Object Orientation: Types, Classes and Objects

```
Type System:
  base types: integers, real numbers, booleans, character strings
  type constructors:
        record structures (struct)
          T1 f1, T2 f2, ..., Tn,fn
        collection types
          array, list, set, bag of T
        reference types
          memory address, disk address (for persistent objects) etc.

Classes and Objects:
  class consists of a type and possibly one or more methods.
  objects are either values of the type
                     called immutable objects (e.g. {2,5,7})
              or variables whose value is of that type
                     called mutable objects e.g. T s; s <- {2,5,7}
```

Object Orientation: Object Identity, Methods, ADTs

Object Identity:

- oid is unique (i.e. No two objects can have the same oid)
- oid must be valid at all time for persistent objects.

Methods:

- associated with a class.

Abstract Data Types:

- classes are also "abstract data types" because the only way to modify the state of an object is via methods.
- key concept in reliable software development.

Object Orientation: Class Hierarchies

Sub-class C of super-class D.

- C inhertis all properties of D including the type of D and all methods of D.
- However C may have additional properties (new methods in addition or in place of superclass methods).
- C may also extend the type of D by adding new fields.

Example:

```
CLASS Account = {
  accountNo: integer;
  balance: real;
  owner: REF Customer;
  deposit(m: real);
  withdraw(m: real);
}
```

```
CLASS SavingsAccount::Account {
  interestRate: real;
  computeInterest();
}
```

ODMG, ODL and OQL

ODMG: Object Data Management Group (a standards group)

ODL: Object Description Language (schema description language)

OQL: Object Query Language (queries an OO database with an ODL schema, similar to SQL)

ODL

ODL Classes (called interfaces): contain definitions for

- attributes
- relationships
- methods

Consider database about movies, stars and studios.

Movies have stars acting in them.
Stars may act in one or more movies.
Studios produce one or more movies.
Each movie is produced by one studio.

ODL Schema for Movie database

```
interface Movie {
  attribute string title;
  attribute integer year;
  attribute integer length;
  attribute enum Film {color,blackAndWhite} filmType;
  relationship Set<Star> stars inverse Star::starredIn;
  relationship Studio ownedBy inverse Studio::owns;
};
interface Star {
  attribute string name;
  attribute Struct Addr {string street, string city} address;
  relationship Set<Movie> starredIn inverse Movie::stars;
};
interface Studio {
  attribute string name;
  attribute strin address;
  relationship Set<Movie> owns inverse Movie::ownedBy;
};
```

Types in ODL

```
Basis Types:
  Atomic Types: integer, float, character string, boolean, enumeration
  Interface Types: such as Movie, Star, Studio etc defined by users.

Type Constructors:
  Collection Types: Set<T>
                    Bag<T>
                    List<T>
                    Array<T,n> array of n objects of type T
  non-Collection Type:
      Struct N {T1 F1, ..., Tn Fn}
```

Restrictions on Attribute Types

- The type of an attribute is built starting with an atomic type or a structure of atomic types
- Then we may apply a collection type to the initial atomic type or structure.

Attribute type examples:

```
integer
Struct N {string field1, integer field2}
List<real>
Array<Struct N {string field1, integer field2}>
```

Illegal attribute types:

```
Set<Movie>
Movie
```

Restrictions on Relationship Types

```
  - The type of a relationship is
      either an interface type or a collection type applied to
         an interface type.

  - relationship type examples:
            Movie
            Bag<Star>

Illegal relationship types:
  Struct N {Movie field1, Start field2}
  Set<integer>
  Set<Array<Star>>

Note: interface types are not allowed in attribute types and
      atomic types are not allowed in relationship types.
```

Subclasses, Keys

Subclasses in ODL

```
interface Cartoon: Movie {
  relationship Set<Star> voices;
}

interface MurderMystery: Movie {
  attribute string weapon;
}
```

Defining keys in ODL

```
interface Movie {
  ...
  (key (title, year))
}
```

ODL Designs to Relational Designs

```
classes -> relations
properties -> attributes
```

Example 1: (simple atomic properties)

```
interface Movie {
  attribute string title;
  attribute integer year;
  attribute integer length;
  attribute enumeration(color,blackAndWhite) filmType;
}
```

translates into

```
Movie(title,year,length,filmType)
```

ODL Designs to Relational Designs – continued

Example 2: (non-atomic property; Struct)

```
interface Star {
  attribute string name;
  attribute Struct Addr {string street, string city} address;
}
```

translates into

```
Star(name,street,city)
```

ODL Designs to Relational Designs – continued

Example 3: (Set constructor)

```
interface Star {
  attribute string name;
  attribute Set<Struct Addr {string street, string city}> address;
  attribute Date birthdate;
}
```

translates into

```
Star(name,birthdate)
Star(name,street,city)
```

Each "multi-valued attribute" results in a separate relation.

ODL Designs to Relational Designs – continued

If the address attribute is a Bag then the relation would be:

    Star(name,street,city,count)

count attribute: number of times address is in the bag.

If the address attribute is a List then the relation would be:

    Star(name,street,city,position)

position attribute: position of the address in the List.

If the address attribute is a fixed-length (say length=2) Array then the relation would be:

    Star(name,street1,city1,street2,city2)

ODL Designs to Relational Designs – continued

Example 4: (Single-valued relationship)

```
interface Movie {
  ...
  relationship Studio ownedBy inverse Studio::owns;
  ...
}
```

corresponding relation:

Movie(...,studioName,...)

where studioName is the primary key of Studio relation derived from Studio class

ODL Designs to Relational Designs – continued

Example 4: (Multi-valued relationship)

```
interface Movie {
  ...
  relationship Set<Star> stars inverse Star::starredIn;
  ...
}
```

corresponding relation for stars relationship:

Stars(title,year,sname)

title,year: key for Movie   sname: key for Star

In the relational model, only one direction need be represented.
(in case of one-to-many relationship, represent the single-valued side).

ODL Designs to Relational Designs – continued

Converting Subclasses to relations:

Consider the Movie --> Cartoon Movie --> MurderMystery

```
interface Cartoon: Movie {
  relationship Set<Star> voices;
}

interface MurderMystery: Movie {
  attribute string weapon;
}
```

ODL Designs to Relational Designs – continued

3 approaches:

(1) one relation per subclass; includes all attributes (including inherited ones)

```
Movie(title,year,length,filmType,studioName)
  MovieStars(title,year,starName)
Cartoon(title,year,length,filmType,studioName)
  CartoonStars(title,year,starName)
  Voices(title,year,voiceName)
MurderMystery(title,year,length,filmType,studioName,weapon)
  MurderMysteryStars(title,year,starName)
```

All information in one place for a particular movie!
However to query common attributes such as length, we have to search all 3 relations.

ODL Designs to Relational Designs – continued

(2) one relation per subclass;
includes only the attributes of the sub-class
not the inherited ones (except primary keys)

```
Movie(title,year,length,filmType,studioName)
  MovieStars(title,year,starName)
Cartoon(title,year)
  Voices(title,year,voiceName)
MurderMystery(title,year,weapon)
```

Information for a particular movie scattered around!
Querying common attributes done on one relation.

ODL Designs to Relational Designs – continued

(3) one relation with all attributes with lots of null values.

```
Movie(title,year,length,filmType,studioName,weapon)
MovieStars(title,year,starName)
Voices(title,year,voiceName)
```

Querying OO Databases

Query-related features of ODL:

Declaring method signatures in ODL: (code is not part of ODL)

```
interface Movie (extent Movies key (title,year)) {
  ...
  float lengthInHours() raises (noLengthFound);
  starNames(out Set<String>);
  otherMovies(in Star, out Set<Movie>) raises(noSuchStar);
};
```

- extent of the class: name for the current set of objects in the class
- OQL queries refer to the extent of a class, not to the class name
- in, out, inout parameters in methods
- functions may raise exceptions
- there is no guarantee function implements what their names suggest!

OQL

OQL (Object Query Language -- ODMG standard)

- OQL queries may be interpreted (as in SQL*Plus of Oracle) or may be embedded in a host language such as C++, Java.
- Much easier to embed OQL queries in host language since both are compatible (values easily transferred between the two)

OQL - continued

```
OQL Type system:
  Constants are constructed as follows:
    - Basic Types: atomic types: integers, floats, characters,
                        strings, and booleans (surrounded by ")
                   enumerations declared in ODL!
    - Complex Types: built using
        Set(...)
        Bag(...)
        List(...)
        Array(...)
        Struct(...)

       examples: bag(2,1,2)
                 struct(foo:bag(2,1,2), bar: "baz")
                 set(struct(title:"My Fair Lady",year:1965),
                     struct(title:"ET",year:1985),
                     struct(title:"Jaws",year:1981))
```

OQL - continued

Path Expressions:

a: an object belonging to class C

p: some property of the class (attribute/relationship/method)

a.p is a path expression interpreted as follows:

- if p is an attribute then a.p is the value of that attribute in object a
- if p is a relationship then a.p is the object or collection of objects related to a by relationship p
- if p is a method (perhaps with parameters) then a.p is the result of applying p to a

OQL - continued

Examples of path expressions:

```
        Movie myMovie;
        Set<string> sNames;

myMovie.length
myMovie.lengthInHours()
myMovie.stars
myMovie.starNames(sNames)
myMovie.ownedBy.name
```

OQL - continued

OQL Queries:

(1) Find the year of movie "Gone With the Wind"

```
select m.year
from Movies m
where m.title = "Gone With the Wind"
```

OQL - continued

Select-from-where statement in OQL is constructed as follows:

SELECT keyword followed by a list of expressions (using constants and variables defined in the FROM clause)

FROM keyword followed by a list of variable declarations
variable is declared by

- giving an expression whose value is a collection type (typically an extent; could be another select-from-where)
- followed by an optional AS keyword
- followed by the name of the variable

WHERE keyword followed by a boolean-valued expression (can use only constants and variables declared in the FROM clause);

The OQL query produces a bag of objects.

OQL - continued

(2) Find the names of the stars of "Casablanca"

```
select s.name
from  Movies m, m.stars s
where m.title = "Casablanca"
```

(3) Eliminating duplicates (distinct keyword)
Find the names of stars of "Disney" movies

```
select distinct s.name
from Movies m, m.stars s
where m.ownedBy.name = "Disney"
```

OQL - continued

Complex output type:

(4) Get set of pairs of stars living at the same address

```
select distinct Struct(star1: s1, star2: s2)
from Stars s1, Stars s2
where s1.addr = s2.addr and s1.name < s2.name
```

The result type of this query is

```
Set<Struct N {star1: Star, star2: Star}>
```

Note: Such a type cannot appear in an ODL declaration!

```
shortcut: select star1: s1, star2: s2
```

OQL - continued

Subqueries:

(5) Get the stars in movies made by Disney.

```
select m
from Movies m
where m.ownedBy.name = "Disney"
```

gives us the Disney Movies. This can be used in the from clause as follows:

```
select distinct s.name
from (select m
      from Movies m
      where m.ownedBy.name = "Disney") d, d.stars s
```

OQL - continued

Ordering the Result:

(6) Get Disney movies ordered by length (ties broken by title)

```
select m
from Movies m
where m.ownedBy.name = "Disney"
order by m.length, m.title
```

- asc or desc may be spevcified after order by (default is asc)

OQL - continued

Quantifier Expressions:

```
for all x in S: C(x)
exists x in S: C(x)
```

(7) Get stars acting in Disney movies

```
select s
from Stars s
where exists m in s.starredIn : m.ownedBy.name = "Disney"
```

(8) Get stars who appear only in Disney movies

```
select s
from Stars s
where for all m in s.starredIn : m.ownedBy.name = "Disney"
```

OQL - continued

Aggregation Expressions:

same 5 operations as in SQL: avg, min, max, sum, count

These apply to collections whose members are of a suitable type.
count applies to any collection
sum, avg apply to any collection of numbers
min, max apply to any collection in which the members can be compared.

(9) Find the average length of all movies.

```
avg(select m.length from Movies m)
```

a bag of movie lengths is created; then the avg operator is applied.
(note: set of movie lengths would be incorrect!)

OQL - continued

Set Operators:

union, difference, and intersection on two objects of set or bag type.

(12) Find movies starring "Harrison Ford" that were not made by "Disney"

```
(select distinct m
 from   Movies m, m.stars s
 where s.name = "Harrison Ford")
except
(select distinct m
 from   Movies m
 where m.ownedBy.name = "Disney")
```

OQL - continued

Note: If the one or both operands of these set operations is a bag, the "bag" meaning is used. Say x appears n1 times in B1 and n2 times in B2 then

x appears n1+n2 times in (B1 union B2)

x appears min(n1,n2) times in (B1 intersect B2)

x appears 0 times in (B1 difference B2) if n1 <= n2 (n1-n2) times otherwise

The result of the query is a set if both operands are sets otherwise it is a bag.

OQL - continued

Object Assignment and Creation in OQL. OQL and host language (good fit!).

Assigning Values to host variables:

```
Set<Movie> oldMovies;
oldMovies = select distinct m from movies m where m.year < 1920;
```

Extracting Elements of Collections:

```
Movie gwtw;
gwtw = element(select m from Movies m where m.title = "Gone With the Wind");
```

element function extracts single element from bag of one element.

OQL - continued

Extracting each element from a collection:

```
List<Movie> movieList;
movieList = (select m from Movies m order by m.title,m.year);
movieList[i] now refers to the ith movie in the list.
```

Note: order by clause automatically converts the result type of query to a list instead of bag/set.

small program fragment to display movie titles, years and lengths:

```
List<Movie> movieList;
movieList = (select m from Movies m order by m.title,m.year);
int numberOfMovies = count(movieList);
for (int i=0;i<numberOfMovies;i++) {
  Movie m = movieList[i];
  System.out.println(m.title,m.year,m.length);
}
```

OQL - continued

Creating New Objects:

```
x = Struct(a:1, b:2);
y = Bag(x,x,Struct(a:3,b:4));

gwtw = Movie(title:"Gone With the Wind", year:1919, length:239, ownedBy:mgm);
ms = oldMovies except Set(gwtw);
```