JPQL is used to define searches against persistent entities independent of the underlying database. JPQL is a query language that takes its roots in the syntax of SQL, which is the standard language for database interrogation. But the main difference is that in SQL the results obtained are in the form of rows and columns (tables), whereas JPQL uses an entity or a collection of entities. JPQL syntax is object oriented and therefore more easily understood by developers whose experience is limited to object-oriented languages. Developers manage their entity domain model, not a table structure, by using the dot notation (e.g., `myClass.myAttribute`).

Under the hood, JPQL uses the mechanism of mapping to transform a JPQL query into language comprehensible by an SQL database. The query is executed on the underlying database with SQL and JDBC calls, and then entity instances have their attributes set and are returned to the application—all in a very simple and powerful manner using a rich query syntax.

The simplest JPQL query selects all the instances of a single entity.

```
SELECT b
FROM Book b
```

If you know SQL, this should look familiar to you. Instead of selecting from a table, JPQL selects entities, here `Book`. The `FROM` clause is also used to give an alias to the entity: `b` is an alias for `Book`. The `SELECT` clause of the query indicates that the result type of the query is the `b` entity (the `Book`). Executing this statement will result in a list of zero or more `Book` instances.

To restrict the result, add search criteria; you can use the `WHERE` clause as follows:

```
SELECT b
FROM Book b
WHERE b.title = 'H2G2'
```

The alias is used to navigate across entity attributes through the dot operator. Since the `Book` entity has a persistent attribute named `title` of type `String`, `b.title` refers to the `title` attribute of the `Book` entity. Executing this statement will result in a list of zero or more `Book` instances that have a title equal to *H2G2*.

The simplest select query consists of two mandatory parts: the `SELECT` and the `FROM` clause. `SELECT` defines the format of the query results. The `FROM` clause defines the entity or entities from which the results will be obtained, and the optional `WHERE`, `ORDER BY`, `GROUP BY`, and `HAVING` clauses can be used to restrict or order the result of a query. Listing 6-20 defines a simplified syntax of a JPQL statement.

Listing 6-20. Simplied JPQL Statement Syntax

```
SELECT <select clause>
FROM <from clause>
[WHERE <where clause>]
[ORDER BY <order by clause>]
[GROUP BY <group by clause>]
[HAVING <having clause>]
```

Listing 6-20 defines a `SELECT` statement, but `DELETE` and `UPDATE` statements can also be used to perform delete and update operations across multiple instances of a specific entity class.

Select

The `SELECT` clause follows the path expressions syntax and results in one of the following forms: an entity, an entity attribute, a constructor expression, an aggregate function, or some sequence of these Path expressions are the

building blocks of queries and are used to navigate on entity attributes or across entity relationships (or a collection of entities) via the dot (.) navigation using the following syntax:

```
SELECT [DISTINCT] <expression> [[AS] <identification variable>]
expression ::= { NEW | TREAT | AVG | MAX | MIN | SUM | COUNT }
```

A simple SELECT returns an entity. For example, if a Customer entity has an alias called c, SELECT c will return an entity or a list of entities.

```
SELECT c
FROM Customer c
```

But a SELECT clause can also return attributes. If the Customer entity has a first name, SELECT c.firstName will return a String or a collection of Strings with the first names.

```
SELECT c.firstName
FROM Customer c
```

To retrieve the first name and the last name of a customer, you create a list containing the following two attributes:

```
SELECT c.firstName, c.lastName
FROM Customer c
```

Since JPA 2.0, an attribute can be retrieved depending on a condition (using a CASE WHEN ... THEN ... ELSE ... END expression). For example, instead of retrieving the price of a book, a statement can return a computation of the price (e.g., 50% discount) depending on the publisher (e.g., 50% discount on the Apress books, 20% discount for all the other books).

```
SELECT CASE b.editor WHEN 'Apress'
                     THEN b.price * 0.5
                     ELSE b.price * 0.8
       END
FROM Book b
```

If a Customer entity has a one-to-one relationship with Address, c.address refers to the address of the customer, and the result of the following query will return not a list of customers but a list of addresses:

```
SELECT c.address
FROM Customer c
```

Navigation expressions can be chained together to traverse complex entity graphs. Using this technique, path expressions such as c.address.country.code can be constructed, referring to the country code of the customer's address.

```
SELECT c.address.country.code
FROM Customer c
```

A constructor may be used in the SELECT expression to return an instance of a Java class initialized with the result of the query. The class doesn't have to be an entity, but the constructor must be fully qualified and match the attributes.

```
SELECT NEW org.agoncal.javaee7.CustomerDTO(c.firstName, c.lastName, c.address.street1)
FROM Customer c
```

The result of this query is a list of `CustomerDTO` objects that have been instantiated with the new operator and initialized with the first name, last name, and street of the customers.

Executing these queries will return either a single value or a collection of zero or more entities (or attributes) including duplicates. To remove the duplicates, the `DISTINCT` operator must be used.

```
SELECT DISTINCT c
FROM Customer c

SELECT DISTINCT c.firstName
FROM Customer c
```

The result of a query may be the result of an aggregate function applied to a path expression. The following aggregate functions can be used in the `SELECT` clause: `AVG`, `COUNT`, `MAX`, `MIN`, `SUM`. The results may be grouped in the `GROUP BY` clause and filtered using the `HAVING` clause.

```
SELECT COUNT(c)
FROM Customer c
```

Scalar expressions also can be used in the `SELECT` clause of a query as well as in the `WHERE` and `HAVING` clauses. These expressions can be used on numeric (`ABS`, `SQRT`, `MOD`, `SIZE`, `INDEX`), String (`CONCAT`, `SUBSTRING`, `TRIM`, `LOWER`, `UPPER`, `LENGTH`, `LOCATE`), and date-time (`CURRENT_DATE`, `CURRENT_TIME`, `CURRENT_TIMESTAMP`) values.

From

The `FROM` clause of a query defines entities by declaring identification variables. An *identification variable*, or *alias*, is an identifier that can be used in the other clauses (`SELECT`, `WHERE`, etc.). The syntax of the `FROM` clause consists of an entity and an alias. In the following example, `Customer` is the entity and `c` the identification variable:

```
SELECT c
FROM Customer c
```

Where

The `WHERE` clause of a query consists of a conditional expression used to restrict the result of a `SELECT`, `UPDATE`, or `DELETE` statement. The `WHERE` clause can be a simple expression or a set of conditional expressions used to filter the query.

The simplest way to restrict the result of a query is to use the attribute of an entity. For example, the following query selects all customers named Vincent:

```
SELECT c
FROM Customer c
WHERE c.firstName = 'Vincent'
```

You can further restrict queries by using the logical operators `AND` and `OR`. The following example uses `AND` to select all customers named Vincent, living in France:

```
SELECT c
FROM Customer c
WHERE c.firstName = 'Vincent' AND c.address.country = 'France'
```

The WHERE clause also uses comparison operators: =, >, >=, <, <=, <>, [NOT] BETWEEN, [NOT] LIKE, [NOT] IN, IS [NOT] NULL, IS [NOT] EMPTY, [NOT] MEMBER [OF]. The following shows an example using two of these operators:

```
SELECT c
FROM Customer c
WHERE c.age > 18

SELECT c
FROM Customer c
WHERE c.age NOT BETWEEN 40 AND 50

SELECT c
FROM Customer c
WHERE c.address.country IN ('USA', 'Portugal')
```

The LIKE expression consists of a String and optional escape characters that define the match conditions: the underscore (_) for single-character wildcards and the percent sign (%) for multicharacter wildcards.

```
SELECT c
FROM Customer c
WHERE c.email LIKE '%mail.com'
```

Binding Parameters

Until now, the WHERE clauses shown herein have only used fixed values. In an application, queries frequently depend on parameters. JPQL supports two types of parameter-binding syntax, allowing dynamic changes to the restriction clause of a query: positional and named parameters.

Positional parameters are designated by the question mark (?) followed by an integer (e.g., ?1). When the query is executed, the parameter numbers that should be replaced need to be specified.

```
SELECT c
FROM Customer c
WHERE c.firstName = ?1 AND c.address.country = ?2
```

Named parameters can also be used and are designated by a String identifier that is prefixed by the colon (:) symbol. When the query is executed, the parameter names that should be replaced need to be specified.

```
SELECT c
FROM Customer c
WHERE c.firstName = :fname AND c.address.country = :country
```

In the "Queries" section later in this chapter, you will see how an application binds parameters.

Subqueries

A subquery is a SELECT query that is embedded within a conditional expression of a WHERE or HAVING clause. The results of the subquery are evaluated and interpreted in the conditional expression of the main query. To retrieve the youngest customers from the database, a subquery with a MIN(age) is first executed and its result evaluated in the main query.

```
SELECT c
FROM Customer c
WHERE c.age = (SELECT MIN(cust. age) FROM Customer cust))
```

Order By

The ORDER BY clause allows the entities or values that are returned by a SELECT query to be ordered. The ordering applies to the entity attribute specified in this clause followed by the ASC or DESC keyword. The keyword ASC specifies that ascending ordering be used; DESC, the inverse, specifies that descending ordering be used. Ascending is the default and can be omitted.

```
SELECT c
FROM Customer c
WHERE c.age > 18
ORDER BY c.age DESC
```

Multiple expressions may also be used to refine the sort order.

```
SELECT c
FROM Customer c
WHERE c.age > 18
ORDER BY c.age DESC, c.address.country ASC
```


Group By and Having

The GROUP BY construct enables the aggregation of result values according to a set of properties. The entities are divided into groups based on the values of the entity field specified in the GROUP BY clause. To group customers by country and count them, use the following query:

```
SELECT c.address.country, count(c)
FROM Customer c
GROUP BY c.address.country
```

The GROUP BY defines the grouping expressions (`c.address.country`) over which the results will be aggregated and counted (`count(c)`). Note that expressions that appear in the GROUP BY clause must also appear in the SELECT clause.

The HAVING clause defines an applicable filter after the query results have been grouped, similar to a secondary WHERE clause, filtering the result of the GROUP BY. Using the previous query, by adding a HAVING clause, a result of countries other than the UK can be returned.

```
SELECT c.address.country, count(c)
FROM Customer c
GROUP BY c.address.country
HAVING c.address.country <> 'UK'
```

GROUP BY and HAVING can only be used within a SELECT clause (not a DELETE or an UPDATE).

Bulk Delete

You know how to remove an entity with the `EntityManager.remove()` method and query a database to retrieve a list of entities that correspond to certain criteria. To remove a list of entities, you can execute a query, iterate through it, and remove each entity individually. Although this is a valid algorithm, it is terrible in terms of performance (too many database accesses). There is a better way to do it: bulk deletes.

JPQL performs bulk delete operations across multiple instances of a specific entity class. These are used to delete a large number of entities in a single operation. The `DELETE` statement looks like the `SELECT` statement, as it can have a restricting `WHERE` clause and use parameters. As a result, the number of entity instances affected by the operation is returned. The syntax of the `DELETE` statement is

```
DELETE FROM <entity name> [[AS] <identification variable>]
[WHERE <where clause>]
```

As an example, to delete all customers younger than 18, you can use a bulk removal via a `DELETE` statement.

```
DELETE FROM Customer c
WHERE c.age < 18
```

Bulk Update

Bulk updates of entities are accomplished with the `UPDATE` statement, setting one or more attributes of the entity subject to conditions in the `WHERE` clause. The `UPDATE` statement syntax is

```
UPDATE <entity name> [[AS] <identification variable>]
SET <update statement> {, <update statement>}*
[WHERE <where clause>]
```

Rather than deleting all the young customers, their first name can be changed to "too young" with the following statement:

```
UPDATE Customer c
SET c.firstName = 'TOO YOUNG'
WHERE c.age < 18
```

Queries

You've seen the JPQL syntax and how to describe statements using different clauses (`SELECT`, `FROM`, `WHERE`, etc.). But how do you integrate a JPQL statement to your application? The answer: through queries. JPA 2.1 has five different types of queries that can be used in code, each with a different purpose.

- *Dynamic queries*: This is the simplest form of query, consisting of nothing more than a JPQL query string dynamically specified at runtime.
- *Named queries*: Named queries are static and unchangeable.
- *Criteria API*: JPA 2.0 introduced the concept of object-oriented query API.
- *Native queries*: This type of query is useful to execute a native SQL statement instead of a JPQL statement.
- *Stored procedure queries*: JPA 2.1 brings a new API to call stored procedures.

The central point for choosing from these five types of queries is the `EntityManager` interface, which has several factory methods, listed in Table 6-4, returning either a `Query`, a `TypedQuery`, or a `StoredProcedureQuery` interface (both `TypedQuery` and `StoredProcedureQuery` extend `Query`). The `Query` interface is used in cases when the result type is `Object`, and `TypedQuery` is used when a typed result is preferred. `StoredProcedureQuery` is used to control stored procedure query execution.

Table 6-4. *`EntityManager` Methods for Creating Queries*

Method	Description
`Query createQuery(String jpqlString)`	Creates an instance of `Query` for executing a JPQL statement for dynamic queries
`Query createNamedQuery(String name)`	Creates an instance of `Query` for executing a named query (in JPQL or in native SQL)
`Query createNativeQuery(String sqlString)`	Creates an instance of `Query` for executing a native SQL statement
`Query createNativeQuery(String sqlString, Class resultClass)`	Native query passing the class of the expected results
`Query createNativeQuery(String sqlString, String resultSetMapping)`	Native query passing a result set mapping
`<T> TypedQuery<T> createQuery(CriteriaQuery<T> criteriaQuery)`	Creates an instance of `TypedQuery` for executing a criteria query
`<T> TypedQuery<T> createQuery(String jpqlString, Class<T> resultClass)`	Typed query passing the class of the expected results
`<T> TypedQuery<T> createNamedQuery(String name, Class<T> resultClass)`	Typed query passing the class of the expected results
`StoredProcedureQuery createStoredProcedureQuery(String procedureName)`	Creates a `StoredProcedureQuery` for executing a stored procedure in the database
1. `StoredProcedureQuery createStoredProcedureQuery(String procedureName, Class... resultClasses)`	Stored procedure query passing classes to which the result sets are to be mapped
2. `StoredProcedureQuery createStoredProcedureQuery(String procedureName, String... resultSetMappings)`	Stored procedure query passing the result sets mapping
`StoredProcedureQuery createNamedStoredProcedureQuery(String name)`	Creates a query for a named stored procedure

When you obtain an implementation of the `Query`, `TypedQuery`, or `StoredProcedureQuery` interface through one of the factory methods in the `EntityManager` interface, a rich API controls it. The `Query` API, shown in Listing 6-21, is used for static queries (i.e., named queries) and dynamic queries using JPQL, and native queries in SQL. The `Query` API also supports parameter binding and pagination control.

Listing 6-21. Query API

```
public interface Query {

  // Executes a query and returns a result
  List getResultList();
  Object getSingleResult();
  int executeUpdate();

  // Sets parameters to the query
  Query setParameter(String name, Object value);
  Query setParameter(String name, Date value, TemporalType temporalType);
  Query setParameter(String name, Calendar value, TemporalType temporalType);
  Query setParameter(int position, Object value);
  Query setParameter(int position, Date value, TemporalType temporalType);
  Query setParameter(int position, Calendar value, TemporalType temporalType);
  <T> Query setParameter(Parameter<T> param, T value);
  Query setParameter(Parameter<Date> param, Date value, TemporalType temporalType);
  Query setParameter(Parameter<Calendar> param, Calendar value, TemporalType temporalType);

  // Gets parameters from the query
  Set<Parameter<?>> getParameters();
  Parameter<?> getParameter(String name);
  Parameter<?> getParameter(int position);
  <T> Parameter<T> getParameter(String name, Class<T> type);
  <T> Parameter<T> getParameter(int position, Class<T> type);
  boolean isBound(Parameter<?> param);
  <T> T getParameterValue(Parameter<T> param);
  Object getParameterValue(String name);
  Object getParameterValue(int position);

  // Constrains the number of results returned by a query
  Query setMaxResults(int maxResult);
  int getMaxResults();
  Query setFirstResult(int startPosition);
  int getFirstResult();

  // Sets and gets query hints
  Query setHint(String hintName, Object value);
  Map<String, Object> getHints();

  // Sets the flush mode type to be used for the query execution
  Query setFlushMode(FlushModeType flushMode);
  FlushModeType getFlushMode();

  // Sets the lock mode type to be used for the query execution
  Query setLockMode(LockModeType lockMode);
  LockModeType getLockMode();

  // Allows access to the provider-specific API
  <T> T unwrap(Class<T> cls);
}
```

The methods that are mostly used in this API are ones that execute the query itself. To execute a SELECT query, you have to choose between two methods depending on the required result.

- The getResultList() method executes the query and returns a list of results (entities, attributes, expressions, etc.).
- The getSingleResult() method executes the query and returns a single result (throws a NonUniqueResultException if more than one result is found).

To execute an update or a delete, the executeUpdate() method executes the bulk query and returns the number of entities affected by the execution of the query.

As you saw in the "JPQL" section earlier, a query can use parameters that are either named (e.g., :myParam) or positional (e.g., ?1). The Query API defines several setParameter methods to set parameters before executing a query.

When you execute a query, it can return a large number of results. Depending on the application, these can be processed together or in chunks (e.g., a web application only displays ten rows at one time). To control the pagination, the Query interface defines setFirstResult() and setMaxResults() methods to specify the first result to be received (numbered from zero) and the maximum number of results to return relative to that point.

The flush mode indicates to the persistence provider how to handle pending changes and queries. There are two possible flush mode settings: AUTO and COMMIT. AUTO (the default) means that the persistence provider is responsible for ensuring pending changes are visible to the processing of the query. COMMIT is when the effect of updates made to entities does not overlap with changed data in the persistence context.

Queries can be locked using the setLockMode(LockModeType) method. Locks are intended to provide a facility that enables the effect of repeatable read whether optimistically or pessimistically.

The following sections illustrate the five different types of queries using some of the methods just described.

Dynamic Queries

Dynamic queries are defined on the fly as needed by the application. To create a dynamic query, use the EntityManager.createQuery() method, which takes a String as a parameter that represents a JPQL query.

In the following code, the JPQL query selects all the customers from the database. The result of this query is a list, so when you invoke the getResultList() method, it returns a list of Customer entities (List<Customer>). However, if you know that your query only returns a single entity, use the getSingleResult() method. It returns a single entity and avoids the work of retrieving the data as a list.

```
Query query = em.createQuery("SELECT c FROM Customer c");
List<Customer> customers = query.getResultList();
```

This JPQL query returns a Query object. When you invoke the query.getResultList() method, it returns a list of untyped objects. If you want the same query to return a list of type Customer, you need to use the TypedQuery as follows:

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

This query string can also be dynamically created by the application, which can then specify a complex query at runtime not known ahead of time. String concatenation is used to construct the query dynamically depending on the criteria.

```
String jpqlQuery = "SELECT c FROM Customer c";
if (someCriteria)
    jpqlQuery += " WHERE c.firstName = 'Betty'";
query = em.createQuery(jpqlQuery);
List<Customer> customers = query.getResultList();
```

The previous query retrieves customers named Betty, but you might want to introduce a parameter for the first name. There are two possible choices for passing a parameter: using names or positions. In the following example, I use a named parameter called :fname (note the : symbol) in the query and bound it with the setParameter method:

```
query = em.createQuery("SELECT c FROM Customer c where c.firstName = :fname");
query.setParameter("fname", "Betty");
List<Customer> customers = query.getResultList();
```

Note that the parameter name fname does not include the colon used in the query. The code using a position parameter would look like the following:

```
query = em.createQuery("SELECT c FROM Customer c where c.firstName = ?1");
query.setParameter(1, "Betty");
List<Customer> customers = query.getResultList();
```

If you need to use pagination to display the list of customers by chunks of ten, you can use the setMaxResults method as follows:

```
query = em.createQuery("SELECT c FROM Customer c", Customer.class);
query.setMaxResults(10);
List<Customer> customers = query.getResultList();
```

An issue to consider with dynamic queries is the cost of translating the JPQL string into an SQL statement at runtime. Because the query is dynamically created and cannot be predicted, the persistence provider has to parse the JPQL string, get the ORM metadata, and generate the equivalent SQL. The performance cost of processing each of these dynamic queries can be an issue. If you have static queries that are unchangeable and want to avoid this overhead, then you can use named queries instead.

Named Queries

Named queries are different from dynamic queries in that they are static and unchangeable. In addition to their static nature, which does not allow the flexibility of a dynamic query, named queries can be more efficient to execute because the persistence provider can translate the JPQL string to SQL once the application starts, rather than every time the query is executed.

Named queries are static queries expressed in metadata inside either a @NamedQuery annotation or the XML equivalent. To define these reusable queries, annotate an entity with the @NamedQuery annotation, which takes two elements: the name of the query and its content. So let's change the Customer entity and statically define three queries using annotations (see Listing 6-22).

Listing 6-22. The Customer Entity Defining Named Queries

```
@Entity
@NamedQueries({
  @NamedQuery(name = "findAll", query="select c from Customer c"),
  @NamedQuery(name = "findVincent", ↪
              query="select c from Customer c where c.firstName = 'Vincent'"),
  @NamedQuery(name = "findWithParam", ↪
              query="select c from Customer c where c.firstName = :fname")
})
```

```
public class Customer {

  @Id @GeneratedValue
  private Long id;
  private String firstName;
  private String lastName;
  private Integer age;
  private String email;
  @OneToOne
  @JoinColumn(name = "address_fk")
  private Address address;

  // Constructors, getters, setters
}
```

Because the Customer entity defines more than one named query, it uses the @NamedQueries annotation, which takes an array of @NamedQuery. The first query, called findAll, selects all customers from the database with no restriction (no WHERE clause). The findWithParam query uses the parameter :fname to restrict customers by their first name. Listing 6-22 shows an array of @NamedQuery, but, if the Customer only had one query, it would have been defined as follows:

```
@Entity
@NamedQuery(name = "findAll", query="select c from Customer c")
public class Customer {...}
```

The way to execute these named queries resembles the way dynamic queries are used. The EntityManager.createNamedQuery() method is invoked and passed to the query name defined by the annotations. This method returns a Query or a TypedQuery that can be used to set parameters, the max results, fetch modes, and so on. To execute the findAll query, write the following code:

```
Query query = em.createNamedQuery("findAll");
```

Again, if you need to type the query to return a list of Customer objects, you'll need to use the TypedQuery as follows:

```
TypedQuery<Customer> query = em.createNamedQuery("findAll", Customer.class);
```

The following is a fragment of code calling the findWithParam named query, passing the parameter :fname, and setting the maximum result to 3:

```
Query query = em.createNamedQuery("findWithParam");
query.setParameter("fname", "Vincent");
query.setMaxResults(3);
List<Customer> customers = query.getResultList();
```

Because most of the methods of the Query API return a Query object, you can use the following elegant shortcut to write queries. You call methods one after the other (setParameter().setMaxResults(), etc.).

```
Query query = em.createNamedQuery("findWithParam").setParameter("fname", "Vincent") ↪
                                                   .setMaxResults(3);
```

Named queries are useful for organizing query definitions and powerful for improving application performance. The organization comes from the fact that the named queries are defined statically on entities and are typically placed on the entity class that directly corresponds to the query result (here the `findAll` query returns customers, so it should be defined in the `Customer` entity).

There is a restriction in that the name of the query is scoped to the persistence unit and must be unique within that scope, meaning that only one `findAll` method can exist. A `findAll` query for customers and a `findAll` query for addresses should be named differently. A common practice is to prefix the query name with the entity name. For example, the `findAll` query for the `Customer` entity would be named `Customer.findAll`.

Another problem is that the name of the query, which is a String, is manipulated, and, if you make a typo or refactor your code, you may get some exceptions indicating that the query doesn't exist. To limit the risks, you can replace the name of a query with a constant. Listing 6-23 shows how to refactor the `Customer` entity.

Listing 6-23. The Customer Entity Defining a Named Query with a Constant

```
@Entity
@NamedQuery(name = Customer.FIND_ALL, query="select c from Customer c"),
public class Customer {

  public static final String FIND_ALL = "Customer.findAll";

  // Attributes, constructors, getters, setters
}
```

The `FIND_ALL` constant identifies the `findAll` query nonambiguously by prefixing the name of the query with the name of the entity. The same constant is then used in the `@NamedQuery` annotation, and you can use this constant to execute the query as follows:

```
TypedQuery<Customer> query = em.createNamedQuery(Customer.FIND_ALL, Customer.class);
```

Criteria API (or Object-Oriented Queries)

Until now, I've been using Strings to write JPQL (dynamic or named queries) statements. This has the advantage of writing a database query concisely but the inconvenience of being error prone and difficult for an external framework to manipulate: it is a String, you end up concatenating Strings and so many typos can be made. For example, you could have typos on JPQL keywords (`SLECT` instead of `SELECT`), class names (`Custmer` instead of `Customer`), or attributes (`firstname` instead of `firstName`). You can also write a syntactically incorrect statement (`SELECT c WHERE c.firstName = 'John' FROM Customer`). Any of these mistakes will be discovered at runtime, and it may sometimes be difficult to find where the bug comes from.

JPA 2.0 created a new API, called *Criteria API* and defined in the package `javax.persistence.criteria`. It allows you to write any query in an object-oriented and syntactically correct way. Most of the mistakes that a developer could make writing a statement are found at compile time, not at runtime. The idea is that all the JPQL keywords (`SELECT`, `UPDATE`, `DELETE`, `WHERE`, `LIKE`, `GROUP BY`...) are defined in this API. In other words, the Criteria API supports everything JPQL can do but with an object-based syntax. Let's have a first look at a query that retrieves all the customers named "Vincent." In JPQL, it would look as follows:

```
SELECT c FROM Customer c WHERE c.firstName = 'Vincent'
```

This JPQL statement is rewritten in Listing 6-24 in an object-oriented way using the Criteria API.

Listing 6-24. A Criteria Query Selecting All the Customers Named Vincent

```
CriteriaBuilder builder = em.getCriteriaBuilder();
CriteriaQuery<Customer> criteriaQuery = builder.createQuery(Customer.class);
Root<Customer> c = criteriaQuery.from(Customer.class);
criteriaQuery.select(c).where(builder.equal(c.get("firstName"), "Vincent"));
Query query = em.createQuery(criteriaQuery).getResultList();
List<Customer> customers = query.getResultList();
```

Without going into too much detail, you can see that the `SELECT`, `FROM`, and `WHERE` keywords have an API representation through the methods `select()`, `from()`, and `where()`. And this rule applies for every JPQL keyword. Criteria queries are constructed through the `CriteriaBuilder` interface that is obtained by the `EntityManager` (the `em` attribute in Listings 6-24 and 6-25). It contains methods to construct the query definition (this interface defines keywords such as `desc()`, `asc()`, `avg()`, `sum()`, `max()`, `min()`, `count()`, `and()`, `or()`, `greaterThan()`, `lowerThan()`...). The other role of the `CriteriaBuilder` is to serve as the main factory of criteria queries (`CriteriaQuery`) and criteria query elements. This interface defines methods such as `select()`, `from()`, `where()`, `orderBy()`, `groupBy()`, and `having()`, which have the equivalent meaning in JPQL. In Listing 6-24, the way you get the alias **c** (as in `SELECT c FROM Customer`) is through the `Root` interface (`Root<Customer> c`). Then you just have to use the builder, the query, and the root to write any JPQL statement you want: from the simplest (select all the entities from the database) to the most complex (joins, subqueries, case expressions, functions . . .).

Listing 6-25. A Criteria Query Selecting All the Customers Older Than 40

```
CriteriaBuilder builder = em.getCriteriaBuilder();
CriteriaQuery<Customer> criteriaQuery = builder.createQuery(Customer.class);
Root<Customer> c = criteriaQuery.from(Customer.class);
criteriaQuery.select(c).where(builder.greaterThan(c.get("age").as(Integer.class), 40));
Query query = em.createQuery(criteriaQuery).getResultList();
List<Customer> customers = query.getResultList();
```

Let's take another example. Listing 6-25 shows a query that retrieves all the customers older than 40. The `c.get("age")` gets the attribute age from the `Customer` entity and checks if it's greater than 40.

I started this section saying that the Criteria API allows you to write error-free statements. But it's not completely true yet. When you look at Listings 6-24 and 6-25, you can still see some strings (`"firstName"` and `"age"`) that represent the attributes of the `Customer` entity. So typos can still be made. In Listing 6-25, we even need to cast the age into an `Integer` (`c.get("age").as(Integer.class)`) because there is no other way to discover that the age attribute is of type `Integer`. To solve these problems, the Criteria API comes with a static metamodel class for each entity, bringing type safety to the API.

Type-Safe Criteria API

Listings 6-24 and 6-25 are almost typesafe: each JPQL keyword can be represented by a method of the `CriteriaBuilder` and `CriteriaQuery` interface. The only missing part is the attributes of the entity that are string based: the way to refer to the customer's `firstName` attribute is by calling `c.get("firstName")`. The `get` method takes a String as a parameter. Type-safe Criteria API solves this by overriding this method with a path expression from the metamodel API classes bringing type safety.

Listing 6-26 shows the `Customer` entity with several attributes of different type (`Long`, `String`, `Integer`, `Address`).

Listing 6-26. A Customer Entity with Several Attributes' Types

```
@Entity
public class Customer {

  @Id @GeneratedValue
  private Long id;
  private String firstName;
  private String lastName;
  private Integer age;
  private String email;
  private Address address;

  // Constructors, getters, setters
}
```

To bring type safety, JPA 2.1 can generate a static metamodel class for each entity. The convention is that each entity X will have a metadata class called X_ (with an underscore). So, the Customer entity will have its metamodel representation described in the Customer_ class shown in Listing 6-27.

Listing 6-27. The Customer_ Class Describing the Metamodel of Customer

```
@Generated("EclipseLink")
@StaticMetamodel(Customer.class)
public class Customer_ {

  public static volatile SingularAttribute<Customer, Long> id;
  public static volatile SingularAttribute<Customer, String> firstName;
  public static volatile SingularAttribute<Customer, String> lastName;
  public static volatile SingularAttribute<Customer, Integer> age;
  public static volatile SingularAttribute<Customer, String> email;
  public static volatile SingularAttribute<Customer, Address> address;
}
```

In the static metamodel class, each attribute of the Customer entity is defined by a subclass of javax.persistence.metamodel.Attribute (CollectionAttribute, ListAttribute, MapAttribute, SetAttribute, or SingularAttribute). Each of these attributes uses generics and is strongly typed (e.g., SingularAttribute<Customer, Integer>, age). Listing 6-28 shows the exact same code as Listing 6-25 but revisited with the static metamodel class (the c.get("age") is turned into c.get(Customer_.age)). Another advantage of type safety is that the metamodel defines the age attribute as being an Integer, so there is no need to cast the attribute into an Integer using as(Integer.class).

Listing 6-28. A Type-Safe Criteria Query Selecting All the Customers Older Than 40

```
CriteriaBuilder builder = em.getCriteriaBuilder();
CriteriaQuery<Customer> criteriaQuery = builder.createQuery(Customer.class);
Root<Customer> c = criteriaQuery.from(Customer.class);
criteriaQuery.select(c).where(builder.greaterThan(c.get(Customer_.age), 40));
Query query = em.createQuery(criteriaQuery).getResultList();
List<Customer> customers = query.getResultList();
```

Again, these are just examples of what you can do with the Criteria API. It is a very rich API that is completely defined in Chapter 5 (*Metamodel API*) and Chapter 6 (*Criteria API*) of the JPA 2.1 specification.

■ **Note** The classes used in the static metamodel, such as `Attribute` or `SingularAttribute`, are standard and defined in the package `javax.persistence.metamodel`. But the generation of the static metamodel classes is implementation specific. EclipseLink uses an internal class called `CanonicalModelProcessor`. This processor can be invoked by your integrated development environment (IDE) while you develop a Java command, an Ant task, or a Maven plug-in.

Native Queries

JPQL has a very rich syntax that allows you to handle entities in any form and ensures portability across databases. JPA enables you to use specific features of a database by using native queries. Native queries take a native SQL statement (`SELECT`, `UPDATE`, or `DELETE`) as the parameter and return a `Query` instance for executing that SQL statement. However, native queries are not expected to be portable across databases.

If the code is not portable, why not use JDBC calls? The main reason to use JPA native queries rather than JDBC calls is because the result of the query will be automatically converted back to entities. If you want to retrieve all the customer entities from the database using SQL, you need to use the `EntityManager.createNativeQuery()` method that has as parameters the SQL query and the entity class that the result should be mapped to.

```
Query query = em.createNativeQuery("SELECT * FROM t_customer", Customer.class);
List<Customer> customers = query.getResultList();
```

As you can see in the preceding code fragment, the SQL query is a String that can be dynamically created at runtime (just like JPQL dynamic queries). Again, the query could be complex, and, because the persistence provider doesn't know in advance, it will interpret it each time. Like named queries, native queries can use annotations to define static SQL queries. Named native queries are defined using the `@NamedNativeQuery` annotation, which must be placed on any entity (see code below). Like JPQL named queries, the name of the query must be unique within the persistence unit.

```
@Entity
@NamedNativeQuery(name = "findAll", query="select * from t_customer")
@Table(name = "t_customer")
public class Customer {...}
```

Stored Procedure Queries

So far all the different queries (JPQL or SQL) have the same purpose: send a query from your application to the database that will execute it and send back a result. Stored procedures are different in the sense that they are actually stored in the database itself and executed within this database.

A stored procedure is a subroutine available to applications that access a relational database. Typical usage could be extensive or complex processing that requires execution of several SQL statements or a data-intensive repetitive task. Stored procedures are usually written in a proprietary language close to SQL and therefore not easily portable across database vendors. But storing the code inside the database even in a nonportable way provides many advantages, like

- Better performance due to precompilation of the stored procedure as well as reutilizing its execution plan,
- Keeping statistics on the code to keep it optimized,
- Reducing the amount of data passed over a network by keeping the code on the server,

- Altering the code in a central location without replicating in several different programs,
- Stored procedures, which can be used by multiple programs written in different languages (not just Java),
- Hiding the raw data by allowing only stored procedures to gain access to the data, and
- Enhancing security controls by granting users permission to execute a stored procedure independently of underlying table permissions.

Let's take a look at a practical example: archiving old books and CDs. After a certain date books and CDs have to be archived in a certain warehouse, meaning they have to be physically transferred from a warehouse to a reseller. Archiving books and CDs can be a time-consuming process as several tables have to be updated (Inventory, Warehouse, Book, CD, Transportation tables, etc.). So we can write a stored procedure to regroup several SQL statements and improve performance. The stored procedure `sp_archive_books` defined in Listing 6-29 takes an archive date and a warehouse code as parameters and updates the `T_Inventory` and the `T_Transport` tables.

Listing 6-29. Abstract of a Stored Procedure Archiving Books

```
CREATE PROCEDURE sp_archive_books @archiveDate DATE, @warehouseCode VARCHAR AS
  UPDATE T_Inventory
  SET Number_Of_Books_Left - 1
  WHERE Archive_Date < @archiveDate AND Warehouse_Code = @warehouseCode;

  UPDATE T_Transport
  SET Warehouse_To_Take_Books_From = @warehouseCode;
END
```

The stored procedure in Listing 6-29 is compiled into the database and can then be invoked through its name (`sp_archive_books`). As you can see, a stored procedure accepts data in the form of input or output parameters. Input parameters (`@archiveDate` and `@warehouseCode` in our example) are utilized in the execution of the stored procedure which, in turn, can produce some output result. This result is returned to the application through the use of a result set.

In JPA 2.1 the `StoredProcedureQuery` interface (which extends `Query`) supports stored procedures. Unlike dynamic, named, or native queries, the API only allows you to invoke a stored procedure that already exists in the database, not define it. You can invoke a stored procedure with annotations (with `@NamedStoredProcedureQuery`) or dynamically.

Listing 6-30 shows the Book entity that declares the `sp_archive_books` stored procedure using named query annotations. The `NamedStoredProcedureQuery` annotation specifies name of the stored procedure to invoke the types of all parameters (`Date.class` and `String.class`), their corresponding parameter modes (`IN`, `OUT`, `INOUT`, `REF_CURSOR`), and how result sets, if any, are to be mapped. A `StoredProcedureParameter` annotation needs to be provided for each parameter.

Listing 6-30. Entity Declaring a Named Stored Procedure

```
@Entity
@NamedStoredProcedureQuery(name = "archiveOldBooks", procedureName = "sp_archive_books",
  parameters = {
    @StoredProcedureParameter(name = "archiveDate", mode = IN, type = Date.class),
    @StoredProcedureParameter(name = "warehouse", mode = IN, type = String.class)
  }
)
```

```
public class Book {

  @Id @GeneratedValue
  private Long id;
  private String title;
  private Float price;
  private String description;
  private String isbn;
  private String editor;
  private Integer nbOfPage;
  private Boolean illustrations;

  // Constructors, getters, setters
}
```

To invoke the sp_archive_books stored procedure, you need to use the entity manager and create a named stored procedure query by passing its name (archiveOldBooks). This returns a StoredProcedureQuery on which you can set the parameters and execute it as shown in Listing 6-31.

Listing 6-31. Calling a StoredProcedureQuery

```
StoredProcedureQuery query = em.createNamedStoredProcedureQuery("archiveOldBooks");
query.setParameter("archiveDate", new Date());
query.setParameter("maxBookArchived", 1000);
query.execute();
```

If the stored procedure is not defined using metadata (@NamedStoredProcedureQuery), you can use the API to dynamically specify a stored procedure query. This means that parameters and result set information must be provided programmatically. This can be done using the registerStoredProcedureParameter method of the StoredProcedureQuery interface as shown in Listing 6-32.

Listing 6-32. Registering and Calling a StoredProcedureQuery

```
StoredProcedureQuery query = em.createStoredProcedureQuery("sp_archive_old_books");
query.registerStoredProcedureParameter("archiveDate", Date.class, ParameterMode.IN);
query.registerStoredProcedureParameter("maxBookArchived", Integer.class, ParameterMode.IN);

query.setParameter("archiveDate", new Date());
query.setParameter("maxBookArchived", 1000);
query.execute();
```

Cache API

Most specifications (not just Java EE) focus heavily on functional requirements, leaving nonfunctional ones like performance, scalability, or clustering as implementation details. Implementations have to strictly follow the specification but may also add specific features. A perfect example for JPA would be caching.

Until JPA 2.0, caching wasn't mentioned in the specification. The entity manager is a first-level cache used to process data comprehensively for the database and to cache short-lived entities. This first-level cache is used on a per-transaction basis to reduce the number of SQL queries within a given transaction. For example, if an object is modified several times within the same transaction, the entity manager will generate only one UPDATE statement at the end of the transaction. A first-level cache is not a performance cache.