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**: `BETWEEN` Expressions.

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.