Schedule

- Nov. 15 (TH) Object Queries (OQL).
  - Read Sections 9.1. Project Part 6 due on Sunday night.
- Nov. 20 (T) More OQL.
  - Read Sections 9.2-9.3. Assignment 7 due (late ones by email).
- Nov. 22 (TH) Thanksgiving – No class scheduled.
- Nov. 27 (T) Semistructured Data, XML.
  - Read Sections 4.6-4.7. Assignment 8 due. Project Part 7 due.
- Nov. 29 (TH) The Real World, Review.
- Dec. 3 (M) Final, 8–11AM.
ODL Subclasses

Follow name of subclass by colon and its superclass.

Example: Ales are Beers with a Color

interface Ales:Beers {
    attribute string color;
}

• Objects of the Ales class acquire all the attributes and relationships of the Beers class.

• While E/R entities can have manifestations in a class and subclass, in ODL we assume each object is a member of exactly one class.
Keys in ODL

Indicate with key(s) following the class name, and a list of attributes forming the key.

• Several lists may be used to indicate several alternative keys.
• Parentheses group members of a key, and also group key to the declared keys.
• Thus, \( \text{key} (a_{12...n}) \) = “one key consisting of all \( n \) attributes.”
  \( \text{key} a_{12...n} \) = “each \( a_i \) is a key by itself.”

Example

interface Beers (key name)
{attribute string name ...}

• Remember: Keys are optional in ODL. The “object ID” suffices to distinguish objects that have the same values in their elements.
Example: A Multiattribute Key

interface Courses
    (key (dept, number), (room, hours))
{
    ...
}
Translating ODL to Relations

1. Classes without relationships: like entity set, but several new problems arise.

2. Classes with relationships:
   a) Treat the relationship separately, as in E/R.
   b) Attach a many-one relationship to the relation for the “many.”
ODL Class Without Relationships

• Problem: ODL allows attribute types built from structures and collection types.
• Structure: Make one attribute for each field.
• Set: make one tuple for each member of the set.
  ◆ More than one set attribute? Make tuples for all combinations.
• Problem: ODL class may have no key, but we should have one in the relation to represent “OID.”
Example

```java
interface Drinkers (key name) {
    attribute string name;
    attribute Struct Addr
        {string street, string city,
         int zip} address;
    attribute Set<string> phone;
}
```

<table>
<thead>
<tr>
<th>name</th>
<th>street</th>
<th>city</th>
<th>zip</th>
<th>phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n_1$</td>
<td>$s_1$</td>
<td>$c_1$</td>
<td>$z_1$</td>
<td>$p_1$</td>
</tr>
<tr>
<td>$n_1$</td>
<td>$s_1$</td>
<td>$c_1$</td>
<td>$z_1$</td>
<td>$p_2$</td>
</tr>
</tbody>
</table>

- Surprise: the key for the class (name) is not the key for the relation (name, phone).
  - `name` in the class determines a unique object, including a set of phones.
  - `name` in the relation does not determine a unique tuple.
  - Since tuples are not identical to objects, there is no inconsistency!
- BCNF violation: separate out name-phone.
ODL Relationships

• If the relationship is many-one from \( A \) to \( B \), put key of \( B \) attributes in the relation for class \( A \).
• If relationship is many-many, we’ll have to duplicate \( A \)-tuples as in ODL with set-valued attributes.
  ◆ Wouldn’t you really rather create a separate relation for a many-many-relationship?
  ◆ You’ll wind up separating it anyway, during BCNF decomposition.
Example

interface Drinkers (key name) {
    attribute string name;
    attribute string addr;
    relationship Set<Beers> likes
        inverse Beers::fans;
    relationship Beers favorite
        inverse Beers::realFans;
    relationship Drinkers husband
        inverse wife;
    relationship Drinkers wife
        inverse husband;
    relationship Set<Drinkers> buddies
        inverse buddies;
}

Drinkers(name, addr, beerName, favBeer, wife, buddy)

- Not in BCNF; decompose to:
  Drinkers(name, addr, favBeer, wife)
  DrBeer(name, beer)
  DrBuddy(name, buddy)
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 \( x.m(\cdots) \) 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.

```sql
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

```sql
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

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

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