

OQL

Motivation:

- Relational languages suffer from *impedance mismatch* when we try to connect them to conventional languages like C or C++.
 - ❖ The data models of C and SQL are radically different, e.g. C does not have relations, sets, or bags as primitive types; C is tuple-at-a-time, SQL is relation-at-a-time.
- OQL is an attempt by the OO community to extend languages like C++ with SQL-like, relation-at-a-time dictions.

OQL Types

- Basic types: strings, ints, reals, etc., plus class names.
- Type constructors:
 - ❖ `Struct` for structures.
 - ❖ Collection types: `set`, `bag`, `list`, `array`.
- Like ODL, but no limit on the number of times we can apply a type constructor.
- `Set(Struct())` and `Bag(Struct())` play special roles akin to relations.

OQL Uses ODL as its Schema-Definition Portion

- For every class we can declare an *extent* = name for the current set of objects of the class.
 - ❖ Remember to refer to the extent, not the class name, in queries.

```

interface Bar
    (extent Bars)
{
    attribute string name;
    attribute string addr;
    relationship Set<Sell> beersSold
        inverse Sell::bar;
}

interface Beer
    (extent Beers)
{
    attribute string name;
    attribute string manf;
    relationship Set<Sell> soldBy
        inverse Sell::beer;
}

interface Sell
    (extent Sells)
{
    attribute float price;
    relationship Bar bar
        inverse Bar::beersSold;
    relationship Beer beer
        inverse Beer::soldBy;
}

```

Path Expressions

Let x be an object of class C .

- If a is an attribute of C , then $x.a =$ the value of a in the x object.
- If r is a relationship of C , then $x.r =$ the value to which x is connected by r .
 - ❖ Could be an object or a collection of objects, depending on the type of r .
- If m is a method of C , then $x.m(\dots)$ is the result of applying m to x .

Examples

Let s be a variable whose type is `Sell`.

- `s.price` = the price in the object s .
- `s.bar.addr` = the address of the bar mentioned in s .
 - ◆ Note: cascade of dots OK because `s.bar` is an *object*, not a collection.

Example of Illegal Use of Dot

`b.beersSold.price`, where b is a `Bar` object.

- Why illegal? Because `b.beersSold` is a *set* of objects, not a single object.

OQL Select-From-Where

```
SELECT <list of values>  
FROM <list of collections and  
      typical members>  
WHERE <condition>
```

- Collections in FROM can be:
 1. Extents.
 2. Expressions that evaluate to a collection.
- Following a collection is a name for a typical member, optionally preceded by AS.

Example

Get the menu at Joe's.

```
SELECT s.beer.name, s.price  
FROM Sells s  
WHERE s.bar.name = "Joe's Bar"
```

- Notice double-quoted strings in OQL.

Example

Another way to get Joe's menu, this time focusing on the Bar objects.

```
SELECT s.beer.name, s.price
FROM Bars b, b.beersSold s
WHERE b.name = "Joe's Bar"
```

- Notice that the typical object b in the first collection of FROM is used to help define the second collection.

Typical Usage

- If x is an object, you can extend the path expression, like s or $s.beer$ in $s.beer.name$.
- If x is a collection, you use it in the FROM list, like $b.beersSold$ above, if you want to access attributes of x .

Tailoring the Type of the Result

- Default: bag of structs, field names taken from the ends of path names in `SELECT` clause.

Example

```
SELECT s.beer.name, s.price
FROM Bars b, b.beersSold s
WHERE b.name = "Joe's Bar"
```

has result type:

```
Bag(Struct(
  name: string,
  price: real
))
```


Rename Fields

Prefix the path with the desired name and a colon.

Example

```
SELECT beer: s.beer.name, s.price
FROM Bars b, b.beersSold s
WHERE b.name = "Joe's Bar"
```

has type:

```
Bag(Struct(
  beer: string,
  price: real
))
```

Change the Collection Type

- Use `SELECT DISTINCT` to get a set of structs.

Example

```
SELECT DISTINCT s.beer.name, s.price
FROM Bars b, b.beersSold s
WHERE b.name = "Joe's Bar"
```

- Use `ORDER BY` clause to get a list of structs.

Example

```
joeMenu =
  SELECT s.beer.name, s.price
  FROM Bars b, b.beersSold s
  WHERE b.name = "Joe's Bar"
  ORDER BY s.price ASC
```

- `ASC` = ascending (default); `DESC` = descending.
- We can extract from a list as if it were an array, e.g.

```
cheapest = joeMenu[1].name;
```

Subqueries

- Used mainly in FROM clauses and with quantifiers EXISTS and FORALL.

Example: Subquery in FROM

Find the manufacturers of the beers served at Joe's.

```
SELECT DISTINCT b.manf
FROM (
    SELECT s.beer
    FROM Sells s
    WHERE s.bar.name = "Joe's Bar"
) b
```

Quantifiers

- Boolean-valued expressions for use in WHERE-clauses.

FOR ALL x IN <collection> :
 <condition>

EXISTS x IN <collection> :
 <condition>

- The expression has value TRUE if the condition is true for all (resp. at least one) elements of the collection.

Example

Find all bars that sell some beer for more than \$5.

```
SELECT b.name
FROM Bars b
WHERE EXISTS s IN b.beersSold :
    s.price > 5.00
```

Problem

How would you find the bars that *only* sold beers for more than \$5?

Example

Find the bars such that the only beers they sell for more than \$5 are manufactured by Pete's.

```
SELECT b.name
FROM Bars b
WHERE FOR ALL be IN (
    SELECT s.beer
    FROM b.beersSold s
    WHERE s.price > 5.00
) :
be.manf = "Pete's"
```