Object-Oriented DBMS's

- ODMG = Object Data Management Group: an OO standard for databases.
- ODL = Object Description Language: design in the OO style.
- OQL = Object Query Language: queries an OO database with an ODL schema, in a manner similar to SQL.

ODL Overview

- Class declarations (*interfaces*).
- Interface includes:
    1. Name for the interface.
    2. Key declaration(s), which are optional.
    3. *Extent* declaration = name for the set of currently existing objects of a class.
    4. *Element* declarations. An element is an attribute, a relationship, or a method.

ODL Class Declarations

```
interface <name> {
    elements = attributes, relationships,
        methods
}
```

Element Declarations

```
attribute <type> <name>;
relationship <rangetype> <name>;
```

- Relationships involve objects; attributes involve non-object values, e.g., integers.

Method Example

```
float gpa(in string) raises(noGrades)
```

- `float` = return type.
- `in`: indicates the argument (a student name, presumably) is read-only.
  - Other options: `out`, `inout`.
- `noGrades` is an exception that can be raised by method `gpa`.

ODL Relationships

- Only binary relations supported.
  - Multiway relationships require a "connecting" class, as discussed for E/R model.
- Relationships come in inverse pairs.
  - Example: "Sells" between beers and bars is represented by a relationship in bars, giving the beers sold, *and* a relationship in beers giving the bars that sell it.
- Many-many relationships have a set type (called a *collection type*) in each direction.
- Many-one relationships have a set type for the one, and a simple class name for the many.
- One-one relations have classes for both.

Beers-Bars-Drinkers Example

```
interface Beers {
    attribute string name;
    attribute string manf;
    relationship Set<Bars> servedAt
        inverse Bars::serves;
    relationship Set<Drinkers> fans
        inverse Drinkers::likes;
}
```

- An element from another class is indicated by <class>`::`
- Form a set type with `Set<type>`.

```
interface Bars {
    attribute string name;
    attribute Struct Addr
        {string street, string city, int zip}
            address;
    attribute Enum Lic {full, beer, none}
            licenseType;
    relationship Set<Drinkers> customers
        inverse Drinkers::frequents;
    relationship Set<Beers> serves
        inverse Beers::servedAt;
}
```

- Structured types have names and bracketed lists of field-type pairs.
- Enumerated types have names and bracketed lists of values.

```
interface Drinkers {
    attribute string name;
    attribute Struct Bars::Addr
            address;
    relationship Set<Beers> likes
        inverse Beers::fans;
    relationship Set<Bars> frequents
        inverse Bars::customers;
}
```

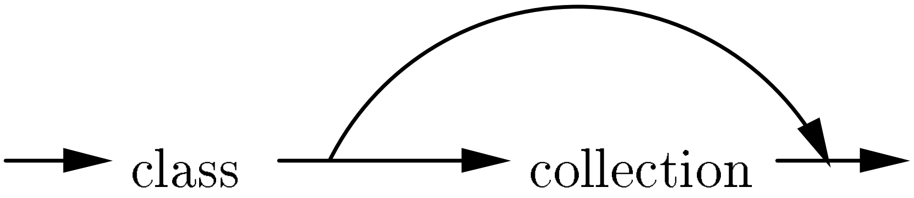
- Note reuse of `Addr` type.

ODL Type System

- Basic types: int, real/float, string, enumerated types, and classes.
- Type constructors: `Struct` for structures and four *collection types*: `Set`, `Bag`, `List`, and `Array`.

Limitation on Nesting

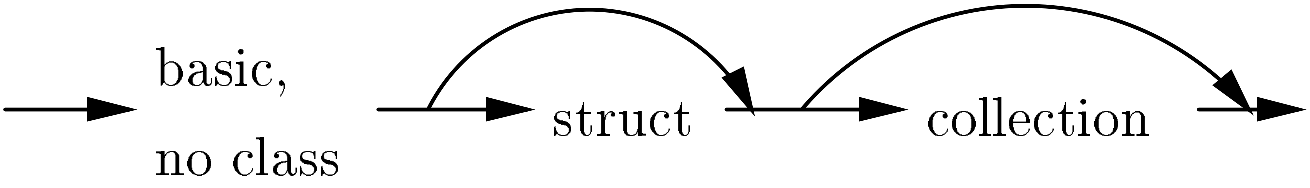
Relationship



class

collection

Attribute



basic,

no class

struct

collection

Many-One Relationships

Don't use a collection type for relationship in the "many" class.

Example: Drinkers Have Favorite Beers

```
interface Drinkers {
    attribute string name;
    attribute Struct Bars::Addr
            address;
    relationship Set<Beers> likes
        inverse Beers::fans;
    relationship Beers favoriteBeer
        inverse Beers::realFans;
    relationship Set<Bars> frequents
        inverse Bars::customers;
}
```

- Also add to `Beers`:

```
    relationship Set<Drinkers> realFans
        inverse Drinkers::favoriteBeer;
```

Example: Multiway Relationship

Consider a 3-way relationship bars-beers-prices. We have to create a connecting class `BBP`.

```
interface Prices {
    attribute real price;
    relationship Set<BBP> toBBP
        inverse BBP::thePrice;
}

interface BBP {
    relationship Bars theBar inverse ...
    relationship Beers theBeer inverse ...
    relationship Prices thePrice
        inverse Prices::toBBP;
}
```

- Inverses for `theBar`, `theBeer` must be added to `Bars`, `Beers`.
- Better in this special case: make no `Prices` class; make `price` an attribute of `BBP`.
- Notice that keys are optional.
  - ✦ `BBP` has no key, yet is not "weak." Object identity suffices to distinguish different `BBP` objects.

Roles in ODL

Names of relationships handle "roles."

Example: Spouses and Drinking Buddies

```
interface Drinkers {
    attribute string name;
    attribute Struct Bars::Addr
            address;
    relationship Set<Beers> likes
        inverse Beers::fans;
    relationship Set<Bars> frequents
        inverse Bars::customers;
    relationship Drinkers husband
        inverse wife;
    relationship Drinkers wife
        inverse husband;
    relationship Set<Drinkers> buddies
        inverse buddies;
}
```

- Notice that `Drinkers::` is optional when the inverse is a relationship of the same class.

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, `(key(`$a_1, a_2, \ldots, a_n$`))` $=$ "one key consisting of all n attributes."
  `(key` $a_1, a_2, \ldots, a_n$`)` $=$ "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

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

<u>name</u>	street	city	zip	p<u>hone</u>
n_1	s_1	c_1	z_1	p_1
n_1	s_1	c_1	z_1	p_2

- 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)
```