

# Enterprise Component Architectures

## EJB-QL

# EJB-QL



# Purpose

- Under EJB 1.x, entity beans were biased toward relational databases.
  - All the CMP finder methods were JDBC-based.
- Under EJB 2.x, a more flexible model was adopted called *EJB query language (EJB-QL)*.
- EJB-QL seeks to provide a level of abstraction between the entity bean and the backing store.
  - Vendors could provide translators from EJB-QL to any kind of proprietary persistence layer.
  - It has many similarities to SQL.



# Approach

- EJB-QL is used when defining CMP 2.x entity bean **finder** and **select** methods.
- It is defined within the deployment descriptor under special `<query>` tags.
- If an entity is to be part of a EJB-QL statement, it must have defined an `<abstract-schema-name>` in its entry within the deployment descriptor.
  - These schema names must be unique.



# An EJB-QL Example (1 of 3)

## □ The home interface

```
public interface CruiseHomeLocal
    extends javax.ejb.EJBLocalHome {
    public CruiseLocal create(String name, ShipLocal ship)
        throws javax.ejb.CreateException;

    public CruiseLocal findByPrimaryKey(Integer primaryKey)
        throws javax.ejb.FinderException;

    public CruiseLocal findByName(String name)
        throws javax.ejb.FinderException;
}
```



# An EJB-QL Example (2 of 3)

## □ The abstract schema

```
<entity>
  <ejb-name>CruiseEJB</ejb-name>
  <local-home>com.titan.cruise.CruiseHomeLocal</local-home>
  <local>com.titan.cruise.CruiseLocal</local>
  <ejb-class>com.titan.cruise.CruiseBean</ejb-class>
  <persistence-type>Container</persistence-type>
  <prim-key-class>java.lang.Integer</prim-key-class>
  <reentrant>False</reentrant>
  <cmp-version>2.x</cmp-version>
  <abstract-schema-name>Cruise</abstract-schema-name>
  <cmp-field><field-name>id</field-name></cmp-field>
  <cmp-field><field-name>name</field-name></cmp-field>
  <primkey-field>id</primkey-field>
```



# An EJB-QL Example (3 of 3)

## □ The EJB-QL query

```
<query>
  <query-method>
    <method-name>findByName</method-name>
    <method-params>
      <method-param>java.lang.String</method-param>
    </method-params>
  </query-method>
  <ejb-ql>
    SELECT OBJECT( crs ) FROM Cruise AS crs
    WHERE crs.name = ?1
  </ejb-ql>
</query>
...
</entity>
```



# Finder Methods

- Defined in the local and remote home interfaces of an entity bean.
- Single-entity find
  - return a single remote or local bean reference type
- Multi-entity find
  - return `java.util.Collection`, which contains remote or local bean references.
- Each finder method (except `findByPrimaryKey`) must be declared in a `<query>` element.
- Finders in remote and locate interface with the same name and parameter types share a single `<query>` element.



# Select Methods

- Similar to finder method, but can only be defined in the bean implementation class.
- Declared as abstract methods `ejbSelect<Method-Name>`.
- Select methods may return
  - local or remote bean references
  - CMP fields
  - `java.util.Collection` or `java.util.Set` of the above

# An Example of Select Methods (1 of 3)

## □ The bean class

```
public abstract class AddressBean
    implements javax.ejb.EntityBean {
    public abstract Collection ejbSelectZipCodes(String state)
        throws FinderException;

    public abstract Collection ejbSelectAll()
        throws FinderException;
}
```

# An Example of Select Methods (2 of 3)

## □ The EJB-QL query

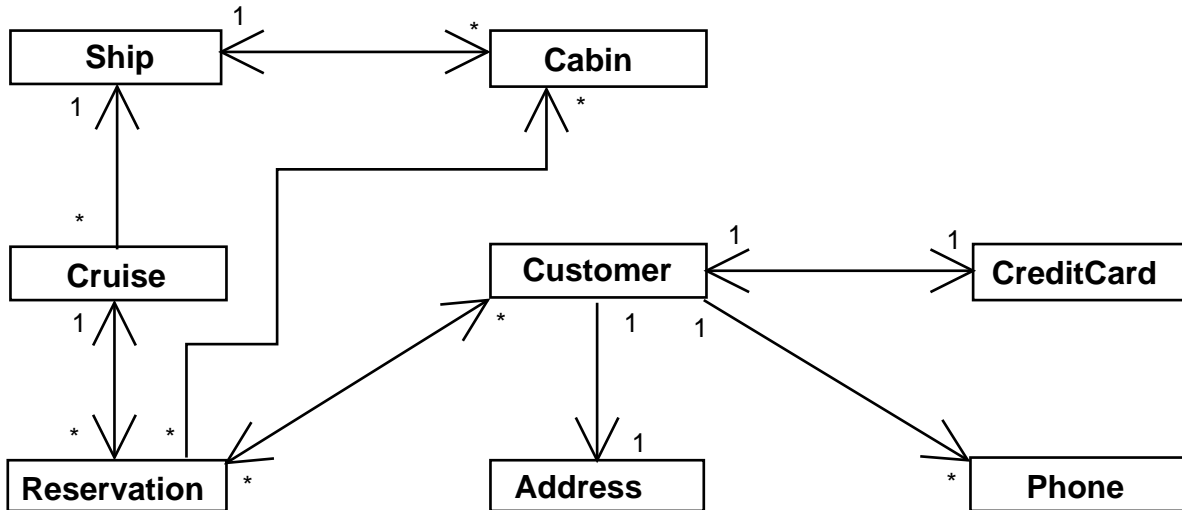
```
<query>
  <query-method>
    <method-name>ejbSelectZipCodes</method-name>
    <method-params>
      <method-param>java.lang.String</method-param>
    </method-params>
  </query-method>
  <ejb-ql>
    SELECT a.zip FROM Address AS a
    WHERE a.state = ?1
  </ejb-ql>
</query>
```

# An Example of Select Methods (3 of 3)

## □ The EJB-QL query

```
<query>
  <query-method>
    <method-name>ejbSelectAll</method-name>
    <method-params/>
  </query-method>
  <ejb-ql>
    SELECT OBJECT(a) FROM Address AS a
  </ejb-ql>
</query>
```

# An Abstract Schema





# Simple Queries

- Simple queries return a single entity bean type represented by an identifier.
  - In these cases you must use the OBJECT keyword.

```
SELECT OBJECT (c) FROM Customer AS c
```

```
SELECT OBJECT (c) FROM Customer c
```



# Simple Query with Paths

- SELECT can return a CMP or CMR field.

```
SELECT c.lastName FROM Customer AS c
```

```
SELECT c.creditCard FROM Customer AS c
```



# Path Queries (1 of 3)

- We can also navigate through the CMP and CMR fields using paths.
  - All this means is that we're following a series of object references.
  - We use the dot-notation.
  - We don't use the OBJECT keyword for the results of path queries.
- Examples: navigate through CMR fields

```
SELECT c.homeAddress.city FROM Customer AS c
```

```
SELECT c.creditCard FROM Customer AS c
```



## Path Queries (2 of 3)

□ There are some restrictions:

- A path can not navigate through non-CMP/CMR fields.  
This means:
  - ★ the instance fields of a bean class are opaque.
  - ★ when a CMP field is a serializable class, its fields are opaque.
- A path can not navigate through collection-based CMR fields.
- A path must end at a single element.

//This is illegal!!!

```
SELECT c.reservations FROM Customer AS c
```

```
SELECT c.reservations.cruise FROM Customer AS c
```



# Selecting Collections

- To return a collection-based relationship, we use the IN operator.
  - We essentially need to convert the collection into a set of single types.
  - The following basically says to visit each customer's reservations and return them.
  - Only customers with non-empty reservation Collections will be assigned to r.

```
SELECT OBJECT( r )  
FROM Customer AS c, IN ( c.reservations ) AS r
```



# Collections and Paths

- We can also use paths after a collection-based relationship.
  - We just use our standard dot-notation.

```
SELECT r.cruise
FROM Customer AS c, IN ( c.reservations ) AS r

SELECT cbn.ship
FROM Customer AS c, IN ( c.reservations ) AS r,
IN (r.cabins) AS cbn
```



# Removing Duplicates

- We might have a query that returns duplicates.
  - If we don't want those duplicates, we need to use the DISTINCT keyword.
  - The following returns all unique reservations for all customers:

```
SELECT DISTINCT OBJECT(c)
FROM Reservation AS r, IN (r.customers) c
```



# Qualified Queries

- We might want to qualify or filter our results based on some criteria.
  - We can do this using the WHERE clause.

```
SELECT OBJECT ( s ) FROM Ship AS s  
WHERE s.tonnage = 1000000
```



# Input Parameters

- Since a **finder** or **select** might require parameters, we need a way to specify them in our **WHERE** clauses.
  - We can do this using the ?-prefix.
  - The first parameter is **?1**, the second is **?2**, and so on.
  - Each of the parameters must be declared in the deployment descriptor entry for the finder or select method.

```
SELECT OBJECT ( s ) FROM Ship AS s  
WHERE s.tonnage = ?1
```



# EJB-QL Operators

□ Arithmetic operators:

+, -, \*, /

□ Comparison operators:

=, >, >=, <, <=, <>,

LIKE, BETWEEN, IN,

IS NULL, IS EMPTY, MEMBER OF

□ Logical operators:

NOT, AND, OR

# Using Operators in WHERE-Clause

```
SELECT OBJECT( r ) FROM Reservation AS r  
WHERE (r.amount * .01) > 300.00
```

```
SELECT OBJECT( s ) FROM Ship AS s  
WHERE s.tonnage >= 80000.00 AND s.tonnage <= 130000.00
```

```
SELECT OBJECT( s ) FROM Ship AS s  
WHERE s.tonnage BETWEEN 80000.00 AND 130000.00
```

```
SELECT OBJECT( s ) FROM Ship AS s  
WHERE s.tonnage NOT BETWEEN 80000.00 AND 130000.00
```



# Null Paths

- We can test to determine if a path expression is null or not.
  - We use the IS NULL expression.
  - The following query identifies customers for whom we have no home address.

```
SELECT OBJECT( c ) FROM Customer c  
WHERE c.homeAddress IS NULL.
```

- The following query identifies customers for whom we have home address.

```
SELECT OBJECT( c ) FROM Customer c  
WHERE c.homeAddress IS NOT NULL.
```



# Selecting Empty Collections

□ Sometimes we might want a list of entities that have empty Collections.

- For instance, we might want to determine which cruises do not have any reservations.

```
SELECT OBJECT( crs )  
FROM Cruise AS crs  
WHERE crs.reservations IS EMPTY
```

- And cruises do have reservations.

```
SELECT OBJECT( crs )  
FROM Cruise AS crs  
WHERE crs.reservations IS NOT EMPTY
```



# Collections and Membership

- We can determine if a particular EJB object is a member of a given Collection-based relationship.
  - We use the MEMBER OF qualifier in the WHERE clause.
  - The following checks to see if a given customer is part of any reservation-customer relationships.

```
SELECT OBJECT( crs )  
FROM Cruise crs, IN ( crs.reservations ) res, Customer cust  
WHERE cust = ?1  
AND cust MEMBER OF res.customers
```



# Pattern Matching (1 of 2)

- Select string fields that match a specified pattern

```
SELECT OBJECT(c) FROM Customer AS c  
WHERE c.lastName LIKE ' %-% '
```

- It matches hyphenated last names.

- The special characters in patterns:

- %: matches any sequence of characters
- \_: matches any single character
- \%: matches the character %
- \\_: matches the character \_



# Pattern Matching (2 of 2)

## □ Examples

- `phone.number LIKE '617%'`
  - ★ Matches phone numbers with 617 prefix
- `phone.number NOT LIKE '617%'`
  - ★ Matches phone numbers with a prefix other than 617
- `cabin.name LIKE 'Suite _100'`
  - ★ Matches suite names end with 100.
- `someField LIKE '\_%'`
  - ★ Matches string fields begin with \_.



# String Functions

- Simple string manipulation is supported
  - `CONCAT(string1, string2)`  
the concatenation of *string1* and *string2*
  - `LENGTH(string)`  
the length of *string*
  - `LOCATE(string1, string2 [, start])`  
the position of *string2* in *string1*.
  - `SUBSTRING(string, start, length)`  
the substring in *string* that begins at *start* with the specified *length*



# Arithmetic Functions

- Simple arithmetic functions are also supported
  - $ABS(number)$   
the absolute value of *number*
  - $SQRT(double)$   
the square root of *double*
  - $MOD(integer1, integer2)$   
the integer remainder of *integer1* divided by *integer2*



# Aggregate Functions (1 of 3)

- COUNT(*path*)

the number of elements in the result set.

- The following query counts the number of customers who live in Wisconsin

```
SELECT COUNT(c) FROM Customer AS c
WHERE c.address.state = 'WI'
```

- The following query counts the number of zip codes that begin with 554

```
SELECT COUNT(c.address.zip)
FROM Customer AS c
WHERE c.address.zip LIKE '554%'
```



# Aggregate Functions (2 of 3)

□ *MAX(path)*

the maximum value in the result set

□ *MIN(path)*

the minimum value in the result set

□ *AVG(path)*

the average of all the values in the result set

□ *SUM(path)*

the sum of all the values in the result set



# Aggregate Functions (3 of 3)

- The DISTINCT operator can be used with any of the aggregate functions to eliminate duplicate values
- The following query counts the number of *different* zip codes that begin with 554

```
SELECT DISTINCT COUNT(c.address.zip)
FROM Customer AS c
WHERE c.address.zip LIKE '554%'
```

# The ORDER-BY Clause (1 of 2)

- To specify the order of the entities in the collection returned by a query
- The following query returns a list of customers in the alphabetical order of their last names.

```
SELECT OBJECT(c) FROM Customer AS c  
ORDERED BY c.lastName
```

# The ORDER-BY Clause (2 of 2)

- You can specify the *ascending* (ASC) or the *descending* (DESC) order

```
SELECT OBJECT(c) FROM Customer AS c  
ORDERED BY c.lastName DESC
```

- You can specify multiple ordered by items

```
SELECT OBJECT(c) FROM Customer AS c  
ORDERED BY c.lastName ASC, c.firstName DESC
```



# CDATA Sections

- Since the deployment descriptor uses XML, the use of our comparison operators such as <, > can cause problems.
  - It's a good idea to embed all of the EJB-QL queries within XML CDATA sections:

```
<![CDATA[  
SELECT OBJECT( r )  
FROM Reservations AS r  
WHERE r.amountPaid > 300  
]]>
```