

# COURSE 11

---

Object-Oriented Databases  
Object Relational Databases

# The Need for a DBMS

---

- On one hand we have a tremendous increase in the amount of data applications have to handle, on the other hand we want a reduced application development time.
  - Object-Oriented programming
  - DBMS features: query capability with optimization, concurrency control, recovery, indexing, etc.
- Can we merge these two to get an object database management system since data is getting more complex?

# What are the needs?

- Images
- Video
- Multimedia in general
- Spatial data (GIS)
- Biological data
- CAD data
- Virtual Worlds
- Games
- List of lists
- User defined data types



# Manipulating New Kinds of Data

- A television channel needs to store video sequences, radio interviews, multimedia documents, geographical information, etc., and retrieve them efficiently.
- A movie producing company needs to store movies, frame sequences, data about actors and theaters, etc. (textbook example)
- A biological lab needs to store complex data about molecules, chromosomes, etc, and retrieve parts of data as well as complete data.
- Think about commercial needs.

# Shortcomings with Relational DBMSs

- No set-valued attributes
- No inheritance
- No complex objects, apart from BLOB (binary large object)
- Impedance mismatch between data access language (declarative SQL) and host language (procedural C or Java): programmer must explicitly tell how things to be done.

⇒ Is there a different solution?

# Existing Object Databases

- Object database is a persistent storage manager for objects:
  - Persistent storage for object-oriented programming languages (C++, SmallTalk, etc.)
  - Object-Database Systems:
    - Object-Oriented Database Systems: alternative to relational systems
    - Object-Relational Database Systems: extension to relational systems
- Market: RDBMS (\$8 billion), OODMS (\$30 million) world-wide
- OODB Commercial Products: *ObjectStore*, *GemStone*, *Orion*...

# Object Data Model

---

- The object data model is the basis of object oriented databases, like the relational data model is the basis for the relational databases.
- The database contains a collection of Objects (similar to the concept of entities)
- An object has a unique ID (OID) and a collection of objects with similar properties is called a class.
- Properties of an object are specified using ODL and objects are manipulated using OML.

# Properties of an Object

---

- **Attributes:** atomic or structured type (set, bag, list, array)
- **Relationships:** reference to an object or set of such objects.
- **Methods:** functions that can be applied to objects of a class.



# Abstract Data Type

---

- One key feature of object database systems is the possibility for the user to define arbitrary new data types.
- A new data type should come with its associated methods to manipulate it. The new data type and its associated methods is called abstract data type (ADT).
- DBMS has built-in types.
- How does the DBMS deal with new data types that were never seen before?

# Encapsulation

---

- Encapsulation = data structure + operations
- It is the main characteristic of object-oriented languages.
- The encapsulation hides the abstract data type internals. ADT= opaque type.
- The DBMS does not need to know how the ADT's data is stored nor how the ADT's methods work. DBMS only needs to know the available methods and how to call them (input/output types of the methods)

# Inheritance

A value has a type An object belongs to a class
--

- Type hierarchy
  - System permits the definition of new types based on other existing types
  - A subtype inherits all properties of its supertype
- Class hierarchy
  - A sub-class  $C'$  of a class  $C$  is a collection of objects such that each object in  $C'$  is also an object in  $C$ .
  - An object in  $C'$  inherits all properties of  $C$
- Multiple inheritance (inherits from  $> 1$  superclass)
- Selective inheritance (inherits only some of the properties of a superclass)

# Object-Oriented Databases

- OO DBMS aims to achieve seamless integration with an object-oriented programming language such as C++, C#, Java etc.
- OO DBMS is aimed at applications when an object-centric view point is appropriate.  
(occasional fetch from object repository)
- ODL = Object Description Language, like **CREATE TABLE** part of SQL.
- OML = Object Manipulation Language, tries to imitate SQL in an OO framework.

# ODL in Object-Oriented DBMS

- ODL is used to define *persistent* classes, those whose objects may be stored permanently in the database.
  - ODL classes look like *Entity* sets with binary *relationships*, plus *methods*.
  - ODL class definitions are part of the extended, OO host language.

# ODL Overview

---

- A class declaration includes:
  - ♦ A name for the class.
  - ♦ Optional key declaration (s).
  - ♦ *Extent* declaration = name for the set of currently existing objects of the class.
  - ♦ Element declarations. An *element* is either an attribute, a relationship, or a method.

```
class <name> {  
    <list of element declarations,  
        separated by semicolons>}  
}
```

# Attribute and Method Declarations

- Attributes are (usually) elements with a type that does not involve classes.

**attribute** <type> <name>;

- The information of a method declaration consists of:
  - Return type (if any)
  - Method name
  - Argument modes - in, out, inout - and types (no names)
  - Any exception the method may raise

**real** grade\_avg(**in** **string**) **raises** (noGrades) ;

# Relationship Declaration

- Relationships connect an object of one class to one or more objects of other class.
- Relationships are stored like pair of inverse pointers (A points to B and B points back to A)
- Relationships are automatically maintained by the system (if A is deleted, B's pointer to A is set to NULL)
- Types of relationships: one-to-one, one-to-many, many-to-many


```
relationship <type> <name> inverse <relationship>;
```



# Example


```
class Movie{  
    attribute date start;  
    attribute date end;  
    attribute string movieName;  
    relationship Set<Cinema> shownAt inverse  
                                                Cinema::nowShowing;  
}
```

*type of relationship*



```
class Cinema {  
    attribute string cinemaName;  
    attribute string address;  
    attribute integer ticketPrice;  
    relationship Set <Movie> nowShowing inverse  
                                                Movie::shownAt  
  
    float numshowing() raises(errorCountingMovies);  
}
```

*:: operator connects a name to the context*



# Types of Relationships

- The type of a relationship is either
  - ♦ A class, like *Movie*. If so, an object with this relationship can be connected to only one *Movie* object.
  - ♦ **Set**<*Movie*>: the object is connected to a set of *Movie* objects.
  - ♦ **Bag**<*Movie*>, **List**<*Movie*>, **Array**<*Movie*>: the object is connected to a bag, list, or array of *Movie* objects.

# Multiplicity of Relationships

- All ODL relationships are binary.
- Many-many relationships have *Collection* for the type of the relationship and its inverse.
- Many-one relationships have *Collection*<...> in the relationship of the “one” and just the class for the relationship of the “many.”
- One-one relationships have classes as the type in both directions.

# Example

```
class Drinker { ...  
  relationship Set<Beer> likes inverse  
                                     Beer::fans;  
  relationship Beer favBeer inverse  
                                     Beer::superfans;  
}  
  
class Beer { ...  
  relationship Set<Drinker> fans inverse  
                                     Drinker::likes;  
  relationship Set<Drinker> superfans inverse  
                                     Drinker::favBeer;  
}
```

*many-to-many*

*one-to-many*

## Example (cont)


```
class Person{  
    attribute ...;
```

```
    relationship Person husband inverse wife;  
    relationship Person wife inverse husband;
```

```
    relationship Set<Person> buddies  
        inverse buddies;
```

```
}
```

*husband and wife are  
one-to-one and inverses  
of each other*



*buddies is many-to-many  
and its own inverse*



# Connecting Classes

---

- Suppose we want to connect classes  $X$ ,  $Y$  and  $Z$  by a relationship  $R$ .
- Create a class  $C$ , whose objects represent a triple of objects  $(x, y, z)$  from classes  $X$ ,  $Y$  and  $Z$ , respectively.
- We need three many-one relationships from  $(x, y, z)$  to each of  $x$ ,  $y$ , and  $z$ .

# Example: Connecting Class

- Suppose we have *BookStore* and *Book* classes, and we want to represent the price at which each *BookStore* sells each book.
  - A many-many relationship between *BookStore* and *Book* cannot have a price attribute as it did in the E/R model.
- Solution 1: create class *Price* and a connecting class *BBP* to represent a related book store, book and price.
- Solution 2: since *Price* objects are just numbers is better to:
  - Give *BBP* objects an attribute price.
  - Use two many-one relationships between a *BBP* object and the *BookStore* and *Book* objects it represents.

## Example: Connecting Class (cont)

- Here is the definition of BBP:

```
class BBP {  
    attribute real price;  
    relationship BookStore theBS inverse  
        BookStore::toBBP;  
    relationship Book theBook inverse  
        Book::toBBP; }
```

- *BookStore* and *Book* must be modified to include relationships, both called *toBBP*, and both of type *Set<BBP>*.



# The ODL Type System

---

- Basic types: *int*, *real/float*, *string*, *enumerated types*, and *classes*.
- Type constructors:
  - *Struct* for structures.
  - Collection types: *Set*, *Bag*, *List*, *Array* and *Dictionary* (mapping from a domain type to a range type).
- Relationship types can only be a class or a single collection type applied to a class.

# ODL Subclasses

- Usual object-oriented subclasses.
- Indicate super-class with a colon and its name.
- Subclass lists only the properties unique to it.
  - Also inherits its super-class properties.

```
class Student:Person
{
    attribute string code;
    . . .
}
```

# ODL Keys and Extents

- You can declare any number of keys for a class.
- After the class name, add: (**key** <list of keys>)
- A key consisting of more than one attribute needs additional parentheses around those attributes.
- For each class there is an *extent*, the set of existing objects of that class:
  - Think of the extent as the one relation with that class as its schema.
- Indicate the extent after the class name, along with keys, as: (**extent** <extent name> ... )
  - Conventionally: use singular for class names, plural for the corresponding extent.

# Example

---

```
class Book
```

```
    (key name) { ... }
```

```
class Course
```

```
    (key (dept, number) ,  
        (room, hours)) { ... }
```

```
class Student
```

```
    (extent Students key code) { ... }
```

# OML in Object Oriented DBMS

- No efficient implementations for OML. There are no good optimizations for a query language
- The most popular query language is OQL (Object Query Language) which is designed to have a syntax similar to SQL.
- OQL is an extension to SQL. It has **select**, **from**, **where** and **group by** clauses.
- The extensions are to accommodate the properties of objects and the operators on complex data types.

# Running Example

```
class Movie (extent Movies key movieName) {
    attribute date start;
    attribute date end;
    attribute string movieName;
    relationship Set<Cinema> shownAt inverse
                                                Cinema::nowShowing;
}

class Cinema (extent Cinemas key cinemaName) {
    attribute string cinemaName;
    attribute string address;
    attribute integer ticketPrice;
    relationship Set<Movie> nowShowing inverse
                                                Movie::shownAt;
    float numshowing() raises(errorCountingMovies);
}
```

# Path Expressions

- Let  $x$  be an object of class  $C$ .
  - ◆ If  $a$  is an attribute of  $C$ , then  $x.a$  is the value of that attribute.
  - ◆ If  $r$  is a relationship of  $C$ , then  $x.r$  is the value to which  $x$  is connected by  $r$ .
    - ◆ Could be an object or a set 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  $x$ .

# OQL Select-From-Where

■ We may compute relation-like collections by an OQL statement: **SELECT** <list of values>

**FROM** <list of collections and  
names for typical members>

**WHERE** <condition>

■ Each term of the FROM clause is: <collection>  
<member name>

■ A collection can be:

- ♦ The extent of some class.
- ♦ An expression that evaluates to a collection

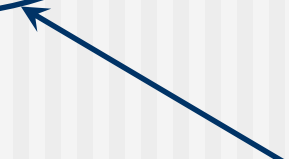
■ To change a field name, precede that term by the name and a colon.



# OQL Example

*Find the movies and cinemas such that the cinemas show more than one movie.*

```
SELECT mname: M.movieName,  
         cname: C.cinemaName  
FROM Movies M, M.shownAt C  
WHERE C.numshowing() >1
```



*Use of path expression  
C is bound to each cinema  
related to movie M by  
relationship shownAt*

# The Result Type

- As a default, the type of the result of **select-from-where** is a *Bag of Structs*.
  - Struct has one field for each term in the SELECT clause. Its name and type are taken from the last name in the path expression.
- If SELECT has only one term, technically the result is a one-field struct.
  - But a one-field struct is identified with the element itself.

## The Result Type (cont)

- Add DISTINCT after SELECT to make the result type a set, and eliminate duplicates.
- Use an ORDER BY clause, as in SQL to make the result a list of structs, ordered by whichever fields are listed in the ORDER BY clause.
  - Ascending (ASC) is the default; descending (DESC) is an option.
- Access list elements by index [1], [2],...
- Gives capability similar to SQL cursors.

# Subqueries

- A select-from-where expression can be surrounded by parentheses and used as a subquery in several ways, such as:
  - ♦ In a FROM clause, as a collection.
  - ♦ In boolean-valued expressions used in WHERE clauses (quantifiers):

**FOR ALL** *x* **IN** <collection> : <condition>

**EXISTS** *x* **IN** <collection> : <condition>

- ♦ True if and only if all (resp. at least one) member of the collection satisfy the condition

# Example

---

- *Find all names of movies that are screened in at least one cinema with a ticket price > 50.000*

```
SELECT m.name
FROM Movies m
WHERE
    EXISTS c IN m.shownAt:
    c.ticketPrice > 50.000
```

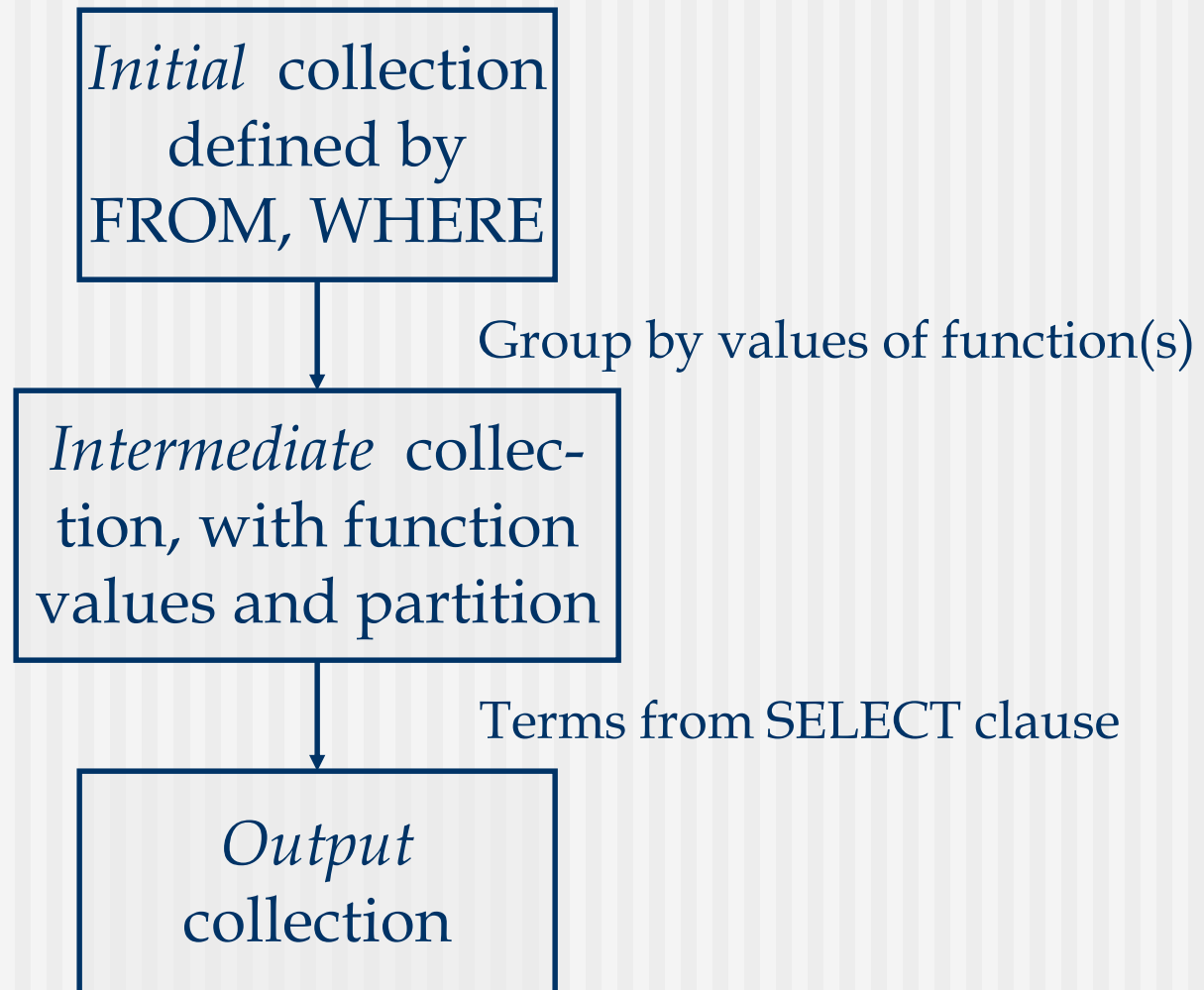
# SQL Grouping Overview

- Recall SQL grouping:
  - ◆ Groups of tuples based on the values of certain (grouping) attributes.
  - ◆ SELECT clause can extract from a group only items that make sense:
    - ◆ Aggregations within a group.
    - ◆ Grouping attributes, whose value is a constant within the group.

# OQL Grouping

- OQL extends the grouping idea in several ways:
  - ◆ Any collection may be partitioned into groups.
  - ◆ Groups may be based on any function(s) of the objects in the initial collection.
  - ◆ Result of the query can be any function of the groups.
- AVG, SUM, MIN, MAX and COUNT apply to any collection where they make sense.

# Outline of OQL GROUP BY





# GROUP BY Example

*Find the different ticket prices and the average number of movies shown at cinemas with that ticket price.*

```
SELECT C.ticketPrice,  
        avgNum:AVG(SELECT P.C.numshowing()  
                  FROM partition P)  
FROM Cinemas C  
GROUP BY C.ticketPrice
```

*Partitioning in OQL*



# GROUP BY Example: Initial Collection

- Based on **FROM** and **WHERE** (which is missing):  
**FROM Cinemas C**
- The initial collection is a *Bag* of structs with one field for each “typical element” in the **FROM** clause.
- Here, a bag of structs of the form `Struct(c: obj )`, where *obj* is a *Cinema* object.

# GROUP BY Example: Intermediate Collection

- In general, bag of structs with one component for each function in the GROUP BY clause, plus one component always called *partition*.
- The partition value is the set of all objects in the initial collection that belong to the group represented by this struct.

```
SELECT C.ticketPrice,  
        avgNum:AVG(  
    SELECT P.C.numshowing()  
    FROM partition P)  
FROM Cinemas C  
GROUP BY C.ticketPrice
```

One grouping function:

- name is *ticketPrice*,
- type is *integer*.

Intermediate collection is a set of structs with fields

- *ticketPrice* : string, and
- *partition*: Set<Struct{c: Cinema}>

## GROUP BY Example: Intermediate Collection

- A typical member of the intermediate collection in our example is:

```
Struct(ticketPrice = 50.000,  
      partition = {c1, c2, ..., cn })
```

- Each member of partition is a Cinema object *c<sub>i</sub>*, for which *c<sub>i</sub>.ticketPrice* = 50.000.

## GROUP BY Example: Output Collection

- The output collection is computed by the **SELECT** clause, as usual.
- Without a **GROUP BY** clause, the **SELECT** clause gets the initial collection from which to produce its output.
- With **GROUP BY**, the **SELECT** clause is computed from the intermediate collection.

# GROUP BY Example: Output Collection

```
SELECT C.ticketPrice, avgNum:AVG (  
  SELECT P.C.numshowing() FROM partition P)
```

Extract the *ticketPrice* field from a group's struct.

From each member  $p$  of the group's partition, get the field  $C$  (the Cinema object), and from that object extract the number of screenings.

Average these numbers to create the value of field *avgNum* in the structs of the output collection.

Typical output struct:  
Struct(ticketPrice = 50.000,  
avgNum = 9.5)

# Evolution of DBMSs

---

- Object-oriented DBMS's failed because they did not offer the efficiencies of well-entrenched relational DBMS's.
- Object-relational extensions to relational DBMS's capture much of the advantages of OO, yet retain the relation as the fundamental abstraction.

# DBMS Classification Matrix

	Simple Data	Complex Data
Query	Relational DBMS	Obj - Rel DBMS
No Query	File System	O-Oriented DBMS



# Object Relational DBMS: What's new?

- Support for storage and manipulation of large data types (BLOB and CLOB)
- Mechanisms to extend the database with application specific types and methods
  - User defined types
  - User defined procedures
  - Operators for structured types
  - Operators for reference types
- Support for inheritance

# User Defined Abstract Data Types

- A user must define methods that enable the DBMS to read in and to output objects for each new atomic type defined.
- The following methods must be registered with the DBMS:
  - Size: returns the number of bytes of storage
  - Import: creates a new object from textual input
  - Export: maps item to a printable form

```
CREATE ABSTRACT DATA TYPE jpeg_image  
(internallength =VARIABLE, input=jpeg_in,  
output=jpeg_out) ;
```

# Structured Types

- Type constructors are used to combine atomic types and user defined types to create more complex structures:
  - $\text{row}(n_1, t_1, \dots, n_n, t_n)$  : tuple of  $n$  fields
  - $\text{listof}(\text{base})$ : list of base-type items
  - $\text{array}(\text{base})$ : array of base-type items
  - $\text{setof}(\text{base})$ : set of base-type items without duplicates
  - $\text{bagof}(\text{base})$ : multiset of base-type items

# Built-In Operators for Structured Types

- Path expression
- Comparisons of sets ( $\subset \subseteq = \supset \supseteq \in \cup \cap -$ )
- Append and prepend for lists
- Postfix square bracket for arrays
- $\rightarrow$  for reference type