The Java Persistence Query Language

The Java Persistence query language defines queries for entities and their persistent state.

The query language allows you to write portable queries that work regardless of the underlying data store.

The query language uses the abstract persistence schemas of entities, including their relationships, for its data model and defines operators and expressions based on this data model.

The scope of a query spans the abstract schemas of related entities that are packaged in the same persistence unit.

The query language uses an SQL-like syntax to select objects or values based on entity abstract schema types and relationships among them.

This chapter relies on the material presented in earlier chapters.

For conceptual information, see Chapter 32, Introduction to the Java Persistence API.

For code examples, see Chapter 33, Running the Persistence Examples.

The following topics are addressed here:

- Query Language Terminology
- Creating Queries Using the Java Persistence Query Language
- Simplified Query Language Syntax
- Example Queries
- Full Query Language Syntax

Query Language Terminology

The following list defines some of the terms referred to in this chapter:

- **Abstract schema: The persistent schema abstraction (persistent entities, their state, and their relationships) over which queries operate.**

The query language translates queries over this persistent schema abstraction into queries that are executed over the database schema to which entities are mapped.

- **Abstract schema type:** The type to which the persistent property of an entity evaluates in the abstract schema.

That is, each persistent field or property in an entity has a corresponding state field of the same type in the abstract schema.

The abstract schema type of an entity is derived from the entity class and the metadata information provided by Java language annotations.

- **Backus-Naur Form (BNF):** A notation that describes the syntax of high-level languages.

  The syntax diagrams in this chapter are in BNF notation.

- **Navigation:** The traversal of relationships in a query language expression.

  The navigation operator is a period.

- **Path expression:** An expression that navigates to a entity's state or relationship field.

- **State field:** A persistent field of an entity.

- **Relationship field:** A persistent relationship field of an entity whose type is the abstract schema type of the related entity.

Creating Queries Using the Java Persistence Query Language

The `EntityManager.createQuery` and `EntityManager.createNamedQuery` methods are used to query the datastore by using Java Persistence query language queries.

The `createQuery` method is used to create dynamic queries, which are queries defined directly within an application's business logic:

```
public List findWithName
(String name){
return em.createQuery(
"SELECT c FROM Customer c WHERE
c.name LIKE :custName")
.setParameter("custName", name)
.setMaxResults(10)
.getResultList(); }
```

The `createNamedQuery` method is used to create static queries, or queries that are defined in metadata by using the `javax.persistence.NamedQuery` annotation.

The `name` element of `@NamedQuery` specifies the name of the query that will be used with the `createNamedQuery` method.

The `query` element of `@NamedQuery` is the `query`:

```
@NamedQuery(
name="findAllCustomersWithName",
query="SELECT c FROM Customer c
WHERE c.name LIKE :custName"
)
```

Here's an example of `createNamedQuery`, which uses the `@NamedQuery`:

```
@PersistenceContext
public EntityManager em;
...
customers = em.createNamedQuery
("findAllCustomersWithName")
.setParameter("custName", "Smith")
.getResultList();
```

Named Parameters in Queries

Named parameters are query parameters that are prefixed with a colon (:).

Named parameters in a query are bound to an argument by the following method:

```
javax.persistence.Query.setParameter
(String name, Object value)
```

In the following example, the `name` argument to the `findWithName` business method is bound to the `:custName` named parameter in the query by calling `Query.setParameter`:

```
public List findWithName
(String name){
return em.createQuery(
"SELECT c FROM Customer c WHERE
c.name LIKE :custName")
.setParameter("custName", name)
.getResultList();
}
```

Named parameters are case-sensitive and may be used by both dynamic and static queries.

Positional Parameters in Queries

You may use positional parameters instead of named parameters in queries.

Positional parameters are prefixed with a question mark (?) followed the numeric position of the parameter in the query.

The `Query.setParameter(integer position, Object value)` method is used to set the parameter values.

In the following example, the `findWithName` business method is rewritten to use input parameters:

```
public List findWithName
(String name){
```

```
return em.createQuery(
"SELECT c FROM Customer c WHERE
c.name LIKE ?1")
.setParameter(1, name)
.getResultList();
}
```

Input parameters are numbered starting from 1. Input parameters are case-sensitive, and may be used by both dynamic and static queries.

Simplified Query Language Syntax

This section briefly describes the syntax of the query language so that you can quickly move on to Example Queries.

When you are ready to learn about the syntax in more detail, see Full Query Language Syntax.

Select Statements

A select query has six clauses: `SELECT`, `FROM`, `WHERE`, `GROUP BY`, `HAVING`, and `ORDER BY`.

The `SELECT` and `FROM` clauses are required, but the `WHERE`, `GROUP BY`, `HAVING`, and `ORDER BY` clauses are optional.

Here is the high-level BNF syntax of a query language select query:

```
QL_statement ::=
select_clause from_clause
[where_clause][groupby_clause]
[having_clause][orderby_clause]
```

- The `SELECT` clause defines the types of the objects or values returned by the query.

- The `FROM` clause defines the scope of the query by declaring one or more identification variables, which can be referenced in the `SELECT` and `WHERE` clauses.

  An identification variable represents one of the following elements:

  - The abstract schema name of an entity

- An element of a collection relationship
- An element of a single-valued relationship
- A member of a collection that is the multiple side of a one-to-many relationship

- The `WHERE` clause is a conditional expression that restricts the objects or values retrieved by the query.

  Although the clause is optional, most queries have a `WHERE` clause.

- The `GROUP BY` clause groups query results according to a set of properties.

- The `HAVING` clause is used with the `GROUP BY` clause to further restrict the query results according to a conditional expression.

- The `ORDER BY` clause sorts the objects or values returned by the query into a specified order.

Update and Delete Statements

Update and delete statements provide bulk operations over sets of entities.

These statements have the following syntax:

```
update_statement :: =
update_clause [where_clause]
delete_statement :: =
delete_clause [where_clause]
```

The update and delete clauses determine the type of the entities to be updated or deleted.

The `WHERE` clause may be used to restrict the scope of the update or delete operation.

Example Queries

The following queries are from the `Player` entity of the `roster` application, which is documented in The `roster` Application.

Simple Queries

If you are unfamiliar with the query language, these simple queries are a good place to start.

A Basic Select Query

```
SELECT p
FROM Player p
```

- **Data retrieved: All players.**

- **Description:** The `FROM` clause declares an identification variable named `p`, omitting the optional keyword `AS`.

  If the `AS` keyword were included, the clause would be written as follows:

- ```
  FROM Player AS
   p
  ```
  ```

The `Player` element is the abstract schema name of the `Player` entity.

- See also: Identification Variables.

Eliminating Duplicate Values

```
SELECT DISTINCT
 p
FROM Player p
WHERE p.position = ?1
```

- **Data retrieved: The players with the position specified by the query's parameter.**

- **Description:** The `DISTINCT` keyword eliminates duplicate values.

  The `WHERE` clause restricts the players retrieved by checking their `position`, a persistent field of the `Player` entity.

  The `?1` element denotes the input parameter of the query.

- **See also:** Input Parameters and The `DISTINCT` Keyword.

Using Named Parameters

```
SELECT DISTINCT p
FROM Player p
WHERE p.position = :position AND
p.name = :name
```

- **Data retrieved:** **The players having the specified positions and names.**

- **Description:** The `position` and `name` elements are persistent fields of the `Player` entity.

  The `WHERE` clause compares the values of these fields with the named parameters of the query, set using the `Query.setNamedParameter` method.

- The query language denotes a named input parameter using a colon (:) followed by an identifier.

  The first input parameter is `:position`, the second is `:name`.

Queries That Navigate to Related Entities

In the query language, an expression can traverse, or navigate, to related entities.

These expressions are the primary difference between the Java Persistence query language and SQL.
Queries navigates to related entities, whereas SQL joins tables.

A Simple Query with Relationships

```
SELECT DISTINCT p
FROM Player p, IN(p.teams) t
```

- **Data retrieved:** All players who belong to a team.

- **Description**: The `FROM` clause declares two identification variables: `p` and `t`.

  The `p` variable represents the `Player` entity, and the `t` variable represents the related `Team` entity.

  The declaration for `t` references the previously declared `p` variable.

The `IN` keyword signifies that `teams` is a collection of related entities.

The `p.teams` expression navigates from a `Player` to its related `Team`.

The period in the `p.teams` expression is the navigation operator.

You may also use the JOIN statement to write the same query:

```
SELECT DISTINCT p
FROM Player p JOIN p.teams t
```

This query could also be rewritten as:

```
SELECT DISTINCT p
FROM Player p
WHERE p.team IS NOT EMPTY
```

Navigating to Single-Valued Relationship Fields

Use the `JOIN` clause statement to navigate to a single-valued relationship field:

```
SELECT t
FROM Team t JOIN t.league l
WHERE l.sport = 'soccer' OR
l.sport ='football'
```

In this example, the query will return all teams that are in either soccer or football leagues.

Traversing Relationships with an Input Parameter

```
SELECT DISTINCT p
FROM Player p, IN (p.teams) AS t
WHERE t.city = :city
```

- **Data retrieved:** The players whose teams belong to the specified city.

- **Description:** This query is similar to the previous example but adds an input parameter.

  The `AS` keyword in the `FROM` clause is optional.

  In the `WHERE` clause, the period preceding the persistent variable `city` is a delimiter, not a navigation operator.

Strictly speaking, expressions can navigate to relationship fields (related entities) but not to persistent fields.

To access a persistent field, an expression uses the period as a delimiter.

Expressions cannot navigate beyond (or further qualify) relationship fields that are collections.

In the syntax of an expression, a collection-valued field is a terminal symbol.

Because the `teams` field is a collection, the `WHERE` clause cannot specify `p.teams.city` (an illegal expression).

- See also: Path Expressions.

Traversing Multiple Relationships

```
SELECT DISTINCT p
FROM Player p, IN (p.teams) t
WHERE t.league = :league
```

- **Data retrieved:** The players who belong to the specified league.

- **Description:** The expressions in this query navigate over two relationships.

  The `p.teams` expression navigates the `Player-Team` relationship, and the `t.league` expression navigates the `Team-League` relationship.

In the other examples, the input parameters are `String` objects; in this example, the parameter is an object whose type is a `League`.

This type matches the `league` relationship field in the comparison expression of the `WHERE` clause.

Navigating According to Related Fields

```
SELECT DISTINCT p
FROM Player p, IN (p.teams) t
WHERE t.league.sport = :sport
```

- **Data retrieved: The players who participate in the specified sport.**

- **Description:** The `sport` persistent field belongs to the `League` entity.

  To reach the `sport` field, the query must first navigate from the `Player` entity to `Team` (`p.teams`) and then from `Team` to the `League` entity (`t.league`).

- **Because it is not a collection, the `league` relationship field can be followed by the `sport` persistent field.**

Queries with Other Conditional Expressions

Every **WHERE** clause must specify a conditional expression, of which there are several kinds.

In the previous examples, the conditional expressions are comparison expressions that test for equality.

The following examples demonstrate some of the other kinds of conditional expressions.

For descriptions of all conditional expressions, see `WHERE` Clause.

The `LIKE` Expression

```
SELECT p
 FROM Player p
 WHERE p.name LIKE 'Mich%'
```

- **Data retrieved: All players whose names begin with "Mich."**

- **Description:** The `LIKE` expression uses wildcard characters to search for strings that match the wildcard pattern.

  In this case, the query uses the `LIKE` expression and the `%` wildcard to find all players whose names begin with the string "Mich." For example, "Michael" and "Michelle" both match the wildcard pattern.
- **See also:** `LIKE` Expressions.

The `IS NULL` *Expression*

```
SELECT t
 FROM Team t
 WHERE t.league IS NULL
```

- **Data retrieved: All teams not associated with a league.**

- **Description:** The `IS NULL` expression can be used to check whether a relationship has been set between two entities.

  In this case, the query checks whether the teams are associated with any leagues and returns the teams that do not have a league.

- **See also:** `NULL` Comparison Expressions and NULL Values.

The `IS EMPTY` Expression

```
SELECT p
FROM Player p
WHERE p.teams IS EMPTY
```

- **Data retrieved: All players who do not belong to a team.**

- **Description:** The `teams` relationship field of the `Player` entity is a collection.

  If a player does not belong to a team, the `teams` collection is empty, and the conditional expression is `TRUE`.

- **See also:** Empty Collection Comparison Expressions.

The BETWEEN *Expression*

```
SELECT DISTINCT p
FROM Player p
WHERE p.salary
BETWEEN
:lowerSalary AND :higherSalary
```

- **Data retrieved: The players whose salaries fall within the range of the specified salaries.**

- **Description:** This **`BETWEEN`** expression has three arithmetic expressions: a persistent field (`p.salary`) and the two input parameters (`:lowerSalary` and `:higherSalary`).

  The following expression is equivalent to the **`BETWEEN`** expression:

```
p.salary >= :lowerSalary AND
p.salary <= :higherSalary
```

- **See also:** <u>**`BETWEEN`** Expressions</u>.

Comparison Operators

```
SELECT DISTINCT p1
FROM Player p1, Player p2
WHERE p1.salary > p2.salary AND
p2.name = :name
```

- **Data retrieved:** All players whose salaries are higher than the salary of the player with the specified name.

- **Description:** The `FROM` clause declares two identification variables (`p1` and `p2`) of the same type (`Player`).

  Two identification variables are needed because the `WHERE` clause compares the salary of one player (`p2`) with that of the other players (`p1`).

- **See also:** Identification Variables.

Bulk Updates and Deletes

The following examples show how to use the `UPDATE` and `DELETE` expressions in queries.

`UPDATE` and `DELETE` operate on multiple entities according to the condition or conditions set in the `WHERE` clause.

The `WHERE` clause in `UPDATE` and `DELETE` queries follows the same rules as `SELECT` queries.

Update Queries

```
UPDATE Player p
SET p.status = 'inactive'
WHERE
p.lastPlayed < :inactiveThresholdDate
```

- **Description:** This query sets the status of a set of players to `inactive` if the player's last game was longer than the date specified in `inactiveThresholdDate`.

Delete Queries

```
DELETE
FROM Player p
WHERE p.status = 'inactive'
AND p.teams IS EMPTY
```

- **Description:** This query deletes all inactive players who are not on a team.

Full Query Language Syntax

This section discusses the query language syntax, as defined in the Java Persistence API 2.0 specification available at http://jcp.org/en/jsr/detail?id=317.

Much of the following material paraphrases or directly quotes the specification.

BNF Symbols

Table 34-1 describes the BNF symbols used in this chapter.

Table 34-1 BNF Symbol Summary

Symbol	Description
::=	The element to the left of the symbol is defined by the constructs on the right.
*	The preceding construct may occur zero or more times.
{ . . . }	The constructs within the braces are grouped together.
[. . .]	The constructs within the brackets are optional.
|	An exclusive OR.
BOLDFACE	A keyword; although capitalized in the BNF diagram, keywords are not case-sensitive.
White space	A whitespace character can be a space, a horizontal tab, or a line feed.

BNF Grammar of the Java Persistence Query Language

Here is the entire BNF diagram for the query language:

```
QL_statement ::= select_statement |
update_statement | delete_statement
```

```
select_statement ::= select_clause
from_clause [where_clause]
[groupby_clause]
[having_clause] [orderby_clause]
update_statement ::= update_clause
[where_clause]
delete_statement ::= delete_clause
[where_clause]
```

```
from_clause ::=
FROM
identification_variable_declaration
{, {
identification_variable_declaration
| collection_member_declaration}}*
identification_variable_declaration
::=
range_variable_declaration
{ join | fetch_join }*
```

```
range_variable_declaration ::=
abstract_schema_name [AS]
identification_variable
join ::= join_spec
join_association_path_expression
[AS]
identification_variable
fetch_join ::= join_specFETCH
join_association_path_expression
```

```
association_path_expression ::=
collection_valued_path_expression |
 single_valued_association_path_
expression
join_spec::= [LEFT [OUTER] |INNER]
```

```
JOIN
join_association_path_expression
::=
join_collection_valued_path_
expression |
join_single_valued_association_path
_expression
```

```
join_collection_valued_path_
expression::=
identification_variable.collection_
valued_association_field
join_single_valued_association_path
_expression::=
identification_variable.single_
valued_association_field
```

```
collection_member_declaration ::=
IN
(collection_valued_path_expression)
[AS]
identification_variable
single_valued_path_expression ::=
state_field_path_expression |
single_valued_association_path_
expression
```

```
state_field_path_expression ::=
{identification_variable |
single_valued_association_path_
expression}.state_field
single_valued_association_path_
expression ::=
identification_variable.
{single_valued_association_field.}*
single_valued_association_field
```

```
collection_valued_path_expression
::=
identification_variable.
{single_valued_association_field.}*
collection_valued_association_field
state_field ::=
{embedded_class_state_field.}*
simple_state_field
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_clause ::= DELETE FROM
abstract_schema_name [[AS]
identification_variable]
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)
where_clause ::=
WHERE conditional_expression
groupby_clause ::= GROUP BY
groupby_item {, groupby_item}*
groupby_item ::=
single_valued_path_expression
having_clause ::=
HAVING conditional_expression
```

```
orderby_clause ::= ORDER BY
orderby_item {, orderby_item}*
orderby_item ::=
state_field_path_expression
[ASC |DESC]
subquery ::= simple_select_clause
subquery_from_clause
[where_clause] [groupby_clause]
[having_clause]
```

```
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_clause ::=
SELECT [DISTINCT]
simple_select_expression
```

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

```
like_expression ::=
string_expression [NOT]
LIKE pattern_value
[ESCAPE escape_character]
null_comparison_expression ::=
{single_valued_path_expression |
input_parameter} IS [NOT]
NULL
```

```
empty_collection_comparison_
expression ::=
collection_valued_path_
expression IS [NOT] EMPTY
collection_member_expression ::=
entity_expression
[NOT] MEMBER [OF]
collection_valued_path_expression
exists_expression::=
[NOT] EXISTS (subquery)
```

```
all_or_any_expression ::=
{ALL |ANY |SOME} (subquery)
comparison_expression ::=
string_expression
comparison_operator
{string_expression |
all_or_any_expression} |
boolean_expression {= |<> }
{boolean_expression |
all_or_any_expression} |
```

```
enum_expression {= |<> }
{enum_expression |
all_or_any_expression} |
datetime_expression
comparison_operator
{datetime_expression |
all_or_any_expression} |
entity_expression {= |<> }
{entity_expression |
all_or_any_expression} |
```

```
arithmetic_expression
comparison_operator
{arithmetic_expression |
all_or_any_expression}
comparison_operator ::=
= |> |>= |< |<= |<>
arithmetic_expression ::=
simple_arithmetic_expression |
(subquery)
```

```
simple_arithmetic_expression ::=
arithmetic_term |
simple_arithmetic_expression
{+ |- }
arithmetic_term
arithmetic_term ::=
arithmetic_factor | arithmetic_term
{* |/ }
arithmetic_factor
arithmetic_factor ::=
[{+ |- }] arithmetic_primary
```

```
arithmetic_primary ::=
state_field_path_expression |
numeric_literal |
(simple_arithmetic_expression) |
input_parameter |
functions_returning_numerics |
aggregate_expression
string_expression ::=
string_primary | (subquery)
```

```
string_primary ::=
state_field_path_expression |
string_literal | input_parameter |
functions_returning_strings |
aggregate_expression
datetime_expression ::=
datetime_primary | (subquery)
datetime_primary ::=
state_field_path_expression |
input_parameter |
functions_returning_datetime |
```

```
aggregate_expression
boolean_expression ::=
boolean_primary | (subquery)
boolean_primary ::=
state_field_path_expression |
boolean_literal | input_parameter
enum_expression ::=
enum_primary | (subquery)
enum_primary ::=
state_field_path_expression |
enum_literal | input_parameter
```

```
entity_expression ::=
single_valued_association_path_
expression |
simple_entity_expression
simple_entity_expression ::=
identification_variable |
input_parameter
functions_returning_numerics::=
LENGTH(string_primary) |
LOCATE(string_primary,
string_primary[,
```

```
simple_arithmetic_expression]) |
ABS(simple_arithmetic_expression) |
SQRT(simple_arithmetic_expression)|
MOD(simple_arithmetic_expression,
simple_arithmetic_expression) |
SIZE
(collection_valued_path_expression)
functions_returning_datetime ::=
CURRENT_DATE | CURRENT_TIME |
CURRENT_TIMESTAMP
```

```
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
```

FROM Clause

The FROM clause defines the domain of the query by declaring identification variables.

Identifiers

An identifier is a sequence of one or more characters.

The first character must be a valid first character (letter, $, _) in an identifier of the Java programming language, hereafter in this chapter called simply “Java”.

Each subsequent character in the sequence must be a valid nonfirst character (letter, digit, `$`, _) in a Java identifier.

(For details, see the Java SE API documentation of the `isJavaIdentifierStart` and `isJavaIdentifierPart` methods of the `Character` class.) The question mark (`?`) is a reserved character in the query language and cannot be used in an identifier.

A query language identifier is case-sensitive, with two exceptions:

- Keywords
- Identification variables

An identifier cannot be the same as a query language keyword.

Here is a list of query language keywords:

ABS	ALL	AND	ANY	AS
ASC	AVG	BETWEEN	BIT_LENGTH	BOTH
BY	CASE	CHAR_LENGTH	CHARACTER_LENGTH	CLASS
COALESCE	CONCAT	COUNT	CURRENT_DATE	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		

It is not recommended that you use an SQL keyword as an identifier, because the list of keywords may expand to include other reserved SQL words in the future.

Identification Variables

An **identification variable** is an identifier declared in the `FROM` clause.

Although they can reference identification variables, the `SELECT` and `WHERE` clauses cannot declare them.

All identification variables must be declared in the `FROM` clause.

Because it is an identifier, an identification variable has the same naming conventions and restrictions as an identifier, with the exception that an identification variables is case-insensitive.

For example, an identification variable cannot be the same as a query language keyword.

(See the preceding section for more naming rules.)

Also, within a given persistence unit, an identification variable name must not match the name of any entity or abstract schema.

The `FROM` clause can contain multiple declarations, separated by commas.

A declaration can reference another identification variable that has been previously declared (to the left).

In the following `FROM` clause, the variable `t` references the previously declared variable `p`:

```
FROM Player p, IN (p.teams) AS t
```

Even if it is not used in the WHERE clause, an identification variable's declaration can affect the results of the query.

For example, compare the next two queries.

The following query returns all players, whether or not they belong to a team:

```
SELECT p
FROM Player p
```

In contrast, because it declares the `t` identification variable, the next query fetches all players who belong to a team:

```
SELECT p
FROM Player p, IN (p.teams) AS t
```

The following query returns the same results as the preceding query, but the `WHERE` clause makes it easier to read:

```
SELECT p
FROM Player p
WHERE p.teams IS NOT EMPTY
```

An identification variable always designates a reference to a single value whose type is that of the expression used in the declaration.

There are two kinds of declarations: range variable and collection member.

Range Variable Declarations

To declare an identification variable as an abstract schema type, you specify a range variable declaration.

In other words, an identification variable can range over the abstract schema type of an entity.

In the following example, an identification variable named `p` represents the abstract schema named `Player`:

```
FROM Player p
```

A range variable declaration can include the optional `AS` operator:

```
FROM Player AS p
```

To obtain objects, a query usually uses path expressions to navigate through the relationships.

But for those objects that cannot be obtained by navigation, you can use a range variable declaration to designate a starting point, or root.

If the query compares multiple values of the same abstract schema type, the FROM clause must declare multiple identification variables for the abstract schema:

```
FROM Player p1, Player p2
```

For an example of such a query, see Comparison Operators.

Collection Member Declarations

In a one-to-many relationship, the multiple side consists of a collection of entities.

An identification variable can represent a member of this collection.

To access a collection member, the path expression in the variable's declaration navigates through the relationships in the abstract schema.

(For more information on path expressions, see Path Expressions.) Because a path expression can be based on another path expression, the navigation can traverse several relationships.

See Traversing Multiple Relationships.

A collection member declaration must include the `IN` operator but can omit the optional `AS` operator.

In the following example, the entity represented by the abstract schema named `Player` has a relationship field called `teams`.

The identification variable called `t` represents a single member of the `teams` collection.

```
FROM Player p,
IN (p.tea ms) t
```

Joins

The `JOIN` operator is used to traverse over relationships between entities and is functionally similar to the `IN` operator.

In the following example, the query joins over the relationship between customers and orders:

```
SELECT c
FROM Customer c JOIN c.orders o
WHERE
c.status = 1 AND
o.totalPrice > 10000
```

The `INNER` keyword is optional:

```
SELECT c
FROM Customer c
INNER JOIN c.orders o
WHERE
c.status = 1 AND
o.totalPrice > 10000
```

These examples are equivalent to the following query, which uses the `IN` operator:

```
SELECT c
FROM Customer c, IN(c.orders) o
WHERE
c.status = 1 AND
o.totalPrice > 10000
```

You can also join a single-valued relationship:

```
SELECT t
FROM Team t JOIN t.league l
WHERE l.sport = :sport
```

A `LEFT JOIN` or `LEFT OUTER JOIN` retrieves a set of entities where matching values in the join condition may be absent.

The `OUTER` keyword is optional.

```
SELECT c.name, o.totalPrice
FROM Order o LEFT JOIN o.customer c
```

A `FETCH JOIN` is a join operation that returns associated entities as a side effect of running the query.

In the following example, the query returns a set of departments and, as a side effect, the associated employees of the departments, even though the employees were not explicitly retrieved by the SELECT clause.

```
SELECT d
FROM Department d
LEFT JOIN FETCH d.employees
WHERE d.deptno = 1
```

Path Expressions

Path expressions are important constructs in the syntax of the query language, for several reasons.

First, path expressions define navigation paths through the relationships in the abstract schema.

These path definitions affect both the scope and the results of a query.

Second, path expressions can appear in any of the main clauses of a query (`SELECT`, `DELETE`, `HAVING`, `UPDATE`, `WHERE`, `FROM`, `GROUP BY`, `ORDER BY`).

Finally, although much of the query language is a subset of SQL, path expressions are extensions not found in SQL.

Examples of Path Expressions

Here, the `WHERE` clause contains a `single_valued_path_expression`; the `p` is an identification variable, and `salary` is a persistent field of `Player`:

```
SELECT DISTINCT p
FROM Player p
WHERE p.salary
BETWEEN
:lowerSalary AND :higherSalary
```

Here, the `WHERE` clause also contains a `single_valued_path_expression`; `t` is an identification variable, `league` is a single-valued relationship field, and `sport` is a persistent field of `league`:

```
SELECT DISTINCT p
FROM Player p, IN (p.teams) t
WHERE t.league.sport = :sport
```

Here, the `WHERE` clause contains a `collection_valued_path_expression`; `p` is an identification variable, and `teams` designates a collection-valued relationship field:

```
SELECT DISTINCT p
FROM Player p
WHERE p.teams IS EMPTY
```

Expression Types

The type of a path expression is the type of the object represented by the ending element, which can be one of the following:

- Persistent field
- Single-valued relationship field
- Collection-valued relationship field

For example, the type of the expression `p.salary` is `double` because the terminating persistent field (`salary`) is a `double`.

In the expression `p.teams`, the terminating element is a collection-valued relationship field (`teams`).

This expression's type is a collection of the abstract schema type named `Team`.

Because `Team` is the abstract schema name for the `Team` entity, this type maps to the entity.

For more information on the type mapping of abstract schemas, see Return Types.

Navigation

A path expression enables the query to navigate to related entities.

The terminating elements of an expression determine whether navigation is allowed.

If an expression contains a single-valued relationship field, the navigation can continue to an object that is related to the field.

However, an expression cannot navigate beyond a persistent field or a collection-valued relationship field.

For example, the expression `p.teams.league.sport` is illegal because `teams` is a collection-valued relationship field.

To reach the `sport` field, the `FROM` clause could define an identification variable named `t` for the `teams` field:

```
FROM Player AS p, IN (p.teams) t
WHERE t.league.sport = 'soccer'
```

WHERE Clause

The WHERE clause specifies a conditional expression that limits the values returned by the query.

The query returns all corresponding values in the data store for which the conditional expression is TRUE.

Although usually specified, the WHERE clause is optional.

If the WHERE clause is omitted, the query returns all values.

The high-level syntax for the WHERE clause follows:

```
where_clause ::=
WHERE conditional_expression
```

Literals

There are four kinds of literals: string, numeric, Boolean, and enum.

- **String literals:** A string literal is enclosed in single quotes:

```
'Duke'
```

If a string literal contains a single quote, you indicate the quote by using two single quotes:

```
'Duke''s'
```

Like a Java `String`, a string literal in the query language uses the Unicode character encoding.

- **Numeric literals**: There are two types of numeric literals: exact and approximate.

An exact numeric literal is a numeric value without a decimal point, such as 65, –233, and +12.

Using the Java integer syntax, exact numeric literals support numbers in the range of a Java `long`.

An approximate numeric literal is a numeric value in scientific notation, such as 57., –85.7, and +2.1.

Using the syntax of the Java floating-point literal, approximate numeric literals support numbers in the range of a Java `double`.

- **Boolean literals:** A Boolean literal is either `TRUE` or `FALSE`.

These keywords are not case-sensitive.

- **Enum literals:** The Java Persistence query language supports the use of enum literals using the Java enum literal syntax.

  The enum class name must be specified as a fully qualified class name:

```
SELECT e
FROM Employee e
WHERE e.status =
com.xyz.EmployeeStatus.FULL_TIME
```

Input Parameters

An input parameter can be either a named parameter or a positional parameter.

- A named input parameter is designated by a colon (:) followed by a string; for example, `:name`.

- A positional input parameter is designated by a question mark (`?`) followed by an integer.

  For example, the first input parameter is `?1`, the second is `?2`, and so forth.

The following rules apply to input parameters.

- They can be used only in a WHERE or HAVING clause.

- Positional parameters must be numbered, starting with the integer 1.

- Named parameters and positional parameters may not be mixed in a single query.

- Named parameters are case-sensitive.

Conditional Expressions

A `WHERE` clause consists of a conditional expression, which is evaluated from left to right within a precedence level.

You can change the order of evaluation by using parentheses.

Operators and Their Precedence

Table 34-2 lists the query language operators in order of decreasing precedence.

Table 34-2 Query Language Order Precedence

Type	Precedence Order
Navigation	. (a period)
Arithmetic	+ – (unary) * / (multiplication and division) + – (addition and subtraction)

Comparison	=
	>
	>=
	<
	<=
	<> (not equal)
	`[NOT] BETWEEN`
	`[NOT] LIKE`
	`[NOT] IN`
	`IS [NOT] NULL`
	`IS [NOT] EMPTY`
	`[NOT] MEMBER OF`

Logical	`NOT`
	
	`AND`
	
	`OR`

BETWEEN Expressions

A **BETWEEN** expression determines whether an arithmetic expression falls within a range of values.

These two expressions are equivalent:

```
p.age BETWEEN 15 AND 19
p.age >= 15 AND p.age <= 19
```

The following two expressions also are equivalent:

```
p.age NOT BETWEEN 15 AND 19
p.age < 15 OR p.age > 19
```

If an arithmetic expression has a **NULL** value, the value of the **BETWEEN** expression is unknown.

`IN` *Expressions*

An `IN` expression determines whether a string belongs to a set of string literals or whether a number belongs to a set of number values.

The path expression must have a string or numeric value.

If the path expression has a `NULL` value, the value of the `IN` expression is unknown.

In the following example, the expression is `TRUE` if the country is `UK` , but `FALSE` if the country is `Peru`.

```
o.country IN ('UK', 'US', 'France')
```

You may also use input parameters:

```
o.country IN
('UK', 'US', 'France', :country)
```

`LIKE` *Expressions*

A **`LIKE`** expression determines whether a wildcard pattern matches a string.

The path expression must have a string or numeric value.

If this value is `NULL`, the value of the `LIKE` expression is unknown.

The pattern value is a string literal that can contain wildcard characters.

The underscore (_) wildcard character represents any single character.

The percent (`%`) wildcard character represents zero or more characters.

The `ESCAPE` clause specifies an escape character for the wildcard characters in the pattern value.

Table 34-3 shows some sample `LIKE` expressions.

Table 34-3 `LIKE` Expression Examples

Expression	TRUE	FALSE
`address.phone LIKE '12%3'`	`'123'` `'12993'`	`'1234'`
`asentence.word LIKE 'l_se'`	`'lose'`	`'loose'`
`aword.underscored LIKE '\_%' ESCAPE '\'`	`'_foo'`	`'bar'`
`address.phone NOT LIKE '12%3'`	`'1234'`	`'123'` `'12993'`

`NULL` *Comparison Expressions*

A **`NULL`** comparison expression tests whether a single-valued path expression or an input parameter has a **`NULL`** value.

Usually, the **`NULL`** comparison expression is used to test whether a single-valued relationship has been set:

```
SELECT t
FROM Team t
WHERE t.league IS NULL
```

This query selects all teams where the league relationship is not set.

Note that the following query is not equivalent:

```
SELECT t
FROM Team t
WHERE t.league = NULL
```

The comparison with **NULL** using the equals operator (=) always returns an unknown value, even if the relationship is not set.

The second query will always return an empty result.

Empty Collection Comparison Expressions

The `IS [NOT] EMPTY` comparison expression tests whether a collection-valued path expression has no elements.

In other words, it tests whether a collection-valued relationship has been set.

If the collection-valued path expression is **NULL**, the empty collection comparison expression has a **NULL** value.

Here is an example that finds all orders that do not have any line items:

```
SELECT o
FROM Order o
WHERE o.lineItems IS EMPTY
```

Collection Member Expressions

The `[NOT] MEMBER [OF]` collection member expression determines whether a value is a member of a collection.

The value and the collection members must have the same type.

If either the collection-valued or single-valued path expression is unknown, the collection member expression is unknown.

If the collection-valued path expression designates an empty collection, the collection member expression is `FALSE`.

The `OF` keyword is optional.

The following example tests whether a line item is part of an order:

```
SELECT o
FROM Order o
WHERE :lineItem
MEMBER OF o.lineItems
```

Subqueries

Subqueries may be used in the `WHERE` or `HAVING` clause of a query.

Subqueries must be surrounded by parentheses.

The following example finds all customers who have placed more than ten orders:

```
SELECT c
FROM Customer c
WHERE
 (SELECT COUNT(o)FROM c.orders o)>10
```

Subqueries may contain **EXISTS**, **ALL**, and **ANY** expressions.

- **EXISTS expressions**: The [NOT] EXISTS expression is used with a subquery and is true only if the result of the subquery consists of one or more values and is false otherwise.

  The following example finds all employees whose spouses are also employees:

```
SELECT DISTINCT emp
FROM Employee emp
```

```
WHERE EXISTS (
SELECT spouseEmp
FROM Employee spouseEmp
WHERE spouseEmp = emp.spouse)
```

- **ALL and ANY expressions:** The ALL expression is used with a subquery and is true if all the values returned by the subquery are true or if the subquery is empty.

The ANY expression is used with a subquery and is true if some of the values returned by the subquery are true.

An ANY expression is false if the subquery result is empty or if all the values returned are false.

The SOME keyword is synonymous with ANY.

The `ALL` and `ANY` expressions are used with the `=`, `<`, `<=`, `>`, `>=`, and `<>` comparison operators.

The following example finds all employees whose salaries are higher than the salaries of the managers in the employee's department:

```
SELECT emp
FROM Employee emp
```

```
WHERE emp.salary > ALL (
SELECT m.salary
FROM Manager m
WHERE
m.department = emp.department)
```

Functional Expressions

The query language includes several string, arithmetic, and date/time functions that may be used in the `SELECT`, `WHERE`, or `HAVING` clause of a query.

The functions are listed in Table 34-4, Table 34-5, and Table 34-6.

In Table 34-4, the `start` and `length` arguments are of type `int` and designate positions in the `String` argument.

The first position in a string is designated by 1.

Table 34-4 String Expressions

Function Syntax	Return Type
CONCAT(String, String)	String
LENGTH(String)	int
LOCATE(String, String [, start])	int
SUBSTRING(String, start, length)	String
TRIM([[LEADING|TRAILING|BOTH] char) FROM] (String)	String
LOWER(String)	String
UPPER(String)	String

The `CONCAT` function concatenates two strings into one string.

The `LENGTH` function returns the length of a string in characters as an integer.

The `LOCATE` function returns the position of a given string within a string.

This function returns the first position at which the string was found as an integer.

The first argument is the string to be located.

The second argument is the string to be searched.

The optional third argument is an integer that represents the starting string position.

By default, `LOCATE` starts at the beginning of the string.

The starting position of a string is `1`.

If the string cannot be located, `LOCATE` returns `0`.

The `SUBSTRING` function returns a string that is a substring of the first argument based on the starting position and length.

The `TRIM` function trims the specified character from the beginning and/or end of a string.

If no character is specified, `TRIM` removes spaces or blanks from the string.

If the optional `LEADING` specification is used, `TRIM` removes only the leading characters from the string.

If the optional `TRAILING` specification is used, `TRIM` removes only the trailing characters from the string.

The default is `BOTH`, which removes the leading and trailing characters from the string.

The LOWER and UPPER functions convert a string to lowercase or uppercase, respectively.

In Table 34-5, the number argument can be an int, a float, or a double.

Table 34-5 Arithmetic Expressions

Function Syntax	Return Type
ABS(number)	int, float, or double
MOD(int, int)	int
SQRT(double)	double
SIZE(Collection)	int

The `ABS` function takes a numeric expression and returns a number of the same type as the argument.

The `MOD` function returns the remainder of the first argument divided by the second.

The `SQRT` function returns the square root of a number.

The `SIZE` function returns an integer of the number of elements in the given collection.

In Table 34-6, the date/time functions return the date, time, or timestamp on the database server.

Table 34-6 Date/Time Expressions

Function Syntax	Return Type
CURRENT_DATE	java.sql.Date
CURRENT_TIME	java.sql.Time
CURRENT_TIMESTAMP	java.sql.Timestamp

Case Expressions

Case expressions change based on a condition, similar to the `case` keyword of the Java programming language.

The `CASE` keyword indicates the start of a case expression, and the expression is terminated by the `END` keyword.

The `WHEN` and `THEN` keywords define individual conditions, and the `ELSE` keyword defines the default condition should none of the other conditions be satisfied.

The following query selects the name of a person and a conditional string, depending on the subtype of the `Person` entity.

If the subtype is `Student`, the string `kid` is returned .

If the subtype is `Guardian` or `Staff`, the string `adult` is returned.

If the entity is some other subtype of `Person`, the string `unknown` is returned.

```
SELECT p.name
CASE TYPE(p)
WHEN Student THEN 'kid'
WHEN Guardian THEN 'adult'
WHEN Staff THEN 'adult'
ELSE 'unknown'
END
FROM Person p
```

The following query sets a discount for various types of customers.

Gold-level customers get a 20% discount, silver-level customers get a 15% discount, bronze-level customers get a 10% discount, and everyone else gets a 5% discount.

```
UPDATE Customer c
SET c.discount =
CASE c.level
WHEN 'Gold' THEN 20
WHEN 'SILVER' THEN 15
WHEN 'Bronze' THEN 10
ELSE 5
END
```

NULL Values

If the target of a reference is not in the persistent store, the target is `NULL`.

For conditional expressions containing `NULL`, the query language uses the semantics defined by SQL92.

Briefly, these semantics are as follows.

- If a comparison or arithmetic operation has an unknown value, it yields a `NULL` value.

- Two `NULL` values are not equal.

  Comparing two `NULL` values yields an unknown value.

- The `IS NULL` test converts a `NULL` persistent field or a single-valued relationship field to `TRUE`.

  The `IS NOT NULL` test converts them to `FALSE`.

- Boolean operators and conditional tests use the three-valued logic defined by Table 34-7 and Table 34-8.

(In these tables, T stands for **TRUE**, F for **FALSE**, and U for unknown.)

•

Table 34-7 AND Operator Logic

AND	T	F	U
T	T	F	U
F	F	F	F
U	U	F	U

Table 34-8 OR Operator Logic

OR	T	F	U
T	T	T	T
F	T	F	U
U	T	U	U

Equality Semantics

In the query language, only values of the same type can be compared.

However, this rule has one exception: Exact and approximate numeric values can be compared.

In such a comparison, the required type conversion adheres to the rules of Java numeric promotion.

The query language treats compared values as if they were Java types and not as if they represented types in the underlying data store.

For example, a persistent field that could be either an integer or a `NULL` must be designated as an `Integer` object and not as an `int` primitive.

This designation is required because a Java object can be `NULL`, but a primitive cannot.

Two strings are equal only if they contain the same sequence of characters.

Trailing blanks are significant; for example, the strings `'abc'` and `'abc  '` are not equal.

Two entities of the same abstract schema type are equal only if their primary keys have the same value.

Table 34-9 shows the operator logic of a negation, and Table 34-10 shows the truth values of conditional tests.

Table 34-9 `NOT` Operator Logic

NOT Value	Value
T	F
F	T
U	U

Table 34-10 Conditional Test

Conditional Test	T	F	U
Expression `IS TRUE`	T	F	F
Expression `IS FALSE`	F	T	F
Expression is unknown	F	F	T

SELECT Clause

The `SELECT` clause defines the types of the objects or values returned by the query.

Return Types

The return type of the `SELECT` clause is defined by the result types of the select expressions contained within it.

If multiple expressions are used, the result of the query is an `Object[]`, and the elements in the array correspond to the order of the expressions in the `SELECT` clause and in type to the result types of each expression.

A `SELECT` clause cannot specify a collection-valued expression.

For example, the `SELECT` clause `p.teams` is invalid because `teams` is a collection.

However, the clause in the following query is valid because the `t` is a single element of the `teams` collection:

```
SELECT t
FROM Player p, IN (p.teams) t
```

The following query is an example of a query with multiple expressions in the SELECT clause:

```
SELECT c.name, c.country.name
FROM customer c
WHERE
c.lastname = 'Coss'
AND c.firstname = 'Roxane'
```

This query returns a list of `Object[]` elements; the first array element is a string denoting the customer name, and the second array element is a string denoting the name of the customer's country.

The result of a query may be the result of an aggregate function, listed in Table 34-11.

Table 34-11 Aggregate Functions in Select Statements

Name	Return Type	Description
`AVG`	`Double`	Returns the mean average of the fields
`COUNT`	`Long`	Returns the total number of results
`MAX`	The type of the field	Returns the highest value in the result set
`MIN`	The type of the field	Returns the lowest value in the result set

`SUM`	`Long` (for integral fields) `Double` (for floating-point fields) `BigInteger` (for `BigInteger` fields) `BigDecimal` (for `BigDecimal` fields)	Returns the sum of all the values in the result set

For select method queries with an aggregate function (`AVG`, `COUNT`, `MAX`, `MIN`, or `SUM`) in the `SELECT` clause, the following rules apply:

- The `AVG`, `MAX`, `MIN`, and `SUM` functions return `null` if there are no values to which the function can be applied.

- The `COUNT` function returns 0 if there are no values to which the function can be applied.

The following example returns the average order quantity:

```
SELECT AVG(o.quantity)
FROM Order o
```

The following example returns the total cost of the items ordered by Roxane Coss:

```
SELECT SUM(l.price)
FROM Order o JOIN o.lineItems l
JOIN o.customer c
WHERE c.lastname = 'Coss'
AND c.firstname = 'Roxane'
```

The following example returns the total number of orders:

```
SELECT COUNT(o)
FROM Order o
```

The following example returns the total number of items that have prices in Hal Incandenza's order:

```
SELECT COUNT(l.price)
FROM Order o
JOIN o.lineItems l
JOIN o.customer c
WHERE
c.lastname = 'Incandenza'
AND c.firstname = 'Hal'
```

The `DISTINCT` Keyword

The `DISTINCT` keyword eliminates duplicate return values.

If a query returns a `java.util.Collection`, which allows duplicates, you must specify the `DISTINCT` keyword to eliminate duplicates.

Constructor Expressions

Constructor expressions allow you to return Java instances that store a query result element instead of an `Object[]`.

The following query creates a `CustomerDetail` instance per `Customer` matching the `WHERE` clause.

A `CustomerDetail` stores the customer name and customer's country name.

So the query returns a `List` of `CustomerDetail` instances:

```
SELECT NEW
com.xyz.CustomerDetail
(c.name, c.country.name)
FROM customer c
WHERE
c.lastname = 'Coss'
AND c.firstname = 'Roxane'
```

`ORDER BY` Clause

As its name suggests, the `ORDER BY` clause orders the values or objects returned by the query.

If the `ORDER BY` clause contains multiple elements, the left-to-right sequence of the elements determines the high-to-low precedence.

The `ASC` keyword specifies ascending order, the default, and the `DESC` keyword indicates descending order.

When using the `ORDER BY` clause, the `SELECT` clause must return an orderable set of objects or values.

You cannot order the values or objects for values or objects not returned by the `SELECT` clause.

For example, the following query is valid because the `ORDER BY` clause uses the objects returned by the `SELECT` clause:

```
SELECT o
FROM Customer c JOIN c.orders o
JOIN c.address a
WHERE a.state = 'CA'
ORDER BY o.quantity, o.totalcost
```

The following example is not valid, because the ORDER BY clause uses a value not returned by the SELECT clause:

```
SELECT p.product_name
FROM Order o, IN(o.lineItems) l
JOIN o.customer c
WHERE c.lastname = 'Faehmel'
AND c.firstname = 'Robert'
ORDER BY o.quantity
```

GROUP BY and HAVING Clauses

The GROUP BY clause allows you to group values according to a set of properties.

The following query groups the customers by their country and returns the number of customers per country:

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

The `HAVING` clause is used with the `GROUP BY` clause to further restrict the returned result of a query.

The following query groups orders by the status of their customer and returns the customer status plus the average `totalPrice` for all orders where the corresponding customers has the same status.

In addition, it considers only customers with status `1`, `2`, or `3`, so orders of other customers are not taken into account:

```
SELECT c.status, AVG(o.totalPrice)
FROM Order o JOIN o.customer c
GROUP BY c.status HAVING c.status
IN (1, 2, 3)
```