKE :custName")

.setParameter("custName", name)

.setMaxResults(10)

.getResultList();

}

The createNamedQuery method is used to create **static queries**, queries that are defined in metadata using the javax.persistence.NamedQuery annotation. The name element of @NamedQuery specifies the name of the query that will be used with the createNamedQuery method. The query element of @NamedQuery is the query.

@NamedQuery(

name="findAllCustomersWithName",

query="SELECT c FROM Customer c WHERE c.name LIKE :custName"

)

Here’s an example of createNamedQuery, which uses the @NamedQuery defined above.

@PersistenceContext

public EntityManager em;

...

customers = em.createNamedQuery("findAllCustomersWithName")

.setParameter("custName", "Smith")

.getResultList();

### Named Parameters in Queries

Named parameters are parameters in a query that are prefixed with a colon (:). Named parameters in a query are bound to an argument by the javax.persistence.Query.setParameter(String name, Object value) method. In the following example, the name argument to the findWithName business method is bound to the :custName named parameter in the query by calling Query.setParameter.

public List findWithName(String name) {

return em.createQuery(

"SELECT c FROM Customer c WHERE c.name LIKE :custName")

.setParameter("custName", name)

.getResultList();

}

Named parameters are case-sensitive, and may be used by both dynamic and static queries.

### Positional Parameters in Queries

You may alternately use positional parameters in queries, instead of named parameters. Positional parameters are prefixed with a question mark (?) followed the numeric position of the parameter in the query. The Query.setParameter(integer position, Object value) method is used to set the parameter values.

In the following example, the findWithName business method is rewritten to use input parameters:

public List findWithName(String name) {

return em.createQuery(

“SELECT c FROM Customer c WHERE c.name LIKE ?1”)

.setParameter(1, name)

.getResultList();

}

Input parameters are numbered starting from 1. Input parameters are case-sensitive, and may be used by both dynamic and static queries.

### Persistence Units

A persistence unit defines a set of all entity classes that are managed by EntityManager instances in an application. This set of entity classes represents the data contained within a single data store.

Persistence units are defined by the persistence.xml configuration file. The JAR file or directory whose META-INF directory contains persistence.xml is called the root of the persistence unit. The scope of the persistence unit is determined by the persistence unit’s root.

Each persistence unit must be identified with a name that is unique to the persistence unit’s scope.

Persistent units can be packaged as part of a WAR or EJB JAR file, or can be packaged as a JAR file that can then be included in an WAR or EAR file.

If you package the persistent unit as a set of classes in an EJB JAR file, persistence.xml should be put in the EJB JAR’s META-INF directory.

If you package the persistence unit as a set of classes in a WAR file, persistence.xml should be located in the WAR file’s WEB-INF/classes/META-INF directory.

If you package the persistence unit in a JAR file that will be included in a WAR or EAR file, the JAR file should be located:

* In the WEB-INF/lib directory of a WAR.
* In the top-level of an EAR file.
* In the EAR file’s library directory.

**The persistence.xml File**

persistence.xml defines one or more persistence units. The following is an example persistence.xml file.

<persistence>

<persistence-unit name="OrderManagement">

<description>This unit manages orders and customers.

It does not rely on any vendor-specific features and can

therefore be deployed to any persistence provider.

</description>

<jta-data-source>jdbc/MyOrderDB</jta-data-source>

<jar-file>MyOrderApp.jar</jar-file>

<class>com.widgets.Order</class>

<class>com.widgets.Customer</class>

</persistence-unit>

</persistence>

This file defines a persistence unit named OrderManagement, which uses a JTA-aware data source jdbc/MyOrderDB. The jar-file and class elements specify managed persistence classes: entity classes, embeddable classes, and mapped superclasses. The jar-file element specifies JAR files that are visible to the packaged persistence unit that contain managed persistence classes, while the class element explicitly names managed persistence classes.

The jta-data-source (for JTA-aware data sources) and non-jta-data-source (non-JTA-aware data sources) elements specify the global JNDI name of the data source to be used by the container.