APPENDIX

D

EJB Query Language

The EJB Query Language (EJB QL) first appeared as part of the EJB 2.0 specification. This has been enhanced and extended in the EJB 3.0 specification and is now called the Java Persistence Query Language. The name change is part of an effort to separate the Persistence part of the specification from the rest of EJB. This has been done because the new Java Persistence API can stand alone from EJB and even from Java Enterprise Edition containers.

JAVA PERSISTENCE QUERY LANGUAGE AND THE SPEC

At the time this book went to press, the EJB 3.0 specification was still in the *proposed* final draft stage in the development process. At that time, the language for queries was still referred to as the EJB Query Language (EJB QL).

EJB 3.0 *entities* are significantly different from EJB 2.x *entity beans*, and thusin the final release of the specification a distinction was made by renaming the associated query language the Java Persistence Query Language.

Since the appendices to the book are being presented in electronic format, we were able to reflect the latest developments in the final release of the specification.

For those familiar with the Structured Query Language (SQL), a lot of what you see in this appendix will look familiar. There are some fundamental differences, however. First, the Persistence query language is truly a standard. While there are SQL standards, every Relational Database Management System (RDBMS) vendor has added proprietary extensions. This had made it challenging to migrate between RDBMS platforms. Another major difference between the two is that SQL operates on tables and fields, whereas the Java Persistence Query

Language operates directly on objects and their properties. Visually, this difference is subtle, but in practice, it completes the picture for EJB's new persistence layer; namely the emphasis on working with objects and not directly with the database.

This appendix draws on the models and code examples from Chapter 9 in order to explore the query language. At its most basic level, selects (queries), updates, and deletes are covered. You will take a close look into the very complex interactions that can be performed with these basic functions.

Although the specification does not indicate a convenient mnemonic (like EJB-QL), this appendix refers to the Java Persistence query language as the JPQL.

Query Types and Naming

Before jumping into examples and the language constructs, here are a few formal definitions.

Every expression in a JPQL query as a type. The type is derived from:

* The *abstract schema types* of variable declarations
* The types that persistent fields evaluate to
* The types that relationships evaluate to
* The structure of the expression
* The types of literals

The abstract schema type is derived from a combination of the entity class definition and the associated annotations. This includes every exposed *persistent-field*, referred to as a state-field and every persistent relationship field referred to as an *association-field*.

A JPQL query can involve all of the abstract schema types of all the entities belonging to the same persistence unit. This is referred to as the query's *domain*.

The association-fields defined in an entity will limit the domain of a query. This is because other entities can only be reached through these association fields. This is referred to as the *navigability* of an entity.

Entities are referenced in queries by their entity names as specified by the `@Entity` annotation (or by the entity class name, which is the default behavior if the name attribute is not used in the annotation). Here is a simple sample JPQL query from Chapter 9:

```
SELECT r FROM RoadVehicleSingle r
```

Source 9.6 shows an entity class, `RoadVehicle`, with the entity name `RoadVehicleSingle`. The abstract schema type is `RoadVehicleSingle` because of the entity name.

The required minimum clauses are present in the above example. Namely, the `SELECT` and `FROM` keywords (case is not important), the entity name, and an alias (r in this case). In the next section, we will look into select statements in more detail.

Query Structure

In this section, we will look at the various clauses, required and optional, that can appear in queries. We will also look at bulk update and delete operations.

Select Statements

The `SELECT` statement is made up of a number of clauses. As mentioned in the previous section, the `SELECT` and `FROM` clauses are required. Other clauses include: `WHERE`, `GROUP BY`, `HAVING`, and `ORDER BY`. As you will see further on, the `WHERE` clause is also used in updates and deletes. Let's take a look at each of these clauses in turn.

* The `SELECT` clause determines the types that will be selected.
* The `FROM` clause specifies the entities for which the other clauses will apply.
* The `WHERE` clause is used to restrict the results returned from the query. This clause is optional.
* The `GROUP BY` clause is used to results to be groups together. This clause is optional.
* The `HAVING` clause is used in conjunction with the `GROUP BY` clause to filter grouped results. This clause is optional.
* The `ORDER BY` clause is used to order the results. This clause is optional.

Let's take a deeper look at the FROM clause, since it is so critical to determining the domain of a query.

FROM in Depth

Identification variables are used in the `FROM` clause as placeholders for entities to be included in the query domain. These identification variables have no limitations on length but must conform to the Java rules for identifiers.

All identification variables are declared on the `FROM` clause and cannot be declared in any of the other clauses. An identification variable cannot be a reserved word or the name of any entity in the persistence unit and are case insensitive.

The elements in the `FROM` clause will each have a value based on the type of the expression used to identify the identification variable. In Chapter 9, you saw the following query:

```
SELECT c FROM CompanyOMUni c JOIN c.employees e
```

The keyword `INNER` before the word `JOIN` is optional. The identification variable `e` in the example above can have a value of any `Employee` that is directly reachable (navigable) from `CompanyOMUni` (because of the `JOIN`, which we will discuss below). The `employees` field is a collection of instances of the abstract schema type `Employee`, and the variable `e` will refer to a particular element of this collection.

Identification variables are processed from left to right. It is important to note that an identification variable can make use of results from previous variable declarations.

An identification variable can be followed by the navigation operator (`.`), followed by a field of the entity. This is called a path expression. In the above example, `c.employees` is a path expression.

One of the fundamental features of the JPQL is the ability to perform *joins*. We have seen some examples of this, but in the next subsection we will look at joins more explicitly.

Joins

In the JPQL, we have three join types at our disposal: *inner*, *outer*, and *fetch*. An inner join, often referred to as the relationship join, is most commonly used to retrieve results associated by foreign key relationships.

With an inner join, if the entity being joined to does not have a value, the result will not be included in the result set of entities. Recall Figure 9.4. We had a Company entity related to the Employee entity in a one-to-many relationship. That is, in a given company, there can be many employees. Let's suppose that in our database we have the arrangement as shown in Tables D.1 and D.2

ID	NAME
1	M*Power Internet Service, Inc.
2	Sun Microsystems
3	Bob's Bait and Tackle

Table D.1 Company

ID	NAME	COMPANY_ID
1	Micah Silverman	1
2	Tes Silverman	1
3	Rima Patel	2

4	James Gosling	3

Table D.2 Employee

The following query would return the three Company entities.

```
SELECT DISTINCT c FROM CompanyOMBid c
```

However, the following query will only return two Company entities.

```
SELECT DISTINCT c FROM CompanyOMBid c JOIN c.employees
```

The inner join causes only companies for which employees exist to be returned. Since Bob's Bait and Tackle does not have any employees associated with it, it is not included in the collection of results.

Left outer joins behave in just the opposite way from inner joins. That is, they allow entities to be retrieved when the matching join condition is absent. If we change the above SQL to the following, all three companies are retrieved once again:

```
SELECT DISTINCT c FROM CompanyOMBid c LEFT JOIN c.employees
```

The keyword `OUTER`, which would appear in between the words `LEFT` and `JOIN`, is optional.

A `FETCH` join allows for the fetching of associated entities as a side effect of a query. Since we are talking only about a side effect, associated entities cannot be referenced as an identification variable. Also, the association on the right side of the `FETCH` join must belong to an entity that is returned from the query. In the following query, the collection of employees associated with a company will be fetched as a side effect.

```
SELECT DISTINCT c FROM CompanyOMBid c JOIN FETCH c.employees
```

Performing a fetch join supersedes `FetchType.LAZY` attributes declared as part of a relationship annotation. That is, if lazy fetching was defined for an associated entity but a query is performed that specifies a fetch join, the associated entity will be populated as if `FetchType.EAGER` has been set.

The `IN` keyword is used to specify collection member declarations. The parameter to `IN` is a collection-valued path expression. The following query will result in a single Company entity being returned. Note the use of `IN` in the query. The identification variable `e` designates a single Employee in the collection of employees.

```
SELECT DISTINCT c
FROM CompanyOMBid c, IN(c.employees) e
WHERE e.name='Micah Silverman'
```

We are going to wrap up this section on the FROM clause with two important notes. First, the FROM clause in a JPQL query is treated in much the same way as a SQL FROM clause. Namely, identification variables impact the results of a query even without the presence of a WHERE clause. The second note is on polymorphic behavior. The FROM clause automatically designates all subclasses of the explicitly declared entity. The entities returned from the query will include subclass instances that satisfy the query.

The WHERE Clause

The WHERE clause is used to limit the result from a SELECT statement or to limit the scope of an UPDATE or DELETE statement. A conditional expression follows the WHERE keyword. Conditional expressions are also used in the HAVING clause, described below.

Conditional Expressions

In this subsection we will examine conditional expressions in depth. These language constructs are used in the WHERE or HAVING clauses.

Literals

String literals are enclosed in single quotation marks. A string literal that contains a single quotation mark must be identified by two single quotation marks. Here is an example of the string literal, don't:

```
'don''t'
```

It should be noted that string literals cannot be escaped in the same way as special characters in Java String literals.

Numeric and floating point literals conform to the same syntax as Java integer and floating point literals, respectively.

Enum literals conform to the Java syntax for enum literals. The enum class must be specified as part of the expression.

Boolean literals are TRUE and FALSE.

Identification Variables

Any identification variable appearing in the WHERE clause must have been defined in the FROM clause. Remember, identification variables represent a member of a collection and do not ever designate the collection in it is entirety.

Input Parameters

Input parameters can be referenced by position or can be named parameters. These should not be mixed in the same query. Note that input parameters can only appear in the WHERE or HAVING clause of a query.

Positional parameters are indicated by a question mark (?) followed by a number indicating the position of the input parameter. Input parameters start at the number 1. The same parameter can be used more than once in a query string. Also, the order of the parameters in the query string does not have to be sequential. Here's an example:

```
Query q =
  em.createQuery("SELECT c FROM CompanyOMBid c WHERE c.name = ?1").
    setParameter(1,"M*Power Internet Services, Inc.");
```

Named parameters are indicated by a colon (:) followed by a name. Named parameters are case sensitive. Here is an example:

```
Query q =
  em.createQuery("SELECT c FROM CompanyOMBid c WHERE c.name = :cname").
    setParameter("cname","M*Power Internet Services, Inc.");
```

Composition

Conditional expressions are composed of the following:

* Other conditional expressions
* Comparison operations
* Logical operations
* Boolean values resulting from path expressions
* Boolean literals
* Boolean input parameters

Arithmetic operations can be used in comparison operations. Arithmetic operations are composed of the following

* Other arithmetic operations
* Numeric values resulting from path expressions
* Numeric literals
* Numeric input parameters

Parentheses ( ) can be used in the standard way for expression evaluation.

The following is a representation of conditional expressions in Backus Naur Form (BNF):

All BNF syntax is taken from the final release of the JSR 220: Enterprise JavaBeans, version 3.0, Java Persistence API.

```
conditional_expression ::= conditional_term | conditional_expression OR
conditional_term
conditional_term ::= conditional_factor | conditional_term AND
conditional_factor
conditional_factor ::= [ NOT ] conditional_primary
```

```
conditional_primary ::= simple_cond_expression |
(conditional_expression)
simple_cond_expression ::=
     comparison_expression |
     between_expression |
     like_expression |
     in_expression |
     null_comparison_expression |
     empty_collection_comparison_expression |
     collection_member_expression |
     exists_expression
```

We will look at each of the types expressions below. First, a note on operator precedence.

Here are the operators in decreasing order of precedence:

* Navigation (`.`)
* Arithmetic:
  * `+`,`-` unary
  * `*`,`/`
  * `+`,`-` add, subtract
* Comparison: `=`, `>`, `>=`, `<`, `<=`, `<>` (not equal), `[NOT] BETWEEN`, `[NOT] LIKE`, `[NOT] IN`, `IS [NOT] NULL`, `IS [NOT] EMPTY`, `[NOT] MEMBER OF`,
* Logical:
  * `NOT`
  * `AND`
  * `OR`

Between Expressions

Here is the BNF syntax for between expressions:

```
arithmetic_expression
     [NOT] BETWEEN
arithmetic_expression
     AND
arithmetic_expression |
string_expression
     [NOT] BETWEEN
string_expression
     AND
string_expression |
datetime_expression
     [NOT] BETWEEN
datetime_expression
     AND
datetime_expression
```

Note that:

```
n BETWEEN m AND o
```

is equivalent to:

```
m <= n AND n <= o
```

Here is an example of BETWEEN logic drawing on the examples based on Figure 9.1 in Chapter 9:

```
SELECT r FROM RoadVehicleSingle r WHERE r.numPassengers between 4 AND 5
```

IN Expressions

Here is the BNF syntax for `[NOT] IN` expressions:

```
in_expression ::=
     state_field_path_expression [NOT] IN
       ( in_item {, in_item}* | subquery)
in_item ::= literal | input_parameter
```

The `state_field_path_expression` has to evaluate to an enum, numeric, or string value.

The type `in_item` value must be the same type or a type that is compatible with the `state_field_path_expression` type. Likewise, the results from a subquery must be the same type or a type that is compatible with the `state_field_path_expression` type. Subqueries are discussed below.

Here is an example of `IN` logic drawing on the examples based on Figure 9.1 in Chapter 9:

```
SELECT r FROM RoadVehicleSingle r WHERE r.numPassengers IN(2,5)
```

LIKE Expressions

Here is the BNF syntax for `[NOT] LIKE` expressions:

```
string_expression [NOT] LIKE pattern_value [ESCAPE escape_character]
```

The `pattern_value` must be a string literal or input parameter with a string value. The underscore (_) is a special character that stands for any single character. The percent sign (%) is a special character that stands for any sequence of characters. The optional `ESCAPE escape_character` clause is used to indicate an escape character or input parameter with character value. This is used to suppress the special meaning of the underscore and percent sign in the `pattern_value`.

Here is an example of `LIKE` logic drawing on the examples based on Figure 9.1 in Chapter 9:

```
SELECT r FROM RoadVehicleSingle r WHERE r.make LIKE 'M%'
```

NULL and Empty Collection Comparisons

Here is the BNF syntax for `IS [NOT] NULL` expressions:

```
{single_valued_path_expression | input_parameter } IS [NOT] NULL
```

A `NULL` comparison returns a collection of values indicating whether or not those values resolve to `NULL`.

Here is an example of `NULL` logic drawing on the examples based on Figure 9.1 in Chapter 9:

```
SELECT r FROM RoadVehicleSingle r WHERE r.model IS NOT NULL
```

Here is the BNF syntax for `IS [NOT] EMPTY` expressions:

```
collection_valued_path_expression IS [NOT] EMPTY
```

The expression is a test that indicates whether or not the referenced collection is empty or not.

Here is an example of `EMPTY` logic drawing on the examples based on Figure 9.4 in Chapter 9:

```
SELECT c FROM CompanyOMBid c WHERE c.employees IS NOT EMPTY
```

Collection Member Expressions

Here is the BNF syntax for `MEMBER [OF]` expressions:

```
entity_expression [NOT] MEMBER [OF] collection_valued_path_expression
entity_expression ::=
     single_valued_association_path_expression |
     simple_entity_expression
simple_entity_expression ::=
identification_variable | input_parameter
```

The expression determines whether or not the value of `entity_expression` is a member of the collection returned from `collection_valued_path_expression`.

Here is an example of `MEMBER OF` logic drawing on examples based on Figure 9.4 in Chapter 9:

```
"SELECT e FROM EmployeeOMBid e, CompanyOMBid c
   WHERE e MEMBER OF c.employees"
```

EXISTS, ALL, or ANY Expressions

Here is the BNF syntax for `EXISTS` expressions:

```
exists_expression::= [NOT] EXISTS (subquery)
```

An EXISTS expression is true only if the specified subquery returns at least one result. We will look at subqueries below.

Here is the BNF syntax for ALL or ANY expressions:

```
all_or_any_expression ::= { ALL | ANY | SOME} (subquery)
```

Any or all comparisons involve the result of a subquery. If ALL is specified, then every single result of the subquery must be true for the comparison. If ANY or SOME is specified, then at least one of the results in the subquery must be true for the comparison. Valid comparison operators are =, <, <=, >, >=, <>. We will look at subqueries below.

Subqueries

Subqueries can be used in the WHERE and HAVING clauses. In the interest of completeness, we will show the BNF syntax for subqueries, but it is basically everything that we have discussed thus far within parentheses. Here is the syntax:

```
subquery ::=
     simple_select_clause subquery_from_clause
             [where_clause] [groupby_clause] [having_clause]
simple_select_clause ::= SELECT [DISTINCT] simple_select_expression
subquery_from_clause ::=
     FROM subselect_identification_variable_declaration
     {, subselect_identification_variable_declaration}*
subselect_identification_variable_declaration ::=
     identification_variable_declaration |
     association_path_expression [AS] identification_variable |
     collection_member_declaration
simple_select_expression::=
     single_valued_path_expression |
     aggregate_expression |
     identification_variable
```

Here is an example of a subquery based on Figure 9.4 from Chapter 9:

```
SELECT c FROM CompanyOMBid c WHERE
     (SELECT COUNT(e) FROM c.employees e) = 0
```

It should be noted that in a future release of the specification, subqueries may be supported in the FROM clause. For this release, subqueries are not supported in the FROM clause.

Functional Expressions

The JPQL has a number of built-in functions that can be used in the WHERE and HAVING clauses. This section examines each of these functions.

String Functions

Here is the BNF syntax for the string functions:

```
functions_returning_strings ::=
     CONCAT(string_primary, string_primary) |
     SUBSTRING(string_primary,
            simple_arithmetic_expression,
            simple_arithmetic_expression) |
     TRIM([[trim_specification] [trim_character] FROM]
            string_primary) |
     LOWER(string_primary) |
     UPPER(string_primary)
trim_specification ::= LEADING | TRAILING | BOTH

functions_returning_numerics::=
     LENGTH(string_primary) |
     LOCATE(string_primary, string_primary[,
            simple_arithmetic_expression])
```

`CONCAT` returns a string that is a combination of the two strings passed in as parameters, one after the other.

`SUBSTRING` takes two numeric arguments indicating the start position and length (respectively) of the string to be returned. The substring is taken from the string passed in as the first argument.

`TRIM` will remove spaces from the beginning and the end of the specified string by default. If a character other than space is to be trimmed, it can be indicated through `trim_character`. The keyword `LEADING` can be used to specify that the trim should be done from the beginning of the string. `TRAILING` can be used to specify that the trim should be done from the end of the string. `BOTH` trims both ends and is the default behavior.

`LOWER` will convert the supplied string to all lowercase.

`UPPER` will convert the supplied string to all uppercase.

`LENGTH` returns the length of the supplied string.

`LOCATE` will search for a string (the first parameter) within a string (the second parameter) and return its position. If its position is not found, 0 is returned. The first position in string is 1. An optional third parameter specifies the starting position in the string to be searched. If it is not supplied, then the starting position is 1.

Arithmetic Functions

Here is the BNF syntax for the arithmetic functions:

```
functions_returning_numerics::=
     ABS(simple_arithmetic_expression) |
     SQRT(simple_arithmetic_expression) |
     MOD(simple_arithmetic_expression, simple_arithmetic_expression) |
     SIZE(collection_valued_path_expression)
```

`ABS` returns the absolute value of the supplied arithmetic expression.
`SQRT` returns the square root of the supplied arithmetic expression.
`MOD` returns the modulo value (the remainder) of the two supplied parameters.
`SIZE` returns the number of elements in the supplied collection.

Datetime Functions

Here is the BNF syntax for the `datetime` functions:

```
functions_returning_datetime:=
     CURRENT_DATE |
     CURRENT_TIME |
     CURRENT_TIMESTAMP
```

These functions return the date, time, and timestamp values (respectively) from the underlying database.

GROUP BY and HAVING Clauses

The `GROUP BY` clause enables grouping functions based on set of properties. The `HAVING` clause allows further restriction of this grouping by placing conditions on the `GROUP BY` clause.

If both a `WHERE` clause and `GROUP BY` clause are present in the query, the `WHERE` clause will be applied first, followed by the forming of the groups according to the `GROUP BY` clause.

When using `GROUP BY`, the structure of the `SELECT` clause follows the rules from SQL. Namely, any item appearing in the `SELECT` clause must also be specified in the `GROUP BY` clause. The only ~~exception to this rule are~~exception to this rule is expressions in the `SELECT` clause that are aggregate functions.

The `HAVING` clause specifies search conditions over the grouped items (or aggregate functions that apply to the grouped items).

It should be noted that while with SQL it is allowable to have a `HAVING` clause without a `GROUP BY` clause (implying that all the elements of the `SELECT` clause are aggregate functions), portable Java Persistence API applications should not rely on this. Implementers of the specification are not required to support a `HAVING` clause without the presence of a `GROUP BY` clause.

More on SELECT

We opened our discussion of the JPQL with `SELECT`. In this section, we will present the formal syntax for the `SELECT` clause. Here is the BNF syntax for the `SELECT` clause:

```
select_clause ::= SELECT [DISTINCT] select_expression
     {, select_expression}*
select_expression ::=
     single_valued_path_expression |
```

```
     aggregate_expression |
     identification_variable |
     OBJECT(identification_variable) |
     constructor_expression
constructor_expression ::=
     NEW constructor_name ( constructor_item {, constructor_item}* )
constructor_item ::= single_valued_path_expression |
     aggregate_expression
aggregate_expression ::=
     { AVG | MAX | MIN | SUM }
     ([DISTINCT] state_field_path_expression) |
     COUNT ([DISTINCT] identification_variable |
     state_field_path_expression |
single_valued_association_path_expression)
```

It is interesting to note the capability of returning the result into a newly constructed Object using the `NEW` keyword of the `SELECT` clause. Take a look at the following example:

```
SELECT NEW EmpInfo(e.name,e.sex) FROM EmployeeOMBid e
```

This presumes the existence of a Java class named `EmpInfo` with a constructor that takes two parameters. The types of the parameters from the `SELECT` clause must match the signature defined in the class.

A `SELECT` clause can have the following return types:

* An expression that represents a state field results in an object of the same type as that state field in the referenced entity.
* An expression that represents an association results in an object of the type of the relationship as defined in the entity and the object relational mapping.
* An identification variable results in the object of the referenced entity
* The result of an aggregate expression (discussed below) is determined by the aggregate expressions used.
* The result type of a constructor expression is an object of the class for which the constructor is defined.

When multiple select expressions are present, the result is an object array (`Object[]`) with elements in the same order as specified in the `SELECT` clause.

ORDER BY Clause

The `ORDER BY` clause allows the returned values to be ordered in a certain way. Here is the BNF syntax for `ORDER BY`:

```
orderby_clause ::= ORDER BY orderby_item {, orderby_item}*
orderby_item ::= state_field_path_expression [ASC | DESC]
```

The `orderby_item` must appear in the `SELECT` clause or else the query will be invalid.

The sequential order of the `orderby_item` elements determines the precedence of the ordering (from left to right).

`ASC` indicates that the order should be ascending.

`DESC` indicates that the order should be descending.

Bulk Updates and Deletes

Bulk update and delete operations are performed over a single entity and, potentially, its subclasses. Here is the BNF syntax:

```
update_statement ::= update_clause [where_clause]
update_clause ::= UPDATE abstract_schema_name
     [[AS] identification_variable] SET update_item {, update_item}*
update_item ::= [identification_variable.]{state_field |
     single_valued_association_field} = new_value
new_value ::= simple_arithmetic_expression |
     string_primary |
     datetime_primary |
     boolean_primary |
     enum_primary
     simple_entity_expression |
     NULL
delete_statement ::= delete_clause [where_clause]
delete_clause ::= DELETE FROM abstract_schema_name [[AS]
     identification_variable]
```

The syntax of the `WHERE` clause is as described above.

A `DELETE` operation applies only to the specified entity (and subclasses). It will not cascade to related entities.

New values specified as part of an update must be compatible with the fields of the entity being updated.

Bulk updates result in a direct corresponding database operation that bypasses optimistic locking checks. To achieve 100% portability, applications should update a version field and/or validate this field to ensure data integrity.

The persistence context is not synchronized when performing a bulk update or delete operation.

Bulk update and delete operations have the potential to create inconsistencies between entities and the database. To mitigate this, these operations should be performed in a separate transaction or at the beginning of a transaction before entities have been synchronized with the persistence context.

Summary

This appendix provided a thorough look at the newly named Java Persistence Query Language. You saw that it looks and behaves much like SQL but is completely database platform–independent and operates on entities rather than tables.

While the JPQL continues to be a part of the EJB specification, it can stand on its own. Vendors are starting to write out-of-container persistence managers that take advantage of the JPQL.

For more information, refer to the JSR 220: “Enterprise JavaBeans, Version 3.0, Java Persistence API.”