JPA with Hibernate 3.0

JPA Queries

After completing this lesson, participants will be able to understand:

- JPQL
- JPA Query API
- Working with JPA Queries

Why JPQL?

Entity Manager's find() method can be used to locate single entity only based on primary key value.

What if you want to load data based on complex criteria?

For example:

- Load employees residing in "Pune"
- Find orders placed between given two dates.
- Load and combine two entities data for reporting.

JAVA PERSISTENCE QUERY LANGUAGE!

What is JPQL?

JPQL is used to make queries against entities stored in a relational database.

JPQL is object-oriented as it automatically populates object with data received from database

Benefits of using JPQL:

- JPQL is portable across implementer and database
- Uses SQL-like syntax to query entities, requires less effort to learn
- No manual conversion of row data into object and vice-versa

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.

JPQL Query Syntax

JPQL query syntax is similar to SQL syntax.

Difference between SQL and JPQL

- JPQL works on Java Classes, Objects and Properties
- SQL works on Database tables, rows and columns

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

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

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

As shown in the slide, 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';
```

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

```
SELECT COUNT(b.id)
FROM Book b;
```

JPA Query API

JPA provides two interfaces to query database:

- Query
- TypedQuery<T>

JPA 2 introduced TypedQuery interface which extends Query to query results in a type safe manner.

Below table list important methods of these interfaces:

Method	Purpose
getSingleResult()	Executes SELECT statement and returns a single object result
getResultList()	Executes SELECT statement and returns the query result as an list of objects.
executeUpdate()	Execute an UPDATE or DELETE statement.

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.

Note: The Query interface should be used mainly when the query result type is unknown or when a query returns polymorphic results. When a more specific result type is expected, use the TypedQuery interface.

Working with JPA Queries

Query work starts with an EntityManager, which serves as a factory for both Query and TypedQuery.

The EntityManager interface has an overloaded createQuery() method with the following signature:

- Query createQuery(java.lang.String query)
- TypedQuery<T> createQuery(java.lang.String query, T resultClass)

```
Query query = entityManager.createQuery("Select s FROM Student
s");
```

```
TypedQuery<Student> query = entityManager.createQuery
                    ("SELECT student from Student student",
Student.class);
```

Query Examples

The following queries to find single book with id 'B13455'

```
String qStr = "SELECT book FROM Book book WHERE
book.id='B13455' ";
TypedQuery<Book> query = entityManager.createQuery(qStr ,
Book.class);
Book book = query.getSingleResult();
```

To list all books from store:

```
String qStr = "SELECT book FROM Book book ";
TypedQuery<Book> query = entityManager.createQuery(qStr,
Book.class);
List<book> bookList = query.getResultList();
```

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.

Note: For type safety and to avoid runtime cast exceptions, it is always good practice to work with TypedQuery<T>.

Dynamic queries with parameters

JPA allows developer to create dynamic queries which are more efficient and can be built dynamically at runtime.

The following code retrieves a book object from the database by its title using named parameter "ptitle":

```
String qStr = "SELECT book FROM Book book WHERE
book.title=:ptitle";
TypedQuery<Book> query =
entityManager.createQuery(qStr,Book.class);
query.setParameter("ptitle", "Introduction to JPA");
Book book = query.getSingleResult();
```

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)

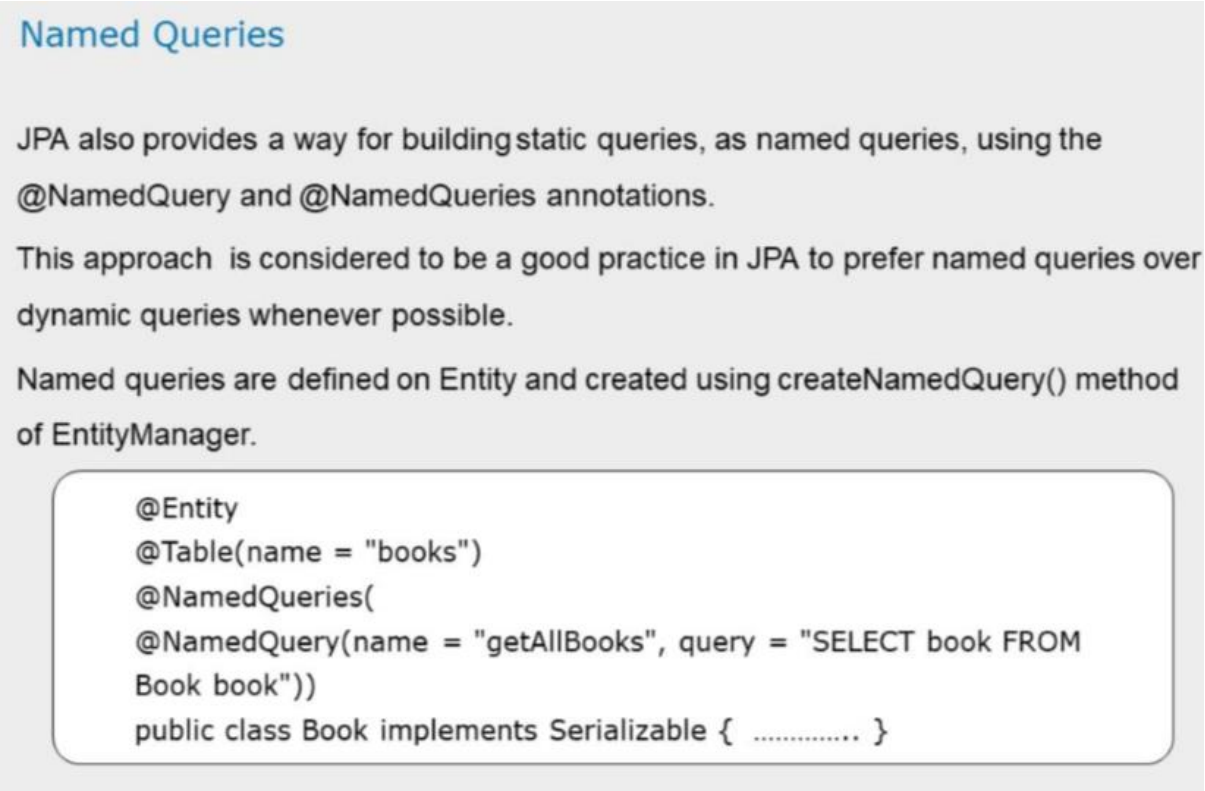
3. Criteria Query Parameters

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

Note: Named parameters should always preceded by colon (:).

While setting the date related parameters, one should make use of TemporalType to indicate a specific mapping of java.util.Date or java.util.Calendar. For example

em.createQuery("SELECT s FROM Student s " + "WHERE s.startDate BETWEEN :start AND :end") .setParameter("start", start, TemporalType.DATE) .setParameter("end", end, TemporalType.DATE) .getResultList();



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.

Once defined later on we can create and execute named query as shown below:

Query query = entityManager.createNamedQuery("getAllBooks");

List<Book> bookList= query.getResultList();