Module 4 – JPA with Hibernate 3.0

Module Overview

Hibernate - This is an ORM (Object Relation Mapping) framework for data layer of software application. ORM framework helps in converting data into POJO (plain java object) and also provide other capabilities like default SQL operations like insert, delete, read and update, user can also create custom queries, caching of data, etc.

Module Objective

At the end of this module, students should be able to demonstrate appropriate knowledge, and show an understanding of the following:

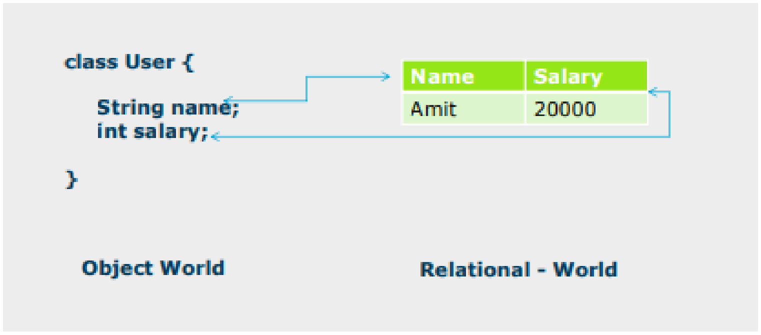
- Understanding Object Relation Mapping
- Learning JPA API
- Associations and Mapping

Introduction to ORM

What is Object Persistence?

Persistence means to make application's data to outlive the applications process.

In Java terms, the objects to live beyond the scope of the JVM so that the same state is available later.



The above diagram depicts mapping of object state into database table columns. To do so, traditionally, we rely on JDBC API, which allows developers to save application data into database, however conversion is required from object format to database table format which un-necessarily increases line of code.

However, there are lot of challenges and mismatch in data processing in these two models. In addition, if database changes, then developer need to make modification in the configuration which is database specific.

So to shorten the development time, and to save application object directly into database, there was need to reinvent the approach of mapping object and relational model.

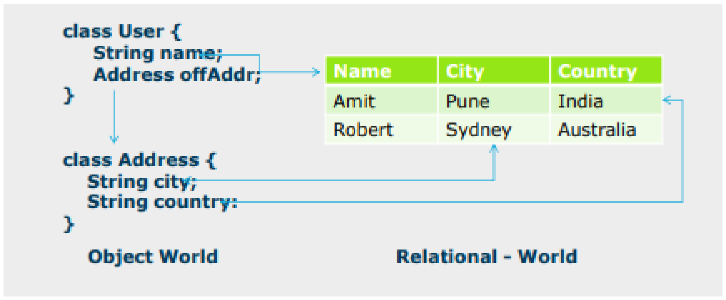
Object-Relation Impedance Mismatch

'Object-relational Impedance Mismatch' means that object models and relational models do not work very well together.

RDBMSs represent data in a tabular format, whereas object-oriented languages, such as Java, represent it as an interconnected graph of objects.

Loading and storing graphs of objects using a tabular relational database exposes us to following mismatch problems...

- Granularity
- Inheritance (subtypes)
- Identity


```
class User {
    String name;
    Address offAddr;
}

class Address {
    String city;
    String country;
}
```

Name	City	Country
Amit	Pune	India
Robert	Sydney	Australia

Conversion of the java datatypes to underlying database types is done by ORM automatically.

Introduction ORM

Storing object-oriented entities in a relational database is often not a simple task and requires a great deal of repetitive code along with conversion between data types.

Object-relational mapper, or O/RM, were created to solve this problem. An O/RM persists entities in and retrieves entities from relational databases without the programmer having to write SQL statements and translate entity properties to statement parameters and result set columns to entity properties.

It consists of:

- An API, to perform CRUD operations on objects of persistent classes
- A language to specify queries that refer to classes and properties of classes
- A facility, to specify mapping metadata
- A technique, for the ORM implementation to interact with transactional objects to perform dirty checking. Lazy association, fetching, and other optimization functions.

Dirty Checking:

A dirty checking feature avoids unnecessary database write actions by performing SQL updates only on the modified fields of persistent objects. For example, if you modify salary of employee on object model, only salary field will be updated instead of updating entire employee object.

Lazy association fetching:

Lazy fetching decides whether to load child objects while loading the Parent Object. For example, Consider department entity consist of many employees, and someone query to fetch department details, ORM fetches only department details and defers loading employees. This will be done, when one request details of employees working in that department.

Why ORM?

It “shields” developers from “messy” SQL.

ORM tools allows developers to focus on the business logic of the application rather than repetitive CRUD (Create Read Update Delete) logic.

Some of the benefits of ORM are:

▪ Productivity

▪ Maintainability

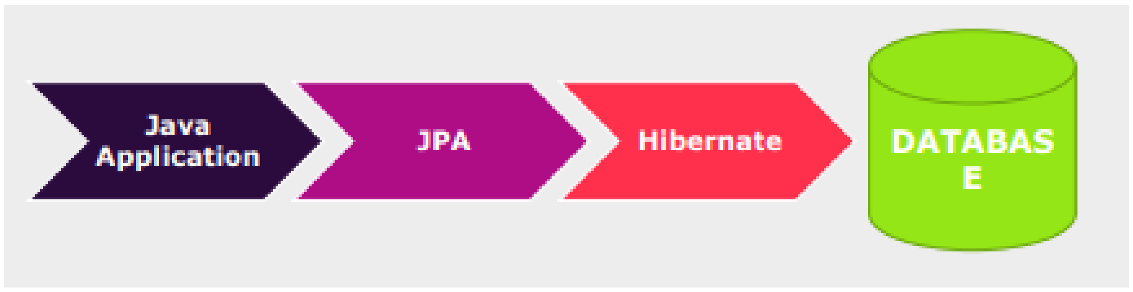
▪ Performance

▪ Vendor independence

Introduction to Java Persistence API

JPA is just an specification from Sun, which is released under JEE 5specification. JPA standardized the ORM persistence technology for Java developers. JPA is not a product and can't be used as it is for persistence. It needs an ORM implementation to work and persist the Java Objects. ORM frameworks that can be used with JPA are Hibernate, Top link, Open JPA etc.

The Java Persistence API (JPA) is one approach to ORM. Via JPA the developer can map, store, update and retrieve data from relational databases to Java Objects and vice versa, JPA permits the developer to work directly with objects rather than with SQL statements. JPA is a specification and several implementations are available.



JPA is not the first attempt to create an ORM solution in Java. Before JPA, there were Java Data Objects (JDO) and Enterprise JavaBeans (EJB). JDO used to be popular, but seems to have run out of steam.

EJB, up to version 2.1, was overly complex and hard to use, harder than losing weight. EJB 3.0 simplifies things a lot and even uses JPA as its persistence mechanism. In short, JPA has started as part of EJB 3.0. However, since people want to use JPA without an EJB container, JPA has become an independent specification.

JPA is merely a specification, i.e. a document. In order for it to be useful, it needs a reference implementation, which is a Java API that implements the specification. There are numerous software packages that are JPA reference implementations. Hibernate, EclipseLink, and Apache OpenJPA are some of them.

<u>Below listed are few advantages of JPA:</u>

1. You don't need to create tables. In some cases, you don't even need to create a database. If any of your entity classes changes, the modern JPA provider can be configured to adapt the tables.

2. You don't need to write SQL statements, even though sometimes you may have to work with JPQL, the Java Persistence Query Language.

3. Changing databases, say from Oracle to MySQL, is a breeze.

There are disadvantages too, but most of them are negligible:

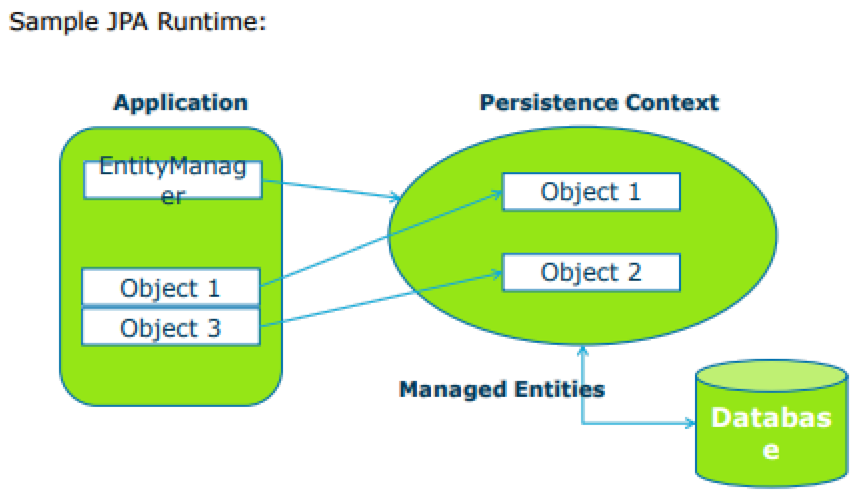
1. JPA adds to the application's memory usage. Negligible in most cases.

2. JPA adds an extra layer to the application, making the system a bit slower than if it accesses the database through JDBC directly. However, the performance penalty is small that it is considered negligible.

The Persistence Life Cycle

Before we start working with ORM, it is very important to understand how ORM works. The slide shows an example of abstraction, when we dial or receive a call on mobile, lot of functionality goes in background. We as a user, least bothered about internal component working due to abstraction.

Similarly, objects created in your application, when passed to ORM, get stored in database table. How it happens? What work goes in background? How your object persisted in database?



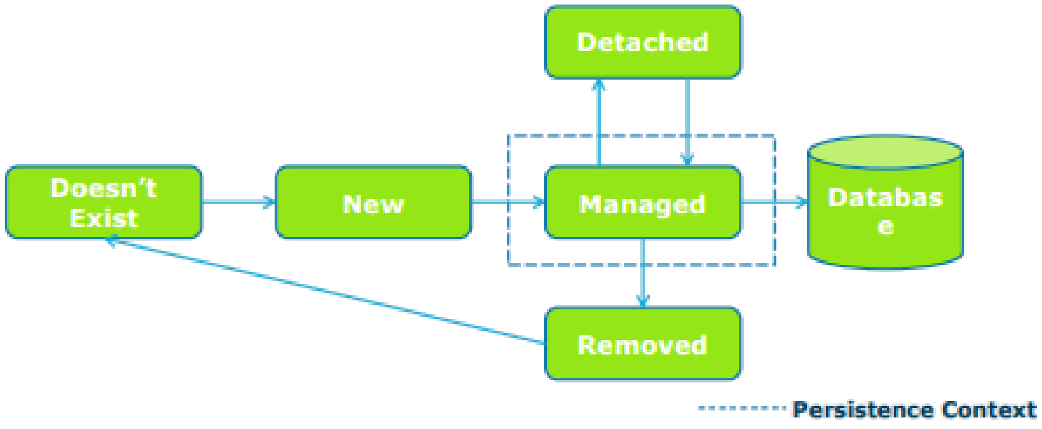
The slide diagram shows a sample JPA runtime

Entity Manager: The EntityManager is the primary interface used by application developers to interact with the JPA runtime.

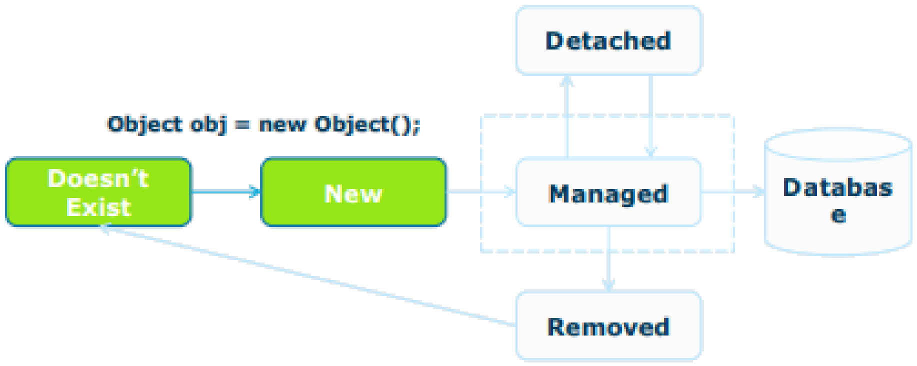
Persistence Context: Persistence context defines a scope under which particular entity instances are created, persisted, and removed.

Every EntityManager manages its own persistence context. In short, persistence context is a memory area for Entity Manager to work on entity instance.

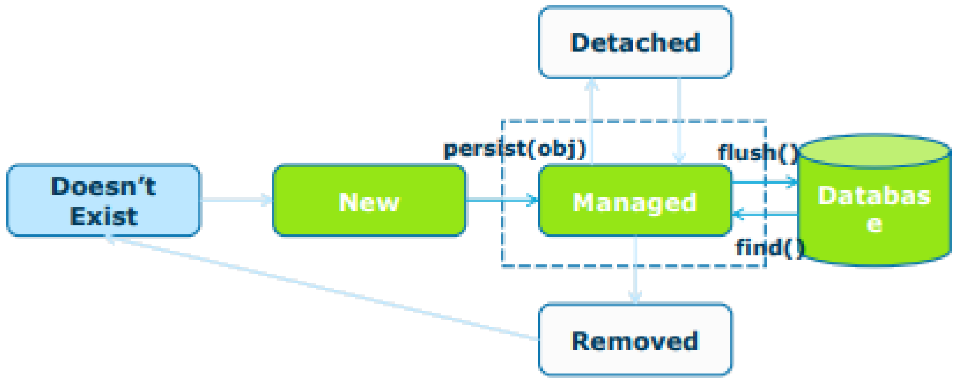
JPA uses EntityManager instance to manage objects which required to be persisted. Such objects are called Entities. Entities managed by EntityManager travels through different life cycle phases. This lesson primarily focuses on entities persistence life cycle.



Object/Entity managed by ORM (using EntityManager) passes through different stages during its persistence.



New State: When an entity object is initially created its state is New. In this state the object is not yet associated with an Entity Manager and has no representation in the database.

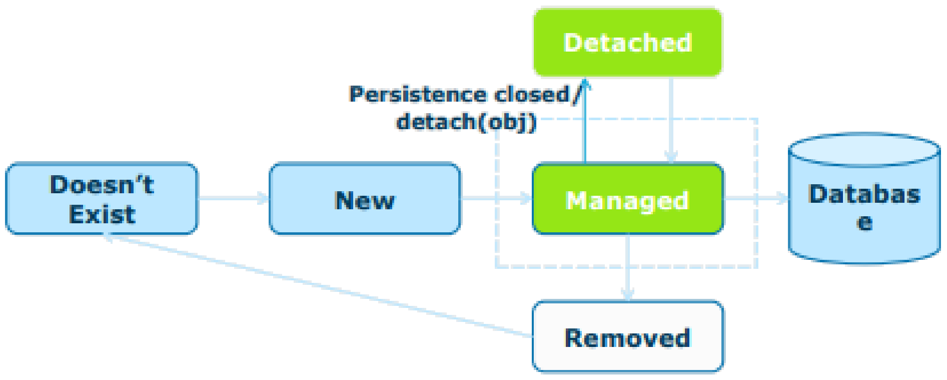


Managed State: An entity object becomes Managed when it is persisted to the database via an EntityManager's persist method which must be invoked within an active transaction. On transaction commit, the owning Entity Manager stores the new entity object to the database.

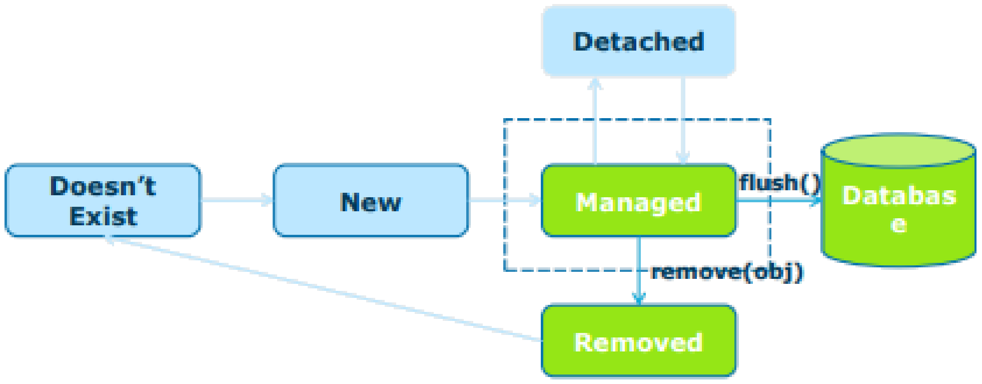
Entity objects retrieved from the database by an EntityManager are also in the Managed state.

If a managed entity object is modified within an active transaction the change is detected by the owning EntityManager and the update is propagated to the database on transaction commit.

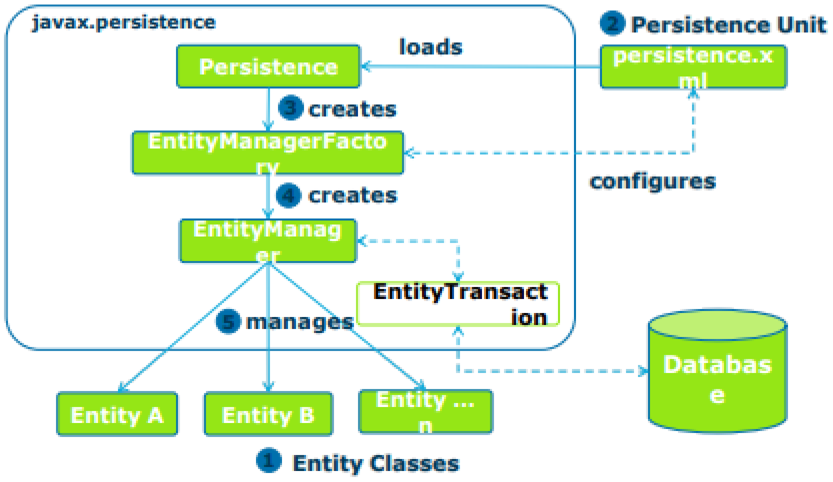
Detached State: represents entity objects that have been disconnected from the EntityManager. For instance, all the managed objects of an EntityManager become detached when the EntityManager is closed.



Removed State : A managed entity object can also be retrieved from the database and marked for deletion, by using the EntityManager's remove method within an active transaction. The entity object changes its state from Managed to Removed, and is physically deleted from the database during commit.



JPA with Hibernate 3.0



Working with JPA

1. You normally start with a persistence strategy by identifying which classes need to be made entities.
2. Next step is to create configuration file (an XML document named persistence.xml) that contains the details about the relational database.
3. EntityManagerFactory is a factory based class responsible for creating EntityManager instance. It is obtained using Persistence class's createEntityManagerFactory static method.
4. EntityManagerFactory class designed to create EntityManager.
5. Once you have an EntityManager, you can start managing your entities. You can persist an entity, find one that matches a set of criteria, and so on. Each work of EntityManager with entities must be governed under EntityTransaction. Let us discuss the each step in detail.

Requirements for Entity Classes:

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.

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 packageprivate and can be accessed directly only by the entity class’s methods.

Clients must access the entity’s state through accessor or business methods.

Entity Annotations:

The @Entity annotation marks this class as an entity bean, so it must have a no-argument constructor that is visible with at least protected scope.

Each entity bean has to have a primary key, which you annotate on the class with the @Id annotation.

In some situation, few properties of an entity, do not need to be stored in the database. In this case, ORM do not take this property for all the Database operation. This can be done using @Transient annotation.

By default, the @Id annotation will automatically determine the most appropriate primary key generation strategy to use—you can override this by also applying the @GeneratedValue annotation. This takes a pair of attributes: strategy and generator.

The strategy attribute must be a value from the GeneratorType enumeration, which defines four types of strategy constants.

1. AUTO: (Default) JPA decides which generator type to use, based on the database’s support for primary key generation.

2. IDENTITY: The database is responsible for determining and assigning the next primary key.

3. SEQUENCE: Some databases support a SEQUENCE column type.

4. TABLE: This type keeps a separate table with the primary key values.

```
<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="unit-name ">
<provider> <!-- JPA provider name like hibernate-->
</provider>
<properties> <!-- Database properties --></properties>
</persistence-unit>
</persistence>
```

To connect with database, you need to set various properties regarding driver class, user name and password. This configuration is done with an XML file named persistence.xml.

Elements in persistence.xml:

The <persistence> is the root element of persistence.xml file. A persistence unit defines all the entity classes that need to be managed and the JDBC details to connect to an underlying relational database.

1. <persistence-unit> : It has the name attribute specifies a name that can be referenced from your Java code. The transaction-type attribute informs ORM about transaction management. It may take values like:

   a. RESOURCE_LOCAL: Application will handle transaction management. i.e. creating, starting and closing of transactions.

   b. JTA: JEE server Container will take care for transaction management.

2. <provider>: Specifies the fully-qualified name of the JMS provider class. E.g. hibernate.

3. <property>: Minimum four properties must be nested using <property> element .

needed. These properties specify the JDBC URL, JDBC username, JDBC password, and driver.

4. <class> : Each class element specifies a fully-qualified name of an entity class. This approach is used to inform which classes needs to be managed by JPA. i.e. Entity classes. There may be more than one class elements.

An EntityManager is responsible for managing entities. It is one of the most important types in the API.

You can get an EntityManagerFactory easily by using the Persistence class's createEntityManagerFactory() static method. It accept string parameter which is name of persistence unit defined in persistence.xml file.

Using the EntityManagerFactory class factory, you can create EntityManager instances using createEntityManager() method.

Working with Entity Manager

The EntityManager interface defines the methods that are used to interact with the persistence context. The EntityManager API is used to create and remove persistent entity instances, to find persistent entities by primary key, and to query over persistent entities.

EntityManager important methods:

1. persist(object): Persists the entity object

2. find(class,primarykey): Retrieves a specific entity object

3. remove(object): Removes an entity object

4. refresh: Refreshes the entity instances in the persistence context from the database

5. contains: Returns true if the entity instance is in the persistence context. This signifies that the entity instance is managed

6. flush: forces the synchronization of the database with entities in the persistence context

7. clear: Clears the entities from the persistence context

8. evict(object): Detaches an entity from the persistence context

9. close(): Flush entity instances first, clears persistence context and nullify the entity manager

Transactions

A transaction is a set of operations that either fail or succeed as a unit. Transactions are a fundamental part of persistence.

A database transaction consists of a set of DML (Data Manipulation Language) operations that are committed or rolled back as a single unit.

An object level transaction is one in which a set of changes made to a set of objects are committed to the database as a single unit.

Transactions can be controller in two ways in JPA

• Java Transaction API (JTA)

- container-managed entity manager

• EntityTransaction API (tx.begin(), tx.commit(), etc)

- application-managed entity manager

Application Managed Entity Manager

```
public class PersistenceProgram {

public static void main(String[] args)

{

EntityManagerFactory emf =

Persistence.createEntityManagerFactory("SomePUnit");

EntityManager em = emf.createEntityManager();

em.getTransaction().begin();    // Perform finds, execute queries,

// update entities, etc.

em.getTransaction().commit();

em.close();

emf.close();

} }
```

JPA Queries

Java Persistence Query Language (JPQL)

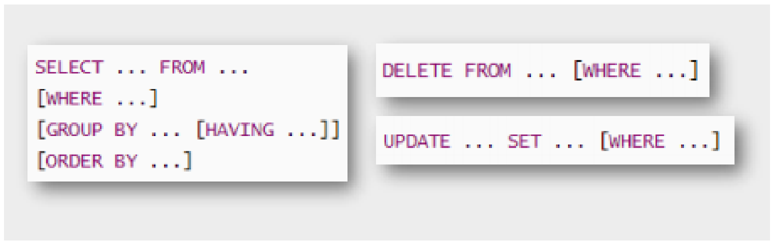
The Java Persistence Query Language (JPQL) is a platform-independent object-oriented query language defined as part of the Java Persistence API (JPA) specification.

JPQL is used to make queries against entities stored in a relational database. The JPQL defines queries for entities and their persistent state. The query language allows you to write portable queries that work regardless of the underlying data store.

The JPQL can be considered as an object oriented version of SQL. Users familiar with SQL should find JPQL very easy to learn and use.

The main difference between SQL and JPQL is that SQL works with relational database tables, records and fields, whereas JPQL works with Java classes and objects.

Similarities between SQL and JPQL



```
SELECT ... FROM ...
[WHERE ...]
[GROUP BY ... [HAVING ...]]
[ORDER BY ...]
```

```
DELETE FROM ... [WHERE ...]
```

```
UPDATE ... SET ... [WHERE ...]
```

As shown above, there is no difference between SQL and JPQL query syntax. Consider the following Entity class,

```
@Entity

public class Book implements Serializable {

@Id

private Long id;

private String bookTitle;

private String author;

private Double price; // getter and setter
methods

}
```

If you want to find all books written by author ‘Jim Kathy’, then you need to write JPQL select statement on above entity class as given below:

```
SELECT b.id,b. bookTitle,b.price --property reference

FROM Book b --object reference

WHERE b.author = ‘Jim Kathy’;
```

Whereas the below query counts total books object available in data store.

```
SELECT COUNT(b.id)

FROM Book b;
```

Queries are represented in JPA 2 by two interfaces - the old Query interface, which was the only interface available for representing queries in JPA 1, and the new TypedQuery<T> JPA interface that was introduced in JPA 2.

The TypedQuery interface extends the Query interface.

It is easier to run queries and process the query results in a type safe manner when using the TypedQuery interface.

The Query/TypedQuery<T> interface defines two methods for running SELECT queries:

1. getSingleResult() - for use when exactly one result object is expected.

2. getResultList() - for general use in any other case.

For UPDATE and DELETE use executeUpdate() method.

Query interface should be used mainly when the query result type is unknown or when a query returns polymorphic results and the lowest known common denominator of all the result objects is Object.

When a more specific result type is expected queries should usually use the TypedQuery.

Query parameters enable the definition of reusable queries. Such queries can be executed with different parameter values to retrieve different results.

There are multiple ways to pass parameters to query.

1. Named Parameters (:name)

2. Ordinal Parameters (?index)

3. Criteria Query Parameters

The slide example shows example of named parameter “ptitle”, which is later injected using setParameter() method on query.

Named Queries

Named queries enables developer to write static queries. Such queries are written on an entity class with class level annotations called @NamedQueries and @NamedQuery.

@NamedQueries accepts array of @NamedQuery, whereas @NamedQuery requires name and query.

Association and Mapping

What is Entity Association?

Association represents relationship between entities. A Java class can contain an object of another class or a set of objects of another class.

There is no directionality involved in relational world, its just a matter of writing a query. But there is notion of directionality which is possible in java.

Hence associations are classified as

▪ Unidirectional

▪ Bidirectional.

Unidirectional one to one:

@OneToOne

Defines a single-valued association to another entity that has one-to-one multiplicity. This annotations can have following optional attributes:

1. cascade (Optional): The operations that must be cascaded to the target of the association. i.e. It indicates JPA operations on associated entity along with owner of association.

2. fetch (Optional) : Whether the association should be lazily loaded or must be eagerly fetched. i.e. When you fetch Student entity, if you want to load the associated entity (Address) immediately, then you have to mention this attribute with 'EAGER'. Default is LAZY, means the associated entity (Address) will be loaded when required.

<u>Cascading associated Entities</u>

Cascade attribute is mandatory, whenever we apply relationship between objects, cascade attribute transfers operations done on one object onto its related child objects.

Cascade Types:

ALL: All cascade operations will be applied to the parent entity's related entity. All is equivalent to specifying cascade={DETACH, MERGE, PERSIST, REFRESH, REMOVE}

DETACH: If the parent entity is detached from the persistence context, the related entity will also be detached.

MERGE: If the parent entity is merged into the persistence context, the related entity will also be merged.

PERSIST: If the parent entity is persisted into the persistence context, the related entity will also be persisted.

REFRESH: If the parent entity is refreshed in the current persistence context, the related entity will also be refreshed.

REMOVE: If the parent entity is removed from the current persistence context, the related entity will also be removed.

Bidirectional one to one:

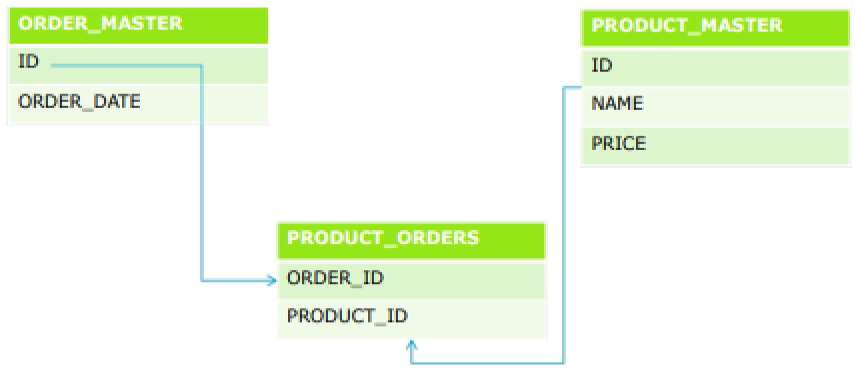
The direction of a relationship can be either bidirectional or unidirectional. A bidirectional relationship has both an owning side and an inverse side. A unidirectional relationship has only an owning side.

The owning side of a relationship determines how the Persistence runtime makes updates to the relationship in the database.

In a bidirectional relationship, each entity has a relationship field or property that refers to the other entity. Through the relationship field or property, an entity class's code can access its related object. If an entity has a related field, the entity is said to "know" about its related object. Such relationship field must be marked with mappedBy attribute.

The inverse side of a bidirectional relationship must refer to its owning side by using the mappedBy element of the @OneToOne, @OneToMany, or @ManyToMany annotation.

Bidirectional Many to many using Join Table:



To store data of many to many relationship, join table can be used. As shown above, the orders stored in ORDER_MASTER table, products stored in PRODUCT_MASTER and there association is stored in PRODUCT_ORDERS.

@JoinTable: This annotation is used to describe join table properties. It has three attributes:

1. name: Name of the join table

2. joinColumns: Join column name for owning side i.e. order table

3. inverseColumns: Join column name for inverse side. i.e. product table.

Mapping Inheritance

Three ways of handling inheritance

1. Single table per class hierarchy (InheritanceType.SINGLE_TABLE)

2. Table per concrete entity class (InheritanceType.TABLE_PER_CLASS)

3. “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 (InheritanceType.JOINED)

Let us explore each in detail.

1. Single Table per Class Hierarchy:

   In this strategy, only one database table is created for all subclasses. It is denormalized table has columns for all attributes.

   JPA Mapping Configuration:

   Single annotation @Inheritance with InheritanceType strategy required only on superclass. Also use ‘@DiscriminatorColumn’ to define discriminator column and it data type, which later will be used to differentiate parent and child rows.

<u>Advantages</u>

1. It is the fastest of all inheritance models

2. Since it does not requires a join to retrieve a persistent instance from the database.

3. Persisting or Updating a persistent instance requires only a single INSERT or UPDATE statement.

<u>Disadvantages</u>

1. The larger the inheritance model gets, the "wider" the mapped table gets, in that for every field in the entire inheritance hierarchy, a column must exist in the mapped table. This may have undesirable consequence on the database size, since a wide or deep inheritance hierarchy will result in tables with many mostly-empty columns.

2. Table per concrete class:

   In this inheritance strategy, one database table will be created for the superclass AND one per subclass. Subclass tables have their object-specific columns along with shared columns from superclass table.

<u>Advantages:</u>

1. This is the easiest method of Inheritance mapping to implement.

<u>Disadvantages:</u>

1. Data that belongs to a parent class is scattered across a number of subclass tables, which represents concrete classes.

2. This hierarchy is not recommended for most cases.

3. Changes to a parent class is reflected to large number of tables

4. A query couched in terms of parent class is likely to cause a large number of select operations

3. Joined Subclass Hierarchy:

   In this inheritance strategy, one database table will be created for the superclass AND one per subclass. Subclass tables have their object-specific columns along with a foreign key column referring primary key of Superclass.

Advantages

1. Using joined subclass tables results in the most normalized database schema, meaning the schema with the least spurious or redundant data.

Disadvantages

1. Retrieving any subclass requires one or more database joins, and storing subclasses requires multiple INSERT or UPDATE statements.