JPA

Queries

JPA Query API

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` interface that was introduced in JPA 2. The `TypedQuery` interface extends the `Query` interface.

In JPA 2 the `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` interface. It is easier to run queries and process the query results in a type safe manner when using the TypedQuery interface.

JPA Query API

Building Queries with createQuery

As with most other operations in JPA, using queries starts with an `EntityManager` (represented by `em` in the following code snippets), which serves as a factory for both `Query` and `TypedQuery`:

```
  Query q1 = em.createQuery("SELECT c FROM Country c");

  TypedQuery<Country> q2 =
      em.createQuery("SELECT c FROM Country c", Country.class);
```

In the above code, the same JPQL query which retrieves all the Country objects in the database is represented by both *q1* and *q2*. When building a `TypedQuery` instance the expected result type has to be passed as an additional argument, as demonstrated for *q2*. Because, in this case, the result type is known (the query returns only Country objects), a TypedQuery is preferred.

JPA Query API

Building queries by passing JPQL query strings directly to the `createQuery` method, as shown above, is referred to in JPA as dynamic query construction because the query string can be built dynamically at runtime.

The JPA Criteria API provides an alternative way for building dynamic queries, based on Java objects that represent query elements (replacing string based JPQL).

JPA also provides a way for building static queries, as named queries, using the `@NamedQuery` and `@NamedQueries` annotations. It is considered to be a good practice in JPA to prefer named queries over dynamic queries when possible.

Running JPA Queries

The `Query` interface defines two methods for running SELECT queries:

- `Query.getSingleResult` - for use when exactly one result object is expected.
- `Query.getResultList` - for general use in any other case.

Similarly, the `TypedQuery` interface defines the following methods:

- `TypedQuery.getSingleResult` - for use when exactly one result object is expected.
- `TypedQuery.getResultList` - for general use in any other case.

In addition, the Query interface defines a method for running DELETE and UPDATE queries:

- `Query.executeUpdate` - for running only DELETE and UPDATE queries.

Running JPA Queries

Ordinary Query Execution (with getResultList)

The following query retrieves all the Country objects in the database. Because multiple result objects are expected, the query should be run using the getResultList method:

```
TypedQuery<Country> query =
    em.createQuery("SELECT c FROM Country c", Country.class);
List<Country> results = query.getResultList();
```

Both Query and TypedQuery define a getResultList method, but the version of Query returns a result list of a raw type (non generic) instead of a parameterized (generic) type:

```
Query query = em.createQuery("SELECT c FROM Country c");
List results = query.getResultList();
```

Running JPA Queries

An attempt to cast the above `results` to a parameterized type (`List<Country>`) will cause a compilation warning. If, however, the new `TypedQuery` interface is used casting is unnecessary and the warning is avoided.
The query result collection functions as any other ordinary Java collection. A result collection of a parameterized type can be iterated easily using an enhanced for loop:

```
for (Country c : results) {
    System.out.println(c.getName());
}
```

Note that for merely printing the country names, a query that uses projection and retrieves country names directly instead of fully built Country instances would be more efficient.

Running JPA Queries

Single Result Query Execution (with getSingleResult)

The `getResultList` method (which was discussed above) can also be used to run queries that return a single result object. In this case, the result object has to be extracted from the result collection after query execution (e.g. by `results.get(0)`). To eliminate this routine operation JPA provides an additional method, `getSingleResult`, as a more convenient method when exactly one result object is expected.
The following aggregate query always returns a single result object, which is a Long object reflecting the number of Country objects in the database:

```
TypedQuery<Long> query = em.createQuery(
    "SELECT COUNT(c) FROM Country c", Long.class);
long countryCount = query.getSingleResult();
```

Notice that when a query returns a single object it might be tempting to prefer Query over TypedQuery even when the result type is known because the casting of a single object is easy and the code is simple:

```
Query query = em.createQuery("SELECT COUNT(c) FROM Country c");
long countryCount = (Long)query.getSingleResult();
```

Running JPA Queries

An aggregate COUNT query always returns one result, by definition. In other cases our expectation for a single object result might fail, depending on the database content. For example, the following query is expected to return a single Country object:

```
Query query = em.createQuery(
    "SELECT c FROM Country c WHERE c.name = 'Canada'");
Country c = (Country)query.getSingleResult();
```

However, the correctness of this assumption depends on the content of the database. If the database contains multiple Country objects with the name 'Canada' (e.g. due to a bug) a NonUniqueResultException is thrown. On the other hand, if there are no results at all a NoResultException is thrown. Therefore, using getSingleResult requires some caution and if there is any chance that these exceptions might be thrown they have to be caught and handled.

Running JPA Queries

DELETE and UPDATE Query Execution (with executeUpdate)

DELETE and UPDATE queries are executed using the `executeUpdate` method.
For example, the following query deletes all the Country instances:

```
int count = em.createQuery("DELETE FROM Country").executeUpdate();
```

and the following query resets the area field in all the Country instances to zero:

```
int count = em.createQuery("UPDATE Country SET area = 0").executeUpdate();
```

A `TransactionRequiredException` is thrown if no transaction is active.
On success - the executeUpdate method returns the number of objects that have been updated or deleted by the query.

Query Parameters in JPA

Query parameters enable the definition of reusable queries. Such queries can be executed with different parameter values to retrieve different results. Running the same query multiple times with different parameter values (arguments) is more efficient than using a new query string for every query execution, because it eliminates the need for repeated query compilations.

Named Parameters (:name)

The following method retrieves a Country object from the database by its name:

```
  public Country getCountryByName(EntityManager em, String name) {
    TypedQuery<Country> query = em.createQuery(
        "SELECT c FROM Country c WHERE c.name = :name", Country.class);
    return query.setParameter("name", name).getSingleResult();
  }
```

The WHERE clause reduces the query results to Country objects whose name field value is equal to `:name`, which is a parameter that serves as a placeholder for a real value. Before the query can be executed a parameter value has to be set using the `setParameter` method. The `setParameter` method supports method chaining (by returning the same `TypedQuery` instance on which it was invoked), so invocation of `getSingleResult` can be chained to the same expression.

Query Parameters in JPA

Named parameters can be easily identified in a query string by their special form, which is a colon (`:`) followed by a valid JPQL identifier that serves as the parameter name. JPA does not provide an API for defining the parameters explicitly (except when using criteria API), so query parameters are defined implicitly by appearing in the query string. The parameter type is inferred by the context. In the above example, a comparison of `:name` to a field whose type is `String` indicates that the type of `:name` itself is `String`.

Queries can include multiple parameters and each parameter can have one or more occurrences in the query string. A query can be run only after setting values for all its parameters (in no matter in which order).

Query Parameters in JPA

Ordinal Parameters (?index)

In addition to named parameter, whose form is `:name`, JPQL supports also ordinal parameter, whose form is `?index`. The following method is equivalent to the method above, except that an ordinal parameter replaces the named parameter:

```
public Country getCountryByName(EntityManager em, String name) {
  TypedQuery<Country> query = em.createQuery(
      "SELECT c FROM Country c WHERE c.name = ?1", Country.class);
  return query.setParameter(1, name).getSingleResult();
}
```

The form of ordinal parameters is a question mark (?) followed by a positive int number. Besides the notation difference, named parameters and ordinal parameters are identical. Named parameters can provide added value to the clarity of the query string (assuming that meaningful names are selected). Therefore, they are preferred over ordinal parameters.

Criteria Query Parameters

In a JPA query that is built by using the JPA Criteria API - parameters (as other query elements) are represented by objects (of type `ParameterExpression` or its super interface `Parameter`) rather than by names or numbers.

Query Parameters in JPA

Parameters vs. Literals

Following is a third version of the same method. This time without parameters:

```
public Country getCountryByName(EntityManager em, String name) {
  TypedQuery<Country> query = em.createQuery(
      "SELECT c FROM Country c WHERE c.name = '" + name + "'",
      Country.class);
  return query.getSingleResult();
}
```

Instead of using a parameter for the queried name the new method embeds the name as a `String` literal. There are a few drawbacks to using literals rather than parameters in queries.
First, the query is not reusable. Different literal values lead to different query strings and each query string requires its own query compilation, which is very inefficient.
Second, embedding strings in queries is unsafe and can expose the application to JPQL injection attacks. Suppose that the name parameter is received as an input from the user and then embedded in the query string as is. Instead of a simple country name, a malicious user may provide JPQL expressions that change the query and may help in hacking the system.
In addition, parameters are more flexible and support elements that are unavailable as literals, such as entity objects.

Query Parameters in JPA

API Parameter Methods

Over half of the methods in `Query` and `TypedQuery` deal with parameter handling. The Query interface defines 18 such methods, 9 of which are overridden in TypedQuery. That large number of methods is not typical to JPA, which generally excels in its thin and simple API. There are 9 methods for setting parameters in a query, which is essential whenever using query parameters. In addition, there are 9 methods for extracting parameter values from a query. These get methods, which are new in JPA 2, are expected to be much less commonly used than the set methods.
Two set methods are demonstrated above - one for setting a named parameter and the other for setting an ordinal parameter. A third method is designated for setting a parameter in a Criteria API query. The reason for having nine set methods rather than just three is that JPA additionally provides three separate methods for setting `Date` parameters as well as three separate methods for setting `Calendar` parameters.

`Date` and `Calendar` parameter values require special methods in order to specify what they represent, such as a pure date, a pure time or a combination of date and time, as explained in detail in the Date and Time (Temporal) Types section.

Query Parameters in JPA

For example, the following invocation passes a Date object as a pure date (no time):

```
query.setParameter("date", new java.util.Date(), TemporalType.DATE);
```

Since `TemporalType.Date` represents a pure date, the time part of the newly constructed `java.util.Date` instance is discarded. This is very useful in comparison against a specific date, when time should be ignored.
The get methods support different ways to extract parameters and their values from a query, including by name (for named parameter), by position (for ordinal parameters) by Parameter object (for Criteria API queries), each with or without an expected type. There is also a method for extracting all the parameters as a set (`getParameters`) and a method for checking if a specified parameter has a value (`isBound`). These methods are not required for running queries and are expected to be less commonly used.

JPA Named Queries

A named query is a statically defined query with a predefined unchangeable query string. Using named queries instead of dynamic queries may improve code organization by separating the JPQL query strings from the Java code. It also enforces the use of query parameters rather than embedding literals dynamically into the query string and results in more efficient queries.

@NamedQuery and @NamedQueries Annotations

The following `@NamedQuery` annotation defines a query whose name is `"Country.findAll"` that retrieves all the Country objects in the database:

```
@NamedQuery(name="Country.findAll", query="SELECT c FROM Country c")
```

The `@NamedQuery` annotation contains four elements - two of which are required and two are optional. The two required elements, `name` and `query` define the name of the query and the query string itself and are demonstrated above. The two optional elements, `lockMode` and `hints`, provide static replacement for the `setLockMode` and `setHint` methods.
Every `@NamedQuery` annotation is attached to exactly one entity class or mapped superclass - usually to the most relevant entity class. But since the scope of named queries is the entire persistence unit, names should be selected carefully to avoid collision (e.g. by using the unique entity name as a prefix).

JPA Named Queries

It makes sense to add the above `@NamedQuery` to the Country entity class:

```
@Entity
@NamedQuery(name="Country.findAll", query="SELECT c FROM Country c")
public class Country {
   ...
}
```

Attaching multiple named queries to the same entity class requires wrapping them in a `@NamedQueries` annotation, as follows:

```
@Entity
@NamedQueries({
    @NamedQuery(name="Country.findAll",
                query="SELECT c FROM Country c"),
    @NamedQuery(name="Country.findByName",
                query="SELECT c FROM Country c WHERE c.name = :name"),
})
public class Country {
   ...
}
```

Note: Named queries can be defined in JPA XML mapping files instead of using the `@NamedQuery` annotation.

JPA Named Queries

Using Named Queries at Runtime

Named queries are represented at runtime by the same `Query` and `TypedQuery` interfaces but different `EntityManager` factory methods are used to instantiate them. The `createNamedQuery` method receives a query name and a result type and returns a `TypedQuery` instance:

```
TypedQuery<Country> query =
    em.createNamedQuery("Country.findAll", Country.class);
List<Country> results = query.getResultList();
```

Another form of `createNamedQuery` receives a query name and returns a `Query` instance:

```
Query query = em.createNamedQuery("SELECT c FROM Country c");
List results = query.getResultList();
```

One of the reasons that JPA requires the listing of managed classes in a persistence unit definition is to support named queries. Notice that named queries may be attached to any entity class or mapped superclass. Therefore, to be able to always locate any named query at runtime a list of all these managed persistable classes must be available.

JPA Criteria API Queries

The JPA Criteria API provides an alternative way for defining JPA queries, which is mainly useful for building dynamic queries whose exact structure is only known at runtime.

JPA Criteria API vs JPQL

JPQL queries are defined as strings, similarly to SQL. JPA criteria queries, on the other hand, are defined by instantiation of Java objects that represent query elements.
A major advantage of using the criteria API is that errors can be detected earlier, during compilation rather than at runtime. On the other hand, for many developers string based JPQL queries, which are very similar to SQL queries, are easier to use and understand.
For simple static queries - string based JPQL queries (e.g. as named queries) may be preferred. For dynamic queries that are built at runtime - the criteria API may be preferred. For example, building a dynamic query based on fields that a user fills at runtime in a form that contains many optional fields - is expected to be cleaner when using the JPA criteria API, because it eliminates the need for building the query using many string concatenation operations.
String based JPQL queries and JPA criteria based queries are equivalent in power and in efficiency. Therefore, choosing one method over the other is also a matter of personal preference.

JPA Criteria API Queries

First JPA Criteria Query

The following query string represents a minimal JPQL query:

```
SELECT c FROM Country c
```

An equivalent query can be built using the JPA criteria API as follows:

```
CriteriaBuilder cb = em.getCriteriaBuilder();

CriteriaQuery<Country> q = cb.createQuery(Country.class);
Root<Country> c = q.from(Country.class);
q.select(c);
```

The `CriteriaBuilder` interface serves as the main factory of criteria queries and criteria query elements. It can be obtained either by the `EntityManagerFactory`'s `getCriteriaBuilder` method or by the `EntityManager`'s `getCriteriaBuilder` method (both methods are equivalent).
In the example above a `CriteriaQuery` instance is created for representing the built query. Then a `Root` instance is created to define a range variable in the FROM clause. Finally, the range variable, c, is also used in the SELECT clause as the query result expression.

JPA Criteria API Queries

A `CriteriaQuery` instance is equivalent to a JPQL string and not to a `TypedQuery` instance. Therefore, running the query still requires a TypedQuery instance:

```
TypedQuery<Country> query = em.createQuery(q);
List<Country> results = query.getResultList();
```

Using the criteria API introduces some extra work, at least for simple static queries, since the equivalent JPQL query could simply be executed as follows:

```
TypedQuery<Country> query =
    em.createQuery("SELECT c FROM Country c", Country.class);
List<Country> results = query.getResultList();
```

Because eventually both types of queries are represented by a `TypedQuery` instance - query execution and query setting is similar, regardless of the way in which the query is built.

JPA Criteria API Queries

Parameters in Criteria Queries

The following query string represents a JPQL query with a parameter:

```
SELECT c FROM Country c WHERE c.population > :p
```

An equivalent query can be built using the JPA criteria API as follows:

```
  CriteriaBuilder cb = em.getCriteriaBuilder();

  CriteriaQuery<Country> q = cb.createQuery(Country.class);
  Root<Country> c = q.from(Country.class);
  ParameterExpression<Integer> p = cb.parameter(Integer.class);
  q.select(c).where(cb.gt(c.get("population"), p));
```

JPA Criteria API Queries

The `ParameterExpression` instance, `p`, is created to represent the query parameter. The `where` method sets the WHERE clause. As shown above, The `CriteriaQuery` interface supports method chaining. See the links in the next sections of this page for detailed explanations on how to set criteria query clauses and build criteria expressions.
Running this query requires setting the parameter:

```
TypedQuery<Country> query = em.createQuery(q);
query.setParameter(p, 10000000);
List<Country> results = query.getResultList();
```

The `setParameter` method takes a `Parameter` (or a `ParameterExpression`) instance as the first argument instead of a name or a position (which are used with string based JPQL parameters).

Setting and Tuning of JPA Queries

The `Query` and `TypedQuery` interfaces define various setting and tuning methods that may affect query execution if invoked before a query is run using `getResultList` or `getSingleResult`.

Result Range (setFirstResult, setMaxResults)

The `setFirstResult` and `setMaxResults` methods enable defining a result window that exposes a portion of a large query result list (hiding anything outside that window). The `setFirstResult` method is used to specify where the result window begins, i.e. how many results at the beginning of the complete result list should be skipped and ignored. The `setMaxResults` method is used to specify the result window size. Any result after hitting that specified maximum is ignored. These methods support the implementation of efficient result paging. For example, if each result page should show exactly `pageSize` results, and `pageId` represents the result page number (`0` for the first page), the following expression retrieves the results for a specified page:

```
List<Country> results =
    query.setFirstResult(pageIx * pageSize)
         .setMaxResults(pageSize)
         .getResultList();
```

These methods can be invoked in a single expression with `getResultList` since the setter methods in `Query` and `TypedQuery` support method chaining (by returning the query object on which they were invoked).

Setting and Tuning of JPA Queries

Flush Mode (setFlushMode)

Changes made to a database using an `EntityManager` `em` can be visible to anyone who uses `em`, even before committing the transaction (but not to users of other `EntityManager` instances). JPA implementations can easily make uncommitted changes visible in simple JPA operations, such as `find`. However, query execution is much more complex. Therefore, before a query is executed, uncommitted database changes (if any) have to be flushed to the database in order to be visible to the query.

Flush policy in JPA is represented by the `FlushModeType` enum, which has two values:

- `AUTO` - changes are flushed before query execution and on commit/flush.
- `COMMIT` - changes are flushed only on explicit commit/flush.

In most JPA implementations the default is AUTO. The default mode can be changed by the application, either at the `EntityManager` level as a default for all the queries in that `EntityManager` or at the level of a specific query, by overriding the default `EntityManager` setting:

```
// Enable query time flush at the EntityManager level:
em.setFlushMode(FlushModeType.AUTO);

// Enable query time flush at the level of a specific query:
query.setFlushMode(FlushModeType.AUTO);
```

Flushing changes to the database before every query execution affects performance significantly. Therefore, when performance is important, this issue has to be considered.

Setting and Tuning of JPA Queries

Lock Mode (setLockMode)

JPA uses optimistic locking to prevent concurrent changes to entity objects by multiple users. JPA 2 adds support for pessimistic locking. The `setLockMode` method sets a lock mode that has to be applied on all the result objects that the query retrieves. For example, the following query execution sets a pessimistic WRITE lock on all the result objects:

```
List<Country> results =
    query.setLockMode(LockModeType.PESSIMISTIC_WRITE)
         .getResultList();
```

Notice that when a query is executed with a requested pessimistic lock mode it could fail if locking fails, throwing a `LockTimeoutException`.

Setting and Tuning of JPA Queries

Query Hints

Additional settings can be applied to queries via hints.

Supported Query Hints

JPA supports the following query hints:

- `"javax.persistence.query.timeout"` - sets maximum query execution time in milliseconds. A `QueryTimeoutException` is thrown if timeout is exceeded.
- `"javax.persistence.lock.timeout"` - sets maximum waiting time for pessimistic locks, when pessimistic locking of query results is enabled. See the Lock Timeout section for more details about lock timeout.

Setting and Tuning of JPA Queries

Setting Query Hint (Scopes)

Query hints can be set in the following scopes (from global to local):

For the entire persistence unit - using a `persistence.xml` property:

```
    <properties>
       <property name="javax.persistence.query.timeout" value="3000"/>
    </properties>
```

For an `EntityManagerFactory` - using the `createEntityManagerFacotory` method:

```
  Map<String,Object> properties = new HashMap();
  properties.put("javax.persistence.query.timeout", 4000);
  EntityManagerFactory emf =
      Persistence.createEntityManagerFactory("pu", properties);
```

For an `EntityManager` - using the `createEntityManager` method:

```
  Map<String,Object> properties = new HashMap();
  properties.put("javax.persistence.query.timeout", 5000);
  EntityManager em = emf.createEntityManager(properties);
```

Setting and Tuning of JPA Queries

or using the `setProperty` method:

```
  em.setProperty("javax.persistence.query.timeout", 6000);
```

For a named query definition - using the `hints` element:

```
@NamedQuery(name="Country.findAll", query="SELECT c FROM Country c",
    hints={@QueryHint(name="javax.persistence.query.timeout", value="7000")})
```

For a specific query execution - using the `setHint` method (before query execution):

```
  query.setHint("javax.persistence.query.timeout", 8000);
```

A hint that is set in a global scope affects all the queries in that scope (unless it is overridden in a more local scope). For example, setting a query hint in an `EntityManager` affects all the queries that are created in that `EntityManager` (except queries with explicit setting of the same hint).

JPA Query Structure (JPQL / Criteria)

The syntax of the Java Persistence Query Language (JPQL) is very similar to the syntax of SQL. Having a SQL like syntax in JPA queries is an important advantage because SQL is a very powerful query language and many developers are already familiar with it.
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. For example, a JPQL query can retrieve and return entity objects rather than just field values from database tables, as with SQL. That makes JPQL more object oriented friendly and easier to use in Java.

JPA Query Structure (JPQL / Criteria)

JPQL Query Structure

As with SQL, a JPQL SELECT query also consists of up to 6 clauses in the following format:

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

The first two clauses, SELECT and FROM are required in every retrieval query (update and delete queries have a slightly different form). The other JPQL clauses, WHERE, GROUP BY, HAVING and ORDER BY are optional.
The structure of JPQL DELETE and UPDATE queries is simpler:

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

```
UPDATE ... SET ... [WHERE …]
```

Besides a few exceptions, JPQL is case insensitive. JPQL keywords, for example, can appear in queries either in upper case (e.g. SELECT) or in lower case (e.g. select). The few exceptions in which JPQL is case sensitive include mainly Java source elements such as names of entity classes and persistent fields, which are case sensitive. In addition, string literals are also case sensitive (e.g. "ORM" and "orm" are different values).

JPA Query Structure (JPQL / Criteria)

A Minimal JPQL Query

The following query retrieves all the Country objects in the database:

```
SELECT c FROM Country AS c
```

Because SELECT and FROM are mandatory, this demonstrates a minimal JPQL query.
The FROM clause declares one or more query variables (also known as identification variables). Query variables are similar to loop variables in programing languages. Each query variable represents iteration over objects in the database. A query variable that is bound to an entity class is referred to as a range variable. Range variables define iteration over all the database objects of a binding entity class and its descendant classes. In the query above, c is a range variable that is bound to the Country entity class and defines iteration over all the Country objects in the database.
The SELECT clause defines the query results. The query above simply returns all the Country objects from the iteration of the c range variable, which in this case is actually all the Country objects in the database.

SELECT clause (JPQL / Criteria API)

The ability to retrieve managed entity objects is a major advantage of JPQL. For example, the following query returns Country objects that become managed by the `EntityManager em`:

```
  TypedQuery<Country> query =
      em.createQuery("SELECT c FROM Country c", Country.class);
  List<Country> results = query.getResultList();
```

Because the results are managed entity objects they have all the support that JPA provides for managed entity objects, including transparent navigation to other database objects, transparent update detection, support for delete, etc.
Query results are not limited to entity objects. JPA 2 adds the ability to use almost any valid JPQL expression in SELECT clauses. Specifying the required query results more precisely can improve performance and in some cases can also reduce the amount of Java code needed.
Notice that query results must always be specified explicitly - JPQL does not support the "SELECT *" expression (which is commonly used in SQL).

SELECT clause (JPQL / Criteria API)

Projection of Path Expressions

JPQL queries can also return results which are not entity objects. For example, the following query returns country names as `String` instances, rather than `Country` objects:

```
SELECT c.name FROM Country AS c
```

Using path expressions, such as `c.name`, in query results is referred to as projection. The field values are extracted from (or projected out of) entity objects to form the query results.

The results of the above query are received as a list of `String` values:

```
TypedQuery<String> query = em.createQuery(
    "SELECT c.name FROM Country AS c", String.class);
List<String> results = query.getResultList();
```

Only singular value path expressions can be used in the SELECT clause. Collection and map fields cannot be included in the results directly, but their content can be added to the SELECT clause by using a bound JOIN variable in the FROM clause.

SELECT clause (JPQL / Criteria API)

Nested path expressions are also supported. For example, the following query retrieves the name of the capital city of a specified country:

```
SELECT c.capital.name FROM Country AS c WHERE c.name = :name
```

Because construction of managed entity objects has some overhead, queries that return non entity objects, as the two queries above, are usually more efficient. Such queries are useful mainly for displaying information efficiently. They are less productive with operations that update or delete entity objects, in which managed entity objects are needed.
Managed entity objects can, however, be returned from a query that uses projection when a result path expression resolves to an entity. For example, the following query returns a managed `City` entity object:

```
SELECT c.capital FROM Country AS c WHERE c.name = :name
```

Result expressions that represent anything but entity objects (e.g. values of system types and user defined embeddable objects) return as results value copies that are not associated with the containing entities. Therefore, embedded objects that are retrieved directly by a result path expression are not associated with an EntityManager and changes to them when a transaction is active are not propagated to the database.

SELECT clause (JPQL / Criteria API)

Multiple SELECT Expressions

The SELECT clause may also define composite results:

```
SELECT c.name, c.capital.name FROM Country AS c
```

The result list of this query contains `Object[]` elements, one per result. The length of each result `Object[]` element is 2. The first array cell contains the country name (`c.name`) and the second array cell contains the capital city name (`c.capital.name`).
The following code demonstrates running this query and processing the results:

```
TypedQuery<Object[]> query = em.createQuery(
    "SELECT c.name, c.capital.name FROM Country AS c", Object[].class);
List<Object[]> results = query.getResultList();
for (Object[] result : results) {
    System.out.println("Country: " + result[0] + ", Capital: " + result[1]);
}
```

As an alternative to representing compound results by `Object` arrays, JPA supports using custom result classes and result constructor expressions.

SELECT clause (JPQL / Criteria API)

Result Classes (Constructor Expressions)

JPA supports wrapping JPQL query results with instances of custom result classes. This is mainly useful for queries with multiple SELECT expressions, where custom result objects can provide an object oriented alternative to representing results as `Object[]` elements.

The fully qualified name of the result class is specified in a NEW expression, as follows:

```
SELECT NEW example.CountryAndCapital(c.name, c.capital.name)
FROM Country AS c
```

This query is identical to the previous query above except that now the result list contains `CountryAndCapital` instances rather than `Object[]` elements.

SELECT clause (JPQL / Criteria API)

The result class must have a compatible constructor that matches the SELECT result expressions, as follows:

```
package example;

public class CountryAndCapital {
    public String countryName;
    public String capitalName;

    public CountryAndCapital(String countryName, String capitalName) {
        this.countryName = countryName;
        this.capitalName = capitalName;
    }
}
```

SELECT clause (JPQL / Criteria API)

The following code demonstrates running this query:

```
String queryStr =
    "SELECT NEW example.CountryAndCapital(c.name, c.capital.name) " +
    "FROM Country AS c";
TypedQuery<CountryAndCapital> query =
    em.createQuery(queryStr, CountryAndCapital.class);
List<CountryAndCapital> results = query.getResultList();
```

Any class with a compatible constructor can be used as a result class. It could be a JPA managed class (e.g. an entity class) but it could also be a lightweight 'transfer' class that is only used for collecting and processing query results.
If an entity class is used as a result class, the result entity objects are created in the NEW state, which means that they are not managed. Such entity objects are missing the JPA functionality of managed entity objects (e.g. transparent navigation and transparent update detection), but they are more lightweight, they are built faster and they consume less memory.

SELECT clause (JPQL / Criteria API)

SELECT DISTINCT

Queries that use projection may return duplicate results. For example, the following query may return the same currency more than once:

```
SELECT c.currency FROM Country AS c WHERE c.name LIKE ‘I%'
```

Both Italy and Ireland (whose name starts with 'I') use Euro as their currency. Therefore, the query result list contains "Euro" more than once.
Duplicate results can be eliminated easily in JPQL by using the DISTINCT keyword:

```
SELECT DISTINCT c.currency FROM Country AS c WHERE c.name LIKE ‘I%'
```

The only difference between SELECT and SELECT DISTINCT is that the later filters duplicate results. Filtering duplicate results might have some effect on performance, depending on the size of the query result list and other factors.

SELECT clause (JPQL / Criteria API)

SELECT in Criteria Queries

The criteria query API provides several ways for setting the SELECT clause.

Single Selection

Setting a single expression SELECT clause is straightforward.
For example, the following JPQL query:

```
SELECT DISTINCT c.currency FROM Country c
```

can be built as a criteria query as follows:

```
CriteriaQuery<Country> q = cb.createQuery(Country.class);
Root<Country> c = q.from(Country.class);
q.select(c.get("currency")).distinct(true);
```

The `select` method takes one argument of type `Selection` and sets it as the SELECT clause content (overriding previously set SELECT content if any). Every valid criteria API expression can be used as selection, because all the criteria API expressions are represented by a sub interface of `Selection` - `Expression` (and its descendant interfaces).
The distinct method can be used to eliminate duplicate results as demonstrated in the above code (using method chaining).

SELECT clause (JPQL / Criteria API)

Multi Selection

The `Selection` interface is also a super interface of `CompoundSelection`, which represents multi selection (which is not a valid expression by its own and can be used only in the SELECT clause). The `CriteriaBuilder` interface provides three factory methods for building `CompoundSelection` instances - `array`, `tuple` and `construct`.

CriteriaBuilder's array

The following JPQL query:

```
SELECT c.name, c.capital.name FROM Country c
```

can be defined using the criteria API as follows:

```
CriteriaQuery<Object[]> q = cb.createQuery(Object[].class);
Root<Country> c = q.from(Country.class);
q.select(cb.array(c.get("name"), c.get("capital").get("name")));
```

The `array` method builds a `CompoundSelection` instance, which represents results as arrays. The following code demonstrates execution of the query and iteration over the results:

```
List<Object[]> results = em.createQuery(q).getResultList();
for (Object[] result : results) {
    System.out.println("Country: " + result[0] + ", Capital: " + result[1]);
}
```

SELECT clause (JPQL / Criteria API)

CriteriaBuilder's tuple

The `Tuple` interface can be used as a clean alternative to `Object[]`:

```
CriteriaQuery<Tuple> q = cb.createTupleQuery();
Root<Country> c = q.from(Country.class);
q.select(cb.tuple(c.get("name"), c.get("capital").get("name")));
```

The `tuple` method builds a `CompoundSelection` instance, which represents `Tuple` results. The following code demonstrates execution of the query and iteration over the results:

```
List<Tuple> results = em.createQuery(q).getResultList();
for (Tuple t : results) {
    System.out.println("Country: " + t.get(0)  + ", Capital: " + t.get(1));
}
```

The `Tuple` interface defines several other methods for accessing the result data.

SELECT clause (JPQL / Criteria API)

CriteriaBuilder's construct

JPQL user defined result objects are also supported by the JPA criteria query API:

```
CriteriaQuery<CountryAndCapital> q = cb.createQuery(CountryAndCapital.class);
Root<Country> c = q.from(Country.class);
q.select(cb.construct(CountryAndCapital.class,
    c.get("name"), c.get("capital").get("name")));
```

The `construct` method builds a `CompoundSelection` instance, which represents results as instances of a user defined class (`CountryAndCapital` in the above example).

The following code demonstrates execution of the query:

```
List<CountryAndCapital> results = em.createQuery(q).getResultList();
```

As expected - the result objects are `CountryAndCapital` instances.

SELECT clause (JPQL / Criteria API)

CriteriaQuery's multiselect

In the above examples, `CompoundSelection` instances were first built by a `CriteriaBuilder` factory method and then passed to the `CriteriaQuery`'s `select` method.
The `CriteriaQuery` interface provides a shortcut method - `multiselect`, which takes a variable number of arguments representing multiple selections, and builds a `CompoundSelection` instance based on the expected query results.

For example, the following invocation of `multiselect`:

```
q.multiselect(c.get("name"), c.get("capital").get("name"));
```

is equivalent to using `select` with one of the factory methods (`array`, `tuple` or `construct`) as demonstrated above.
The behavior of the `multiselect` method depends on the query result type (as set when `CriteriaQuery` is instantiated):

- For expected `Object` and `Object[]` result type - `array` is used.
- For expected `Tuple` result - `tuple` is used.
- For any other expected result type - `construct` is used.

FROM clause (JPQL / Criteria API)

The FROM clause declares query identification variables that represent iteration over objects in the database. A query identification variable is similar to a variable of a Java enhanced for loop in a program, since both are used for iteration over objects.

Range Variables

Range variables are query identification variables that iterate over all the database objects of a specific entity class hierarchy (i.e. an entity class and all its descendant entity classes). Identification variables are always polymorphic. JPQL does not provide a way to exclude descendant classes from iteration at the FROM clause level. JPA 2, however, adds support for filtering instances of specific types at the WHERE clause level by using a type expression. For example, in the following query, c iterates over all the Country objects in the database:

```
SELECT c FROM Country AS c
```

The AS keyword is optional, and the same query can also be written as follows:

```
SELECT c FROM Country c
```

FROM clause (JPQL / Criteria API)

By default, the name of an entity class in a JPQL query is the unqualified name of the class (e.g. just `Country` with no package name). The default name can be overridden by specifying another name explicitly in the `@Entity`'s `name` annotation element.
Multiple range variables are allowed. For example, the following query returns all the pairs of countries that share a common border:

```
SELECT c1, c2 FROM Country c1, Country c2
WHERE c2 MEMBER OF c1.neighbors
```

Multiple variables are equivalent to nested loops in a program. The FROM clause above defines two loops. The outer loop uses `c1` to iterate over all the `Country` objects. The inner loop uses `c2` to also iterate over all the `Country` objects. A similar query with no `WHERE` clause would return all the possible combinations of two countries. The `WHERE` clause filters any pair of countries that do not share a border, returning as results only neighbor countries.

FROM clause (JPQL / Criteria API)

Caution is required when using multiple range variables. Iteration over about 1,000,000 database objects with a single range variable might be acceptable. But iteration over the same objects with two range variables forming nested loops (outer and inner) might prevent query execution within a reasonable response time.
Database Management Systems (DBMS) try to optimize execution of multi-variable queries. Whenever possible, full nested iteration over the entire Cartesian product is avoided. The above query, for example, can be executed as follows. An outer loop iterates with c1 over all the Country objects in the database. An inner loop iterates with c2 only over the neighbors collection of the outer c1. In this case, by propagation of a WHERE constraint to the FROM phase, a full iteration over the Cartesian product is avoided.

FROM clause (JPQL / Criteria API)

[INNER] JOIN

As discussed above, range variables represent iteration over all the database objects of a specified entity type. JPQL provides an additional type of identification variable, a join variable, which represent a more limited iteration over specified collections of objects.
The following query uses one range variable and one join variable:

```
SELECT c1, c2 FROM Country c1 INNER JOIN c1.neighbors c2
```

In JPQL, JOIN can only appear in a FROM clause. The INNER keyword is optional (i.e. INNER JOIN is equivalent to JOIN). `c1` is declared as a range variable that iterates over all the `Country` objects in the database. `c2` is declared as a join variable that is bound to the `c1.neighbors` path and iterates only over objects in that collection.
You might have noticed that this query is equivalent to the previous neighbors query, which has two range variables and a WHERE clause. However, this second query form that uses a join variable is preferred. Besides being shorter and cleaner, the second query describes the right and efficient way for executing the query (which is using a full range outer loop and a collection limited inner loop) without relying on DBMS optimizations.

FROM clause (JPQL / Criteria API)

It is quite common for JPQL queries to have a single range variable that serves as a root and additional join variables that are bound to path expressions. Join variables can also be bound to path expressions that are based on other join variables that appear earlier in the FROM clause.
Join variables can also be bound to a single value path expression. For example:

```
SELECT c, p.name FROM Country c JOIN c.capital p
```

In this case, the inner loop iterates over a single object because every `Country c` has only one `Capital p`. Join variables that are bound to a single value expression are less commonly used because usually they can be replaced by a simpler long path expression (which is not an option for a collection). For example:

```
SELECT c, c.capital.name FROM Country c
```

One exception is when OUTER JOIN is required because path expressions function as implicit INNER JOIN variables.

FROM clause (JPQL / Criteria API)

LEFT [OUTER] JOIN

To understand the purpose of OUTER JOIN, consider the following INNER JOIN query that retrieves pairs of (country name, capital name):

```
SELECT c.name, p.name FROM Country c JOIN c.capital p
```

The FROM clause defines iteration over (country, capital) pairs. A country with no capital city (e.g. Nauru, which does not have an official capital) is not part of any iterated pair and is therefore excluded from the query results. INNER JOIN simply skips any outer variable value (e.g. any `Country`) that has no matching inner variable (e.g. a `Capital`).
The behavior of OUTER JOIN is different, as demonstrated by the following query variant:

```
SELECT c, p.name FROM Country c LEFT OUTER JOIN c.capital p
```

The OUTER keyword is optional (LEFT OUTER JOIN is equivalent to LEFT JOIN). When using OUTER JOIN, if a specific outer variable does not have any matching inner value it gets at least a `NULL` value as a matching value in the FROM iteration. Therefore, a `Country c` with no `Capital` city has a minimum representation of (c, `NULL`) in the FROM iteration.
For example, unlike the INNER JOIN variant of this query that skips Nauru completely, the OUTER JOIN variant returns Nauru with a `NULL` value as its capital.

FROM clause (JPQL / Criteria API)

[LEFT [OUTER] | INNER] JOIN FETCH

JPA support of transparent navigation and fetch makes it very easy to use, since it provides the illusion that all the database objects are available in memory for navigation. But this feature could also cause performance problems.
For example, let's look at the following query execution and result iteration:

```
TypedQuery<Country> query =
    em.createQuery("SELECT c FROM Country c", Country.class);
List<Country> results = query.getResultList();
for (Country c : results) {
    System.out.println(c.getName() + " => " + c.getCapital().getName());
}
```

The query returns only `Country` instances. Consequently, the loop that iterates over the results is inefficient, since retrieval of the referenced `Capital` objects is performed one at a time, i.e. the number of round trips to the database is much larger than necessary.

FROM clause (JPQL / Criteria API)

A simple solution is to use the following query, which returnes exactly the same result objects (Country instances):

```
SELECT c FROM Country c JOIN FETCH c.capital
```

The JOIN FETCH expression is not a regular JOIN and it does not define a JOIN variable. Its only purpose is specifying related objects that should be fetched from the database with the query results on the same round trip. Using this query improves the efficiency of iteration over the result `Country` objects because it eliminates the need for retrieving the associated `Capital` objects separately.
Notice, that if the Coutry and Capital objects are needed only for their names - the following report query could be even more efficient:

```
SELECT c.name, c.capital.name FROM Country c
```

FROM clause (JPQL / Criteria API)

Reserved Identifiers

The name of a JPQL query variable must a valid Java identifier but cannot be one of the following reserved words:

```
ABS, ALL, AND, ANY, AS, ASC, AVG, BETWEEN, BIT_LENGTH, BOTH, BY, CASE,
CHAR_LENGTH, CHARACTER_LENGTH, CLASS, COALESCE, CONCAT, COUNT, CURRENT_DATE,
CURRENT_TIME, CURRENT_TIMESTAMP,DELETE, DESC, DISTINCT, ELSE, EMPTY, END, ENTRY,
ESCAPE, EXISTS, FALSE, FETCH,FROM, GROUP, HAVING, IN, INDEX, INNER, IS, JOIN,
KEY, LEADING, LEFT, LENGTH, LIKE,LOCATE, LOWER, MAX, MEMBER, MIN, MOD, NEW, NOT,
NULL, NULLIF, OBJECT, OF, OR, ORDER, OUTER, POSITION, SELECT, SET, SIZE, SOME,
SQRT, SUBSTRING, SUM, THEN,TRAILING, TRIM, TRUE, TYPE, UNKNOWN, UPDATE, UPPER,
VALUE, WHEN, WHERE.
```

JPQL variables as well as all the reserved identifiers in the list above are case insensitive. Therefore, ABS, abs, Abs and aBs are all invalid variable names.

FROM clause (JPQL / Criteria API)

FROM query identification variables are represented in criteria queries by sub interfaces of `From`:

- Range variables are represented by the `Root` interface.
- Join variables are represented by the `Join` interface (and its sub interfaces).

Criteria Query Roots

The `CriteriaQuery`'s `from` method serves as a factory of `Root` instances.
For example, the following JPQL query, which defines two uncorrelated range variables - c1, c2:

```
SELECT c1, c2 FROM Country c1, Country c2
```

can be built as a criteria query using the following code:

```
CriteriaQuery<Country> q = cb.createQuery(Country.class);
Root<Country> c1 = q.from(Country.class);
Root<Country> c2 = q.from(Country.class);
q.multiselect(c1, c2);
```

Unlike other `CriteriaQuery` methods - invocation of the `from` method does not override a previous invocation of that method. Every time the `from` method is invoked - a new variable is added to the query.

FROM clause (JPQL / Criteria API)

Criteria Query Joins

JOIN variables are represented in criteria queries by the `Join` interface (and its sub interfaces).
For example, the following JPQL query:

```
SELECT c, p.name FROM Country c LEFT OUTER JOIN c.capital p
```

can be built as a criteria query using the following code:

```
CriteriaQuery<Country> q = cb.createQuery(Country.class);
Root<Country> c = q.from(Country.class);
Join<Country> p = c.join("capital", JoinType.LEFT);
q.multiselect(c, p.get("name"));
```

The `From` interface provides various forms of the `join` method. Every invocation of `join` adds a new JOIN variable to the query. Since `From` is the super interface of both `Root` and `Join` - `join` methods can be invoked on Root instances (as demonstrated above) as well as on previously built `Join` instances.

FROM clause (JPQL / Criteria API)

Criteria Query Fetch Joins

The following JPQL query:

```
SELECT c FROM Country c JOIN FETCH c.capital
```

can be built as a criteria query using the following code:

```
CriteriaQuery<Country> q = cb.createQuery(Country.class);
Root<Country> c = q.from(Country.class);
Fetch<Country,Capital> p = c.fetch("capital");
q.select(c);
```

Several forms of the `fetch` method are defined in the `Path` interface, which represents path expressions and is also the super interface of the `Root` and `Join` interfaces.

WHERE clause (JPQL / Criteria API)

The WHERE clause adds filtering capabilities to the FROM-SELECT structure. It is essential in any JPQL query that retrieves selective objects from the database. Out of the four optional clauses of JPQL queries, the WHERE clause is definitely the most frequently used.

How a WHERE Clause Works

The following query retrieves only countries with population size that exceeds a specified limit, which is represented by the parameter `p`:

```
SELECT c FROM Country c WHERE c.population > :p
```

The FROM clause of this query defines an iteration over all the `Country` objects in the database using the `c` range variable. Before passing these `Country` objects to the SELECT clause for collecting as query results, the WHERE clause gets an opportunity to function as a filter. The boolean expression in the WHERE clause, which is also known as the WHERE predicate, defines which objects to accept. Only `Country` objects for which the predicate expression evaluates to TRUE are passed to the SELECT clause and then collected as query results.

WHERE clause (JPQL / Criteria API)

WHERE Predicate and Indexes

Formally, the WHERE clause functions as a filter between the FROM and the SELECT clauses. Practically, if a proper index is available, filtering is done earlier during FROM iteration. In the above population query, if an index is defined on the `population` field JPA can use that index to iterate directly on `Country` objects that satisfy the WHERE predicate. For entity classes with millions of objects in the database there is a huge difference in query execution time if proper indexes are defined.

WHERE Filter in Multi Variable Queries

In a multi-variable query the FROM clause defines iteration on tuples. In this case the WHERE clause filters tuples before passing them to the SELECT clause.
For example, the following query retrieves all the countries with population size that exceeds a specified limit and also have an official language from a specified set of languages:

```
SELECT c, l FROM Country c JOIN c.languages l
WHERE c.population > :p AND l IN :languages
```

The FROM clause of this query defines iteration over (country, language) pairs. Only pairs that satisfy the WHERE clause are passed through to the SELECT.
In multi-variable queries the number of tuples for iteration might be very large even if the database is small, making indexes even more essential.

WHERE clause (JPQL / Criteria API)

JPQL Expressions in WHERE

The above queries demonstrate only a small part of the full capabilities of a WHERE clause. The real power of the JPQL WHERE clause is derived from the rich JPQL expression syntax, which includes many operators (arithmetic operators, relational operators, logical operators) and functions (numeric functions, string functions, collection functions). The WHERE predicate is always a boolean JPQL expression. JPQL expressions are also used in other JPQL query clauses but they are especially dominant in the WHERE clause.

WHERE clause (JPQL / Criteria API)

The `CriteriaQuery` interface provides two `where` methods for setting the WHERE clause.

Single Restriction

The first `where` method takes one `Expression<Boolean>` argument and uses it as the WHERE clause content (overriding previously set WHERE content if any). For example, the following JPQL query:

```
SELECT c FROM Country c WHERE c.population > :p
```

can be built by using the criteria query API as follows:

```
CriteriaQuery<Country> q = cb.createQuery(Country.class);
Root<Country> c = q.from(Country.class);
q.select(c);
ParameterExpression<Integer> p = cb.parameter(Integer.class);
q.where(cb.gt(c.get("population"), p));
```

WHERE clause (JPQL / Criteria API)

Multiple Restrictions

The second where method takes a variable number of arguments of `Predicate` type and uses an AND conjunction as the WHERE clause content (overriding previously set WHERE content if any):

For example, the following JPQL query:

```
SELECT c FROM Country WHERE c.population > :p AND c.area < :a
```

can be built as a criteria query as follows:

```
  CriteriaQuery q = cb.createQuery(Country.class);
  Root<Country> c = q.from(Country.class);
  q.select(c);
  ParameterExpression<Integer> p = cb.parameter(Integer.class);
  ParameterExpression<Integer> a = cb.parameter(Integer.class);
  q.where(
      cb.gt(c.get("population"), p),
      cb.lt(c.get("area"), a)
  );
```

WHERE clause (JPQL / Criteria API)

The `where` setting above is equivalent to explicitly building an AND conjunction, as so:

```
q.where(
    cb.and(
        cb.gt(c.get("population"), p),
        cb.lt(c.get("area"), a)
    )
);
```

The variable argument form of the `where` method always uses AND. Therefore, using OR requires building an OR expression explicitly:

```
q.where(
    cb.or(
        cb.gt(c.get("population"), p),
        cb.lt(c.get("area"), a)
    )
);
```

See the Logical Operators page for explanations on boolean expressions and predicates that can be used in a criteria query WHERE clause.

GROUP BY and HAVING clauses

The GROUP BY clause enables grouping of query results. A JPQL query with a GROUP BY clause returns properties of generated groups instead of individual objects and fields.
The position of a GROUP BY clause in the query execution order is after the FROM and WHERE clauses, but before the SELECT clause. When a GROUP BY clause exists in a JPQL query, database objects (or tuples of database objects) that are generated by the FROM clause iteration and pass the WHERE clause filtering (if any) are sent to grouping by the GROUP BY clauses before arriving at the SELECT clause.

GROUP BY and HAVING clauses

GROUP BY as DISTINCT (no Aggregates)

The following query groups all the countries by their first letter:

```
SELECT SUBSTRING(c.name, 1, 1)
FROM Country c
GROUP BY SUBSTRING(c.name, 1, 1);
```

The FROM clause defines iteration over all the Country objects in the database. The GROUP BY clause groups these Country objects by the first letter of the country name. The next step is to pass the groups to the SELECT clause which returns the first letters as result.
Note that the query above might not be supported by some JPA implementations. Only identification variables and path expressions are currently supported in the GROUP BY clause by all the JPA implementations.
Grouping the Country objects makes them inaccessible to the SELECT clause as individuals.
Therefore, the SELECT clause can only use properties of the groups, which include:

- The properties that are used for grouping (each group has unique value combination).
- Aggregate calculations (count, sum, avg, max, min) that are carried out on all the objects (or the object tuples) in the group.

The aggregate calculation gives the GROUP BY clause its power. Actually, without aggregate calculations - the GROUP BY functions merely as a DISTINCT operator. For example, the above query (which does not use aggregates) is equivalent to the following query:

```
SELECT DISTINCT SUBSTRING(c.name, 1, 1) FROM Country c
```

GROUP BY and HAVING clauses

GROUP BY with Aggregate Functions

JPQL supports the five aggregate functions of SQL:

- `COUNT` - returns a `long` value representing the number of elements.
- `SUM` - returns the sum of numeric values.
- `AVG` - returns the average of numeric values as a `double` value.
- `MIN` - returns the minimum of comparable values (numeric, strings, dates).
- `MAX` - returns the maximum of comparable values (numeric, strings, dates).

The following query counts for every letter the number of countries with names that start with that letter and the number of different currencies that are used by these countries:

```
SELECT SUBSTRING(c.name, 1, 1), COUNT(c), COUNT(DISTINCT c.currency)
FROM Country c
GROUP BY SUBSTRING(c.name, 1, 1);
```

The query returns `Object[]` arrays of length 3, in which the first cell contains the initial letter as a `String` object, the second cell contains the number of countries in that letter's group as a Long object and the third cell contains the distinct number of currencies that are in use by countries in that group. The DISTINCT keyword in a COUNT aggregate expression, as demonstrated above), eliminates duplicate values when counting.

GROUP BY and HAVING clauses

Only the COUNT aggregate function can be applied to entity objects directly. Other aggregate functions are applied to fields of objects in the group by using path expressions.
The following query groups countries in Europe by their currency, and for each group returns the currency and the cumulative population size in countries that use that currency:

```
SELECT c.currency, SUM(c.population)
FROM Country c
WHERE 'Europe' MEMBER OF c.continents
GROUP BY c.currency
```

Because grouping is performed in this query on a path expression, this query is standard and it is expected to be supported by all JPA implementations.

GROUP BY and HAVING clauses

GROUP BY with HAVING

Groups in JPQL grouping queries can be filtered using the HAVING clause. The HAVING clause for the GROUP BY clause is like the WHERE clause for the FROM clause.
The following query uses HAVING to change the previous query in a way that single country groups are ignored:

```
SELECT c.currency, SUM(c.population)
FROM Country c
WHERE 'Europe' MEMBER OF c.continents
GROUP BY c.currency
HAVING COUNT(c) > 1
```

The HAVING clause stands as a filter between the GROUP BY clause and the SELECT clause in such a way that only groups that are accepted by the HAVING filter are passed to the SELECT clause. The same restrictions on SELECT clause in grouping queries also apply to the HAVING clause, which means that individual object fields are inaccessible. Only group properties which are the expressions that are used for grouping (e.g. c.currency) and aggregate expressions are allowed in the HAVING clause.

GROUP BY and HAVING clauses

Global Aggregates (no GROUP BY)

JPQL supports a special form of aggregate queries that do not have a GROUP BY clause in which all the FROM/WHERE objects (or object tuples) are considered as one group.
For example, the following query returns the sum and average population in countries that use the English language:

```
SELECT SUM(c.population), AVG(c.population)
FROM Country c
WHERE 'English' MEMBER OF c.languages
```

All the Country objects that pass the FROM/WHERE phase are considered as one group, for which the cumulative population size and the average population size is then calculated. Any JPQL query that contains an aggregate expression in the SELECT clause is considered a grouping query with all the attached restrictions, even when a GROUP BY clause is not specified. Therefore, in this case, only aggregate functions can be specified in the SELECT clause and individual objects and their fields become inaccessible.

GROUP BY and HAVING clauses

The `CriteriaQuery` interface provides methods for setting the GROUP BY and HAVING clauses.
For example, the following JPQL query:

```
SELECT c.currency, SUM(c.population)
FROM Country c
WHERE 'Europe' MEMBER OF c.continents
GROUP BY c.currency
HAVING COUNT(c) > 1
```

can be built using the criteria query API as follows:

```
CriteriaQuery<Country> q = cb.createQuery(Country.class);
  Root<Country> c = q.from(Country.class);
  q.multiselect(c.get("currency"), cb.sum(c.get("population")));
  q.where(cb.isMember("Europe", c.get("continents")));
  q.groupBy(c.get("currency"));
  g.having(cb.gt(cb.count(c), 1));
```

GROUP BY and HAVING clauses

The `CriteriaBuilder` interface provides methods for building aggregate expressions:

- `count`, `countDistinct` - return a long type expression representing the number of elements.
- `sum` , `sumAsLong`, `sumAsDouble` - return an expression representing the sum of values.
- `avg` - returns a double type expression representing the average of numeric values.
- `min`, `least` - return an expression representing the minimum of comparable values.
- `max`, `greatest` - return an expression representing the maximum of comparable values.

The `groupBy` method takes a variable number of arguments specifying one or more grouping expressions (or a list of expressions in another form of `groupBy`).
Setting a HAVING clause is very similar to setting a WHERE clause. As with the WHERE clause - two forms of the `having` method are provided. One `having` form takes an `Expression<Boolean>` argument and the other `having` form takes a variable number of `Predicate` arguments (and uses an AND conjunction).
When a groupBy or a `having` method is invoked, previously set values (if any) are discarded.

ORDER BY clause (JPQL / Criteria API)

The ORDER BY clause specifies a required order for the query results. Any JPQL query that does not include an ORDER BY clause produces results in an undefined and non-deterministic order.

ORDER BY Expressions

The following query returns names of countries whose population size is at least one million people, ordered by the country name:

```
SELECT c.name FROM Country c WHERE c.population > 1000000 ORDER BY c.name
```

When an ORDER BY clause exists it is the last to be executed. First the FROM clause produces objects for examination and the WHERE clause selects which objects to collect as results. Then the SELECT clause builds the results by evaluating the result expressions. Finally the results are ordered by evaluation of the the ORDER BY expressions.
Only expressions that are derived directly from expressions in the SELECT clause are allowed in the ORDER BY clause. The following query, for example, is invalid because the ORDER BY expression is not part of the results:

```
SELECT c.name
FROM Country c
WHERE c.population > 1000000
ORDER BY c.population
```

ORDER BY clause (JPQL / Criteria API)

On the other hand, the following query is valid because, given a `Country c`, the `c.population` expression can be evaluated from `c`:

```
SELECT c
FROM Country c
WHERE c.population > 1000000
ORDER BY c.population
```

Any JPQL expression whose type is comparable (i.e. numbers, strings and date values) and is derived from the SELECT expressions can be used in the ORDER BY clause. Some JPA implementation are more restrictive. Path expressions are supported by all the JPA implementations but support of other JPQL expressions is vendor dependent.
Query results can also be ordered by multiple order expressions. In this case, the first expression is the primary order expression. Any additional order expression is used to order results for which all the previous order expressions produce the same values.

ORDER BY clause (JPQL / Criteria API)

The following query returns Country objects ordered by currency as the primary sort key and by name as the secondary sort key:

```
SELECT c.currency, c.name
FROM Country c
ORDER BY c.currency, c.name
```

To avoid repeating result expressions in the ORDER BY JPQL supports defining aliases for SELECT expressions and then using the aliases in the ORDER BY clause. The following query is equivalent to the query above:

```
SELECT c.currency AS currency, c.name AS name
FROM Country c
ORDER BY currency, name
```

Alias variables are referred to as result variables to distinguish them from the identification variables that are defined in the FROM clause.

ORDER BY clause (JPQL / Criteria API)

Order Direction (ASC, DESC)

The default ordering direction is ascending. Therefore, when ascending order is required it is usually omitted even though it could be specified explicitly, as follows:

```
SELECT c.name FROM Country c ORDER BY c.name ASC
```

On the other hand, to apply descending order the DESC keyword must be added explicitly to the order expression:

```
SELECT c.name FROM Country c ORDER BY c.name DESC
```

ORDER BY clause (JPQL / Criteria API)

Grouping (GROUP BY) Order

The ORDER BY clause is always the last in the query processing chain. If a query contains both an ORDER BY clause and a GROUP BY clause the SELECT clause receives groups rather than individual objects and ORDER BY can order these groups. For example:

```
SELECT c.currency, SUM(c.population)
FROM Country c
WHERE 'Europe' MEMBER OF c.continents
GROUP BY c.currency
HAVING COUNT(c) > 1
ORDER BY c.currency
```

The ORDER BY clause in the above query orders the results by the currency name. Without an ORDER BY clause the result order would be undefined.

ORDER BY in Criteria ORDER BY clause (JPQL / Criteria API)

The `CriteriaQuery` interface provides methods for setting the ORDER BY clause.
For example, the following JPQL query:

```
SELECT c
FROM Country c
ORDER BY c.currency, c.population DESC
```

can be built using the criteria query API as follows:

```
CriteriaQuery<Country> q = cb.createQuery(Country.class);
Root<Country> c = q.from(Country.class);
q.select(c);
q.orderBy(cb.asc(c.get("currency")), cb.desc(c.get("population")));
```

Unlike other methods for setting criteria query clauses - the `orderBy` method takes a variable number of `Order` instances as arguments (or a list of `Order`) rather than `Expression` instances.
The `Order` interface is merely a thin wrapper around `Expression`, which adds order direction - either ascending (ASC) or descending (DESC). The `CriteriaBuilder`'s `asc` and `desc` methods (which are demonstrated above) take an expression and return an ascending or descending `Order` instance (respectively).

DELETE Queries in JPA/JPQL

As explained in chapter 2, entity objects can be deleted from the database by:

- Retrieving the entity objects into an `EntityManager`.
- Removing these objects from the `EntityManager` within an active transaction, either explicitly by calling the `remove` method or implicitly by a cascading operation.
- Applying changes to the database by calling the `commit` method.

JPQL DELETE queries provide an alternative way for deleting entity objects. Unlike `SELECT` queries, which are used to retrieve data from the database, DELETE queries do not retrieve data from the database, but when executed, delete specified entity objects from the database. Removing entity objects from the database using a DELETE query may be slightly more efficient than retrieving entity objects and then removing them, but it should be used cautiously because bypassing the `EntityManager` may break its synchronization with the database. For example, the `EntityManager` may not be aware that a cached entity object in its persistence context has been removed from the database by a DELETE query. Therefore, it is a good practice to use a separate `EntityManager` for DELETE queries.

As with any operation that modifies the database, DELETE queries can only be executed within an active transaction and the changes are visible to other users (which use other `EntityManager` instances) only after `commit`.

DELETE Queries in JPA/JPQL

Delete All Queries

The simplest form of a DELETE query removes all the instances of a specified entity class (including instances of subclasses) from the database.
For example, the following three equivalent queries delete all the Country instances:

```
DELETE FROM Country       // no variable
DELETE FROM Country c     // an optional variable
DELETE FROM Country AS c  // AS + an optional variable
```

DELETE queries are executed using the `executeUpdate` method:

```
  int deletedCount = em.createQuery("DELETE FROM Country").executeUpdate();
```

A `TransactionRequiredException` is thrown if no transaction is active.
On success - the executeUpdate method returns the number of objects that have been deleted by the query.

DELETE Queries in JPA/JPQL

Selective Deletion

The structure of DELETE queries is very simple relative to the structure of SELECT queries. DELETE queries cannot include multiple variables and JOIN, and cannot include the GROUP BY, HAVING and ORDER BY clauses.
A WHERE clause, which is essential for removing selected entity objects, is supported.
For example, the following query deletes the countries with population size that is smaller than a specified limit:

```
DELETE FROM Country c WHERE c.population < :p
```

The query can be executed as follows:

```
  Query query = em.createQuery(
      "DELETE FROM Country c WHERE c.population < :p");
  int deletedCount = query.setParameter(p, 100000).executeUpdate();
```

UPDATE SET Queries in JPA/ JPQL

Existing entity objects can be updated, as explained in chapter 2, by:

- Retrieving the entity objects into an `EntityManager`.
- Updating the relevant entity object fields within an active transaction.
- Applying changes to the database by calling the `commit` method.

JPQL UPDATE queries provide an alternative way for updating entity objects. Unlike SELECT queries, which are used to retrieve data from the database, UPDATE queries do not retrieve data from the database, but when executed, update the content of specified entity objects in the database.

Updating entity objects in the database using an UPDATE query may be slightly more efficient than retrieving entity objects and then updating them, but it should be used cautiously because bypassing the `EntityManager` may break its synchronization with the database. For example, the `EntityManager` may not be aware that a cached entity object in its persistence context has been modified by an UPDATE query. Therefore, it is a good practice to use a separate `EntityManager` for UPDATE queries.

As with any operation that modifies the database, UPDATE queries can only be executed within an active transaction and the changes are visible to other users (which use other `EntityManager` instances) only after `commit`.

UPDATE SET Queries in JPA/ JPQL

Update All Queries

The simpler form of UPDATE queries acts on all the instances of a specified entity class in the database (including instances of subclasses).
For example, the following three equivalent queries increase the population size of all the countries by 10%:

```
UPDATE Country SET population = population * 11 / 10
UPDATE Country c SET c.population = c.population * 11 / 10
UPDATE Country AS c SET c.population = c.population * 11 / 10
```

The UPDATE clause defines exactly one range variable (with or without an explicit variable name) for iteration. Multiple variables and JOIN are not supported. The SET clause defines one or more field update expressions (using the range variable name - if defined).
Multiple field update expressions, separated by commas, are also allowed. For example:

```
UPDATE Country SET population = 0, area = 0
```

UPDATE queries are executed using the `executeUpdate` method:

```
  Query query = em.createQuery(
      "UPDATE Country SET population = 0, area = 0");
  int updateCount = em.executeUpdate();
```

A `TransactionRequiredException` is thrown if no transaction is active.
On success - the `executeUpdate` method returns the number of objects that have been modified by the query.

UPDATE SET Queries in JPA/ JPQL

Selective Update

UPDATE queries cannot include the GROUP BY, HAVING and ORDER BY clauses, but the WHERE clause, which is essential for updating selected entity objects, is supported.
For example, the following query updates the population size of countries whose population size that is smaller than a specified limit:

```
UPDATE Country
SET population = population * 11 / 10
WHERE c.population < :p
```

The query can be executed as follows:

```
  Query query = em.createQuery(
      "UPDATE Country SET population = population * 11 / 10 " +
      "WHERE c.population < :p");
  int updateCount = query.setParameter(p, 100000).executeUpdate();
```