OBJECT-ORIENTED & OBJECT-RELATIONAL DATABASES

CS561-SPRING 2012

WPI, MOHAMED ELTABAKH

HISTORY OF DATABASES

file systems (1950s)	• store data after process created it has ceased to exist
hierarchical/ network (1960s)	• concurrency • recovery • fast access • complex structures
relational (1970-80s)	• more reliability • less redundancy • more flexibility • multiple views
ODBMS (1990s)	• better simulation • more (and complex) data types • more relationships (e.g. aggregation, specialisation) • single language for database AND programming • better versioning • no 'reconstruction' of objects • other OO advantages (reuse, inheritance etc.)

STONEBRAKER'S APPLICATION MATRIX

	No Query	Query
Complex Data	OODBMS	ORDBMS
Simple Data	File System	RDBMS

Thesis: Most applications will move to the upper right.

MOTIVATION

- **Relational model (70's):**
  - Clean and simple.
  - Great for administrative and transactional data.
  - Not as good for other kinds of **complex** data (e.g., multimedia, networks, CAD).
- **Object-Oriented models (80's):**
  - Complicated, but some influential ideas from Object Oriented
  - Complex data types.
- **Idea: Build DBMS based on OO model.**

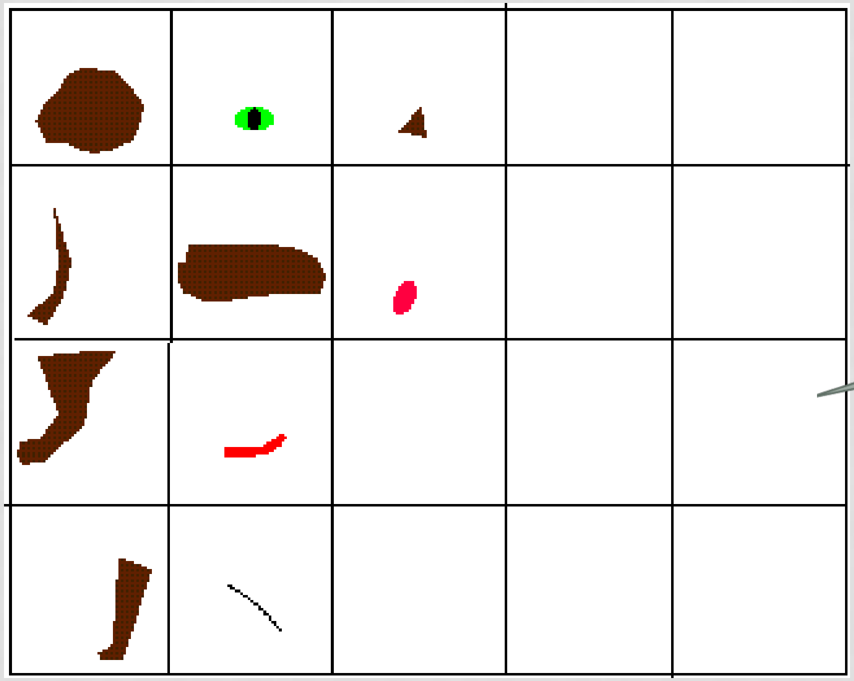
Programming languages have evolved from Procedural to Object Oriented. So why not DBMSs ???

RELATIONAL MODEL

- Relations are the key concept, everything else is around relations
- Primitive data types, e.g., strings, integer, date, etc.
- Great normalization, query optimization, and theory
- **What is missing??**
  - Handling of complex objects
  - Handling of complex data types
  - Code is not coupled with data
  - No inherence, encapsulation, etc.

RELATIONAL MODEL OF A 'CAT'

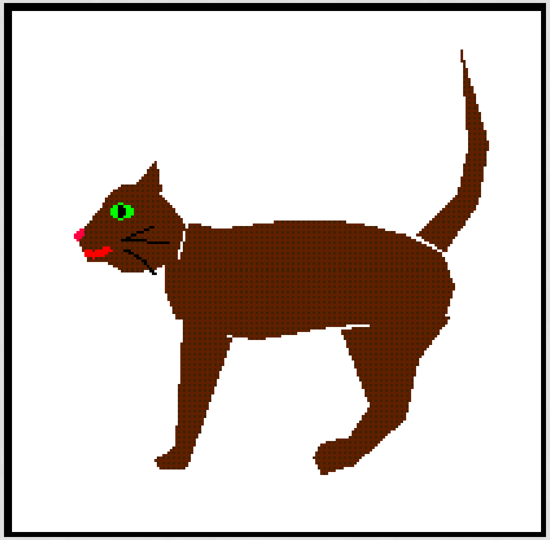
Relational database of a cat:



At query time, try to put things together as you want !!!!

OBJECT ORIENTED MODEL OF A 'CAT'

Object-oriented database of a cat:



The first areas where ODBMS were widely used were:

- CASE : Computer aided software engineering
- CAD : Computer aided design
- CAM : Computer aided manufacture

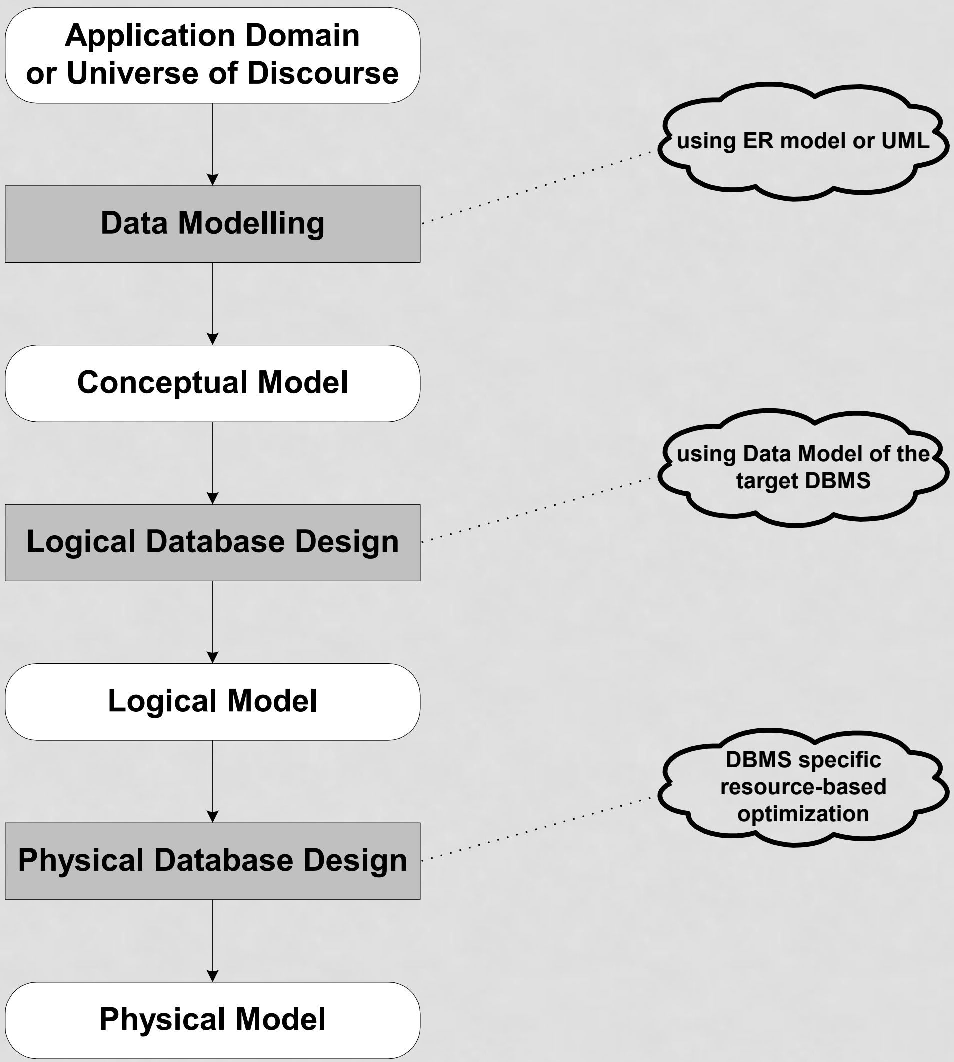
Increasingly now used in:

- telecommunications
- healthcare
- finance
- multimedia
- text/document/quality management

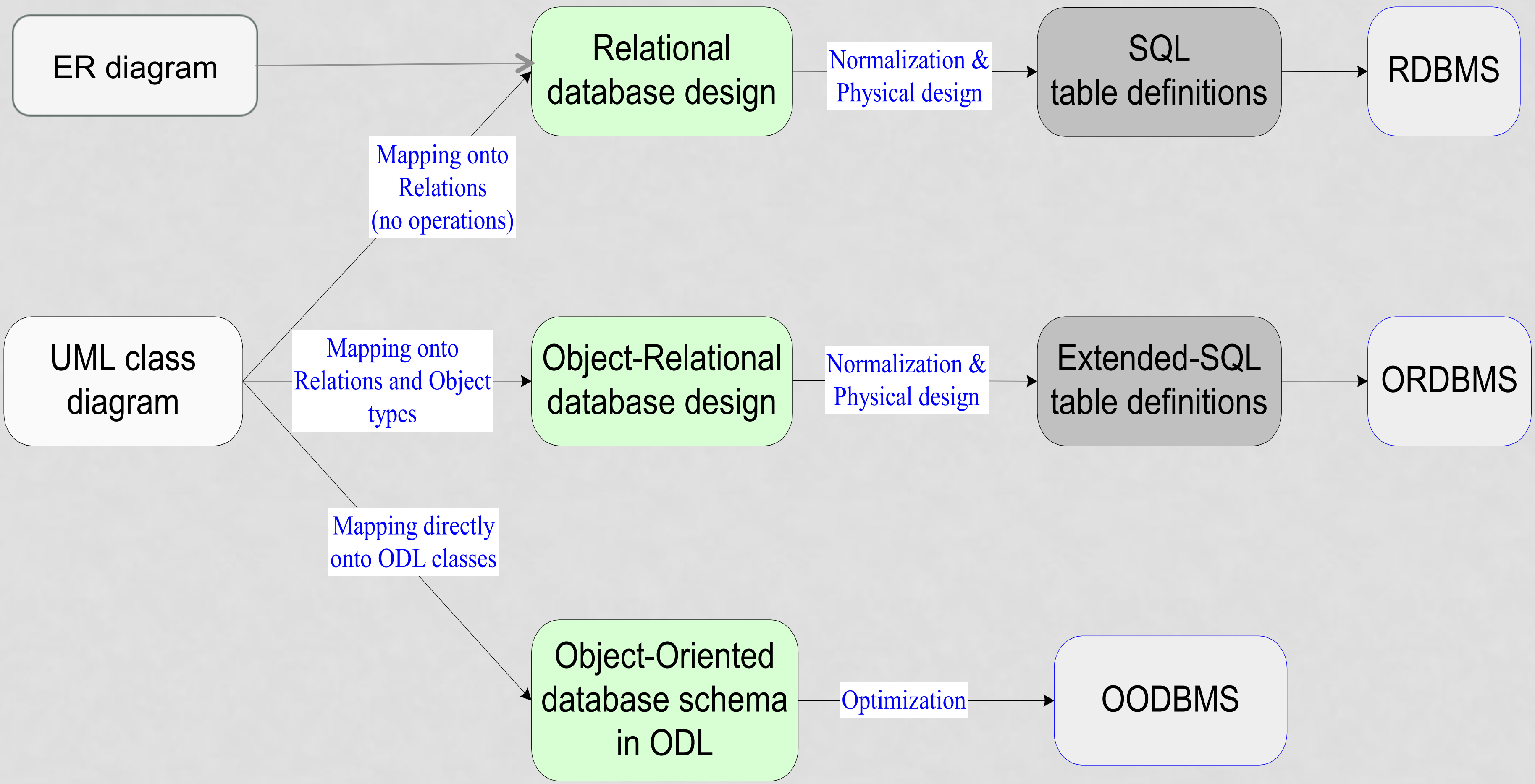
TWO APPROACHES

- **Object-Oriented Model (OODBMS)**
  - Pure OO concepts
- **Object-Relational Model (ORDBMS)**
  - Extended relational model with OO concepts

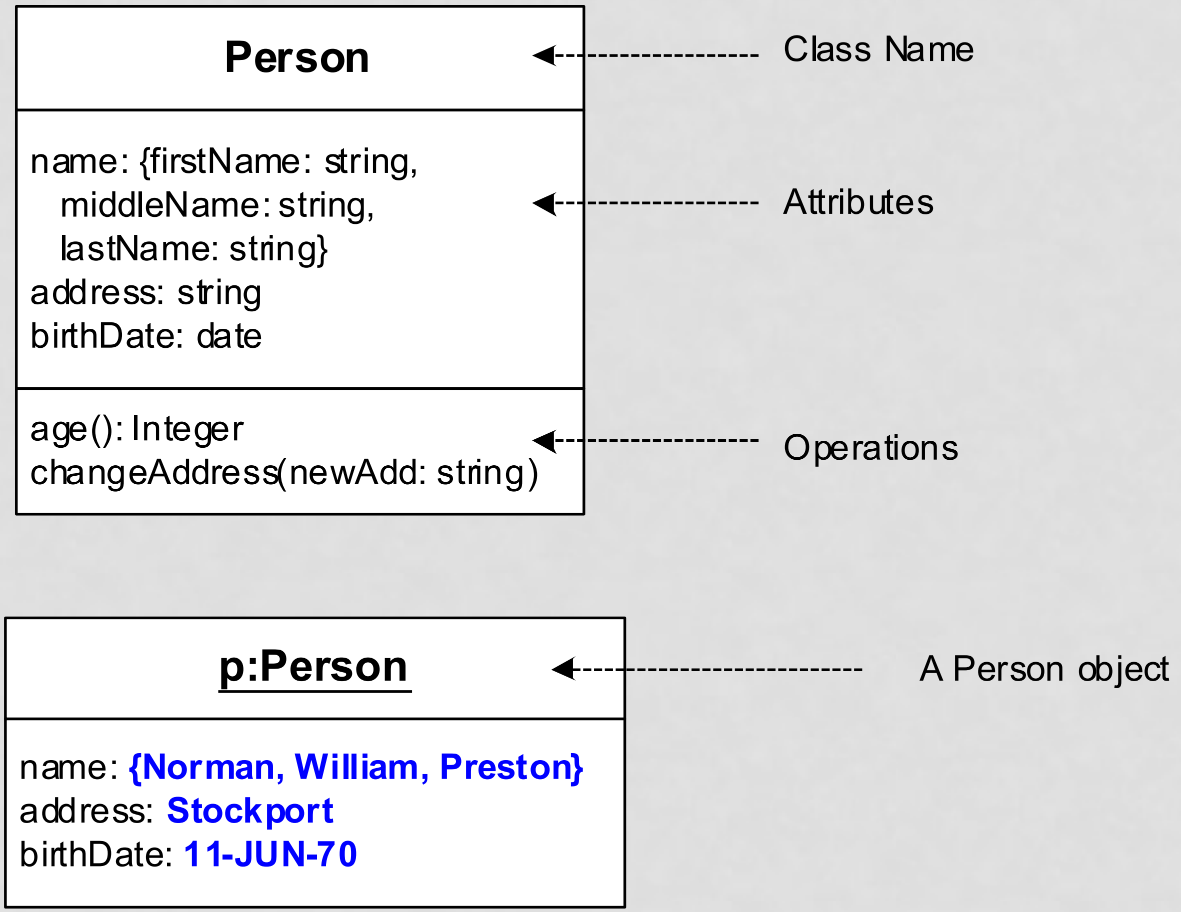
DATABASE DESIGN PROCESS



LOGICAL & PHYSICAL LAYERS



EXAMPLE OF UML CLASSES



FIRST APPROACH: OBJECT-ORIENTED MODEL

- Relations are not the central concept, **classes** and **objects** are the main concept
- Object-Oriented DBMS(OODBMS) are DBMS based on an Object-Oriented Data Model inspired by OO programming languages
- **Main Features:**
  - Powerful type system
  - Classes
  - Object Identity
  - Inheritance
- OODBMS are capable of storing complex objects, I.e., objects that are composed of other objects, and/or multi-valued attributes.

FEATURE 1: POWERFUL TYPE SYSTEM

- **Primitive types**
  - Integer, string, date, Boolean, float, etc.
- **Structure type**
  - Attribute can be a *record* with a schema

Struct {integer x, string y}

- **Collection type**
  - Attribute can be a *Set, Bag, List, Array* of other types
- **Reference type**
  - Attribute can be a *Pointer* to another object

FEATURE 2: CLASSES

- A '**class**' is in replacement of '**relation**'
- Same concept as in OO programming languages
  - All objects belonging to a same class share the same properties and behavior
- An '**object**' can be thought of as '**tuple**' (but richer content)
- Classes encapsulate data + methods + relationships
  - Unlike relations that contain data only
- In OODBMSs objects are persistency (unlike OO programming languages)

FEATURE 3: OBJECT IDENTITY

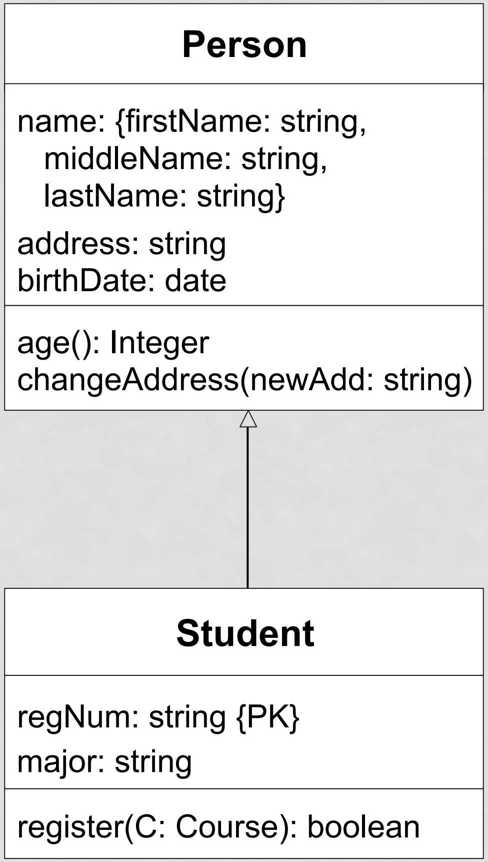
- OID is a unique identity of each object regardless of its content
  - Even if all attributes are the same, still objects have different OIDs

- Easier for references

- **An object is made of two things:**
  - **State:** attributes (name, address, birthDate of a person)
  - **Behaviour:** operations (age of a person is computed from birthDate and current date)

FEATURE 4: INHERITANCE

- A class can be defined in terms of another one.
- Person is super-class and Student is sub-class.
- Student class inherits attributes and operations of Person.



STANDARDS FOR OBJECT-ORIENTED MODEL

- **ODMG: Object Data Management Group (1991)**
  - provide a standard where previously there was none
  - support portability between products
  - standardize model, querying and programming issues

- **Language of specifying the structure of object database**
  - **ODL: Object Definition Language**
  - **OQL: Object Query Language**

- ODL is somehow similar to DDL (Data Definition Language) in SQL

Overview of ODL & OQL

ODL: CLASSES & ATTRIBUTES

Keyword *attribute*

```
1)  class Movie {
2)     attribute string title;
3)     attribute integer year;
4)     attribute integer length;
5)     attribute enum Film {color,blackAndWhite} filmType;
    };
```

Two classes with their attributes

```
1)  class Star {
2)     attribute string name;
3)     attribute Struct Addr
           {string street, string city} address;
    };
```

Attribute as a structure

ODL: RELATIONSHIPS

```
class Movie {
   attribute string title;
   attribute integer year;
   attribute integer length;
   attribute enum Film {color,blackAndWhite} filmType;
   relationship Set<Star> stars



};

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


};
```

Keyword ***relationship***

Keyword ***set***

Set: set of unsorted unique objects

Bag: set of unsorted objects with possible duplication

List: set of sorted list

Array: set of sorted list referenced by index

ODL: RELATIONSHIPS & INVERSE RELATIONSHIPS

```
1)  class Movie {
2)     attribute string title;
3)     attribute integer year;
4)     attribute integer length;
5)     attribute enum Film {color,blackAndWhite} filmType;
6)     relationship Set<Star> stars
                    inverse Star::starredIn;
7)     relationship Studio ownedBy
                    inverse Studio::owns;
    };

8)  class Star {
9)     attribute string name;
10)    attribute Struct Addr
           {string street, string city} address;
11)    relationship Set<Movie> starredIn
                    inverse Movie::stars;
    };

12) class Studio {
13)    attribute string name;
14)    attribute string address;
15)    relationship Set<Movie> owns
                    inverse Movie::ownedBy;
    };
```

Keyword *inverse*

Refers to

Inverse of each other

ODL: MULTIPLICITY OF RELATIONSHIPS

```
1)  class Movie {
2)     attribute string title;
3)     attribute integer year;
4)     attribute integer length;
5)     attribute enum Film {color,blackAndWhite} filmType;
6)     relationship Set<Star> stars
                    inverse Star::starredIn;
7)     relationship Studio ownedBy
                    inverse Studio::owns;
    };

8)  class Star {
9)     attribute string name;
10)    attribute Struct Addr
           {string street, string city} address;
11)    relationship Set<Movie> starredIn
                    inverse Movie::stars;
    };

12) class Studio {
13)    attribute string name;
14)    attribute string address;
15)    relationship Set<Movie> owns
                    inverse Movie::ownedBy;
    };
```

Based on the use of collection types (set, bag, etc.)

Many-to-Many relationship

One-to-Many relationship

What about multiway relationships???

--Not supported
--Need to convert a multiway to multiple binary relationships

ODL: METHODS

```
1)   class Movie {
2)      attribute string title;
3)      attribute integer year;
4)      attribute integer length;
5)      attribute enumeration(color,blackAndWhite) filmType;
6)      relationship Set<Star> stars
                    inverse Star::starredIn;
7)      relationship Studio ownedBy
                    inverse Studio::owns;
8)      float lengthInHours() raises(noLengthFound);
9)      void starNames(out Set<String>);
10)     void otherMovies(in Star, out Set<Movie>)
                    raises(noSuchStar);
     };
```

Three methods declarations

Parameters are either IN, OUT, or INOUT

Definition (implementation) is not part of the class

ODL: INHERITANCE

- Same Idea as in OO programming (C++ or Java)
- Subclass inherits all attributes, relationships, and methods
  - Plus adding additional fields

```
class Cartoon extends Movie {
   relationship Set<Star> voices;
};
```

Keyword *extends*

Cartoon movie is a movie with voices of characters

```
class MurderMystery extends Movie {
   attribute string weapon;
};
```

Murder movie is a movie with the weapons used

```
class CartoonMurderMystery
    extends MurderMystery : Cartoon;
```

Inherits from two other classes

ODL: INSTANCES & KEYS

- Instance of a class are all objects currently exist of that class
  - In ODL that is called **extent** (and is given a name)
- Keys are not as important for referencing objects
  - Because each object already has a unique OID
- Defining keys in ODL is optional
- ODL allows defining multiple keys (Comma separated)

```
class Movie
    (extent Movies key (title, year))
{
    attribute string title;
        ...
```

Keywords extent & *key*

The key is the pair of (title, year)

```
class Employee
   (extent Employees key (empID, ssNo))
       ...
```

The key is the pair of (empID, SSN)

```
class Employee
   (extent Employees key empID, ssNo)
       ...
```

Two keys empID and SSN

WHAT'S NEXT

- **First Approach: Object-Oriented Model**
  - Concepts from OO programming languages
  - ODL: Object Definition Language
  - What about querying OO databases???
    - **OQL: Object Oriented Query Language**

OQL: OBJECT-ORIENTED QUERY LANGUAGE

- OQL is a query language designed to operate on databases described in ODL.
- Tries to bring some concepts from the relational model to the ODBMs
  - E.g., the SELECT statement, joins, aggregation, etc.
- Reference of class properties (attributes, relationships, and methods) using:
  - Dot notation (p.a), or
  - Arrow notation (p->a)
- In OQL both notations are equivalent

OQL: EXAMPLE QUERIES I

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


class Star
    (extent Stars key name)
{
    attribute string name;
    attribute Struct Addr
        {string street, string city} address;
    relationship Set<Movie> starredIn
                 inverse Movie::stars;
};


class Studio
    (extent Studios key name)
{
    attribute string name;
    attribute string address;
    relationship Set<Movie> owns
                 inverse Movie::ownedBy;
};
```

Reference the extent (instance of class)

```
SELECT m.year
FROM Movies m
WHERE m.title = "Gone With the Wind"
```

Select the year of movie 'Gone with the wind'

For each movie m, s is the set of stars in that movie (follow a relationship)

```
SELECT s.name
FROM Movies m, m.stars s
WHERE m.title = "Casablanca"
```

Select star names from movie 'Casablanca'

Another notation

```
SELECT s.name
FROM m IN Movies, s IN m.stars
WHERE m.title = "Casablanca"
```

OQL: EXAMPLE QUERIES II

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

```
class Star
    (extent Stars key name)
{
    attribute string name;
    attribute Struct Addr
        {string street, string city} address;
    relationship Set<Movie> starredIn
                 inverse Movie::stars;
};
```

```
class Studio
    (extent Studios key name)
{
    attribute string name;
    attribute string address;
    relationship Set<Movie> owns
                 inverse Movie::ownedBy;
};
```

```
SELECT DISTINCT s.name
FROM Movies m, m.stars s
WHERE m.ownedBy.name = "Disney"
```

```
SELECT DISTINCT s.name
FROM (SELECT m
      FROM Movies m
      WHERE m.ownedBy.name = "Disney") d,
     d.stars s
```

subquery

Select distinct star names in movies owned by ‘Disney’

```
SELECT m
FROM Movies m
WHERE m.ownedBy.name = "Disney"
ORDER BY m.length, m.title
```

order movies owned by ‘Disney’ based on length and title

Report set of structures

Join two classes

```
SELECT DISTINCT Struct(star1: s1, star2: s2)
FROM Stars s1, Stars s2
WHERE s1.address = s2.address AND s1.name < s2.name
```

Report pairs of stats who have the same address

OQL OUTPUT

- Unlike SQL which produces relations, OQL produces collection (set, bag, list) of objects
  - The object can be of any type

```
SELECT DISTINCT s.name
FROM Movies m, m.stars s
WHERE m.ownedBy.name = "Disney"
```

← Set of strings

```
SELECT m
FROM Movies m
WHERE m.ownedBy.name = "Disney"
ORDER BY m.length, m.title
```

← Set of objects of type Movie

```
SELECT DISTINCT Struct(star1: s1, star2: s2)
FROM Stars s1, Stars s2
WHERE s1.address = s2.address AND s1.name < s2.name
```

← Set of structures

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

OQL: AGGREGATION

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


class Star
    (extent Stars key name)
{
    attribute string name;
    attribute Struct Addr
        {string street, string city} address;
    relationship Set<Movie> starredIn
                 inverse Movie::stars;
};


class Studio
    (extent Studios key name)
{
    attribute string name;
    attribute string address;
    relationship Set<Movie> owns
                 inverse Movie::ownedBy;
};
```

Aggregate over the partition

```
SELECT stdo, yr, sumLength: SUM(SELECT p.m.length
                                FROM partition p)
FROM Movies m
GROUP BY stdo: m.ownedBy.name, yr: m.year
HAVING MAX(SELECT p.m.length FROM partition p) > 120
```

Intermediate result

```
Struct{
      stdo: ...,
      yr: ...,
      partition: bag(struct {m: ...})
      };
```

Grouping fields

Bag of structures with members follow what's in the FROM clause

OQL: COLLECTION OPERATORS

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


class Star
    (extent Stars key name)
{
    attribute string name;
    attribute Struct Addr
        {string street, string city} address;
    relationship Set<Movie> starredIn
                 inverse Movie::stars;
};


class Studio
    (extent Studios key name)
{
    attribute string name;
    attribute string address;
    relationship Set<Movie> owns
                 inverse Movie::ownedBy;
};
```

- Like in SQL, we have ANY, ALL, EXISTS
- OQL has similar operators

```
1)  SELECT s
2)  FROM Stars s
3)  WHERE EXISTS m IN s.starredIn :
4)     m.ownedBy.name = "Disney"
```

Select stars who participated in a movie made by 'Disney'

```
SELECT s
FROM Stars s
WHERE FOR ALL m IN s.starredIn :
    m.ownedBy.name = "Disney"
```

Select stars who participated only in movies made by 'Disney'

INTEGRATING OQL & EXTERNAL LANGUAGES

- OQL fits naturally in OO host languages
- Returned objects are assigned in variables in the host program

```
oldMovies = SELECT DISTINCT m
            FROM Movies m
            WHERE m.year < 1920;
```

Variable in host language (C++ or Java)

Array of objects of type Movie

```
movieList = SELECT m
            FROM Movies m
            ORDER BY m.title, m.year;
numberOfMovies = COUNT(movieList);
for(i=0; i<numberOfMovies; i++) {
    movie = movieList[i];
    cout << movie.title << " " << movie.year << " "
         << movie.length << "\n";
}
```

Iterate over the list in a natural way

WHAT'S NEXT

- **First Approach: Object-Oriented Model**
  - Concepts from OO programming languages
  - ODL: Object Definition Language
  - What about querying OO databases???
    - OQL: Object Oriented Query Language
- **Second Approach: Object-Relational Model**

SECOND APPROACH: OBJECT-RELATIONAL MODEL

- Object-oriented model tries to bring the main concepts from relational model to the OO domain
  - The heart is OO concepts with some extensions

- Object-relational model tries to bring the main concepts from the OO domain to the relational model
  - The heart is the relational model with some extensions
  - Extensions through user-defined types

CONCEPTUAL VIEW OF OBJECT-RELATIONAL MODEL

- Relation is still the fundamental structure

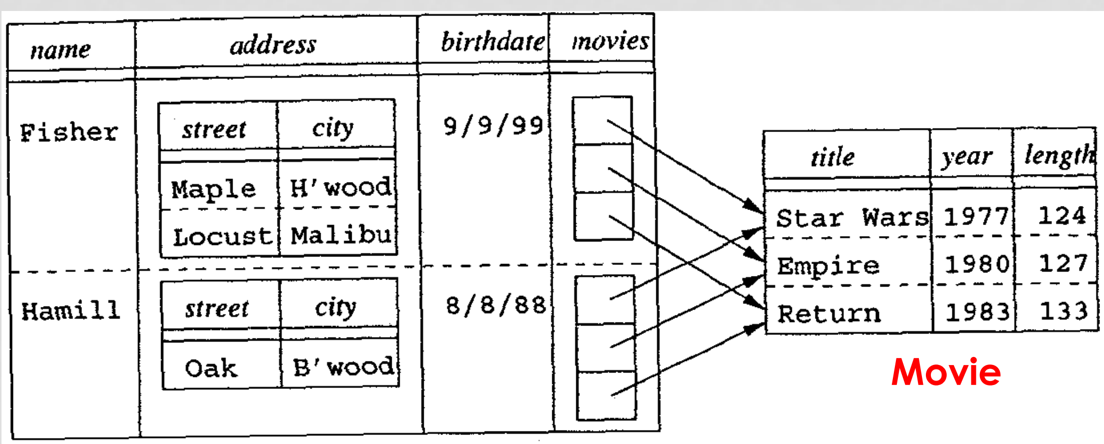
- **Relational model extended with the following features**
  - **Type system with primitive and structure types (UDT)**
    - Including set, bag, array, list collection types
    - Including structures like records
  - **Methods**
    - Special operations can be defined over the user-defined types (UDT)
    - Specialized operators for complex types, e.g., images, multimedia, etc.
  - **Identifiers for tuples**
    - Unique identifiers even for identical tuples
  - **References**
    - Several ways for references and de-references

CONCEPTUAL VIEW OF OBJECT-RELATIONAL MODEL

name	address: street	address: city	birthdate	movies: title	movies: year	movies: length
Fisher	Maple	H'wood	9/9/99	Star Wars	1977	124
	Locust	Malibu		Empire	1980	127
				Return	1983	133
Hamill	Oak	B'wood	8/8/88	Star Wars	1977	124
				Empire	1980	127
				Return	1983	133

Star(name, address(street, city), birthdate, movies(title, year, length))

- Allow of nested relations
- Repeating movies inside the stars records is redundancy
- To avoid redundancy, use pointers (references)



Star

SUPPORT FROM VENDORS

- Several major software companies including **IBM**, **Informix**, **Microsoft**, **Oracle**, and **Sybase** have all released object-relational versions of their products

- Extended SQL standards called SQL-99 or SQL3

SQL-99: QUERY LANGUAGE FOR OBJECT-RELATIONAL MODEL

- User-defied types (UDT) replace the concept of classes

- Create relations on top of the UDTs
  - Multiple relations can be created on top of the same UDT

Create Type <*name*> AS (attributes and method declarations)

CREATING UDT

```
CREATE TYPE AddressType AS (
    street  CHAR(50),
    city    CHAR(20)
);
```

Creating a type for the address of stars

```
CREATE TYPE StarType AS (
    name    CHAR(30),
    address AddressType
);
```

A hierarchy of types **(inheritance)**

```
CREATE TYPE AddressType AS (
    street  CHAR(50),
    city    CHAR(20)
    )
    METHOD houseNumber() RETURNS CHAR(10);
```

Adding a method declaration for a type (not definition) **(Encapsulation)**

```
 CREATE METHOD houseNumber() RETURNS CHAR(10)
 FOR AddressType
BEGIN
    ...
END;
```

COLLECTIONS AND LARGE OBJECTS

- **Book Type contains collections**
  - Arrays of authors → capture the order of authors
  - Set of keywords

```
create type Book as
        (title          varchar(20),
         author-array   varchar(20) array [10],
         pub-date       date,
         publisher      Publisher,
         keyword-set    setof(varchar(20)))
```

- **Large object types**
  - **CLOB:** Character large objects

    book-review **CLOB(10KB)**
  - **BLOB**: binary large objects

    image **BLOB(10MB)**

    movie **BLOB(2GB)**

Usually provide methods inside the UDT to manipulate CLOB & BLOB

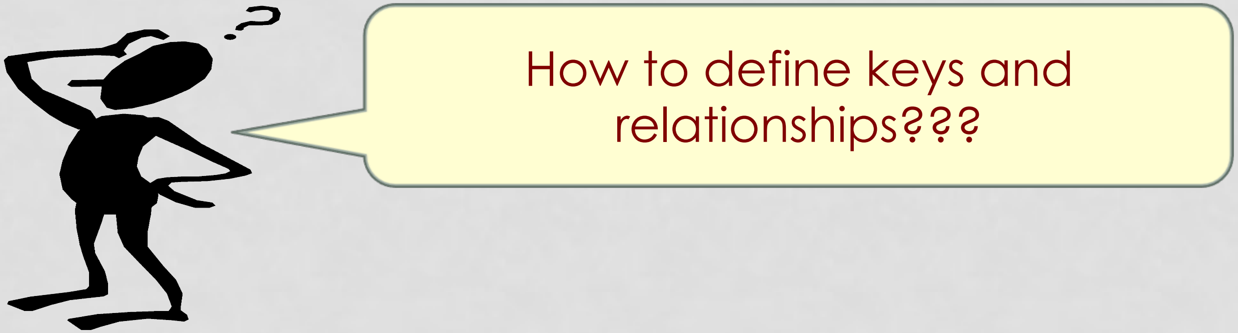
CREATING RELATIONS

- Once types are created, we can create relations

- In general, we can create tables without types
  - But types provide encapsulation, inheritance, etc.

```
CREATE TYPE StarType AS (
    name    CHAR(30),
    address AddressType
);
```

```
Create Table MovieStar OF StarType;
```



CREATING RELATIONS

- A single primary key can be defined using ***Primary Key*** keyword

- To reference another relation R, R has to be ***referenceable*** using ***REF keyword***

Tuples can be referenced using attribute movieID (system generated)

```
CREATE TYPE MovieType AS (
    title   CHAR(30),
    year    INTEGER,
    inColor BOOLEAN
);
```

Create type for movies

```
CREATE TABLE Movie OF MovieType (
    REF IS movieID SYSTEM GENERATED,
    PRIMARY KEY (title, year)
);
```

Define primary key

Create Movie table

```
CREATE TYPE StarType AS (
    name    CHAR(30),
    address AddressType
);
```

Create type for stars

```
CREATE TABLE MovieStar OF StarType (
    REF IS starID SYSTEM GENERATED
);
```

Referenceable, but no primary key

Create MovieStar table

DEFINING RELATIONSHIPS

- **One-to-many Or one-to-one**
  - Plug it inside the existing types

- **Many-to-many**
  - Create a new type or new table referencing existing types

```
CREATE TYPE StarType AS (
    name      CHAR(30),
    address   AddressType,
    bestMovie REF(MovieType) SCOPE Movie
);
```

For each star, keep the best movie (one-many)

```
CREATE TABLE StarsIn (
    star    REF(StarType) SCOPE MovieStar,
    movie   REF(MovieType) SCOPE Movie
);
```

Table for each star participated in which movies (many-many)

SCOPE points to a 'referenceable' table

WHAT'S NEXT

- **First Approach: Object-Oriented Model**
  - Concepts from OO programming languages
  - ODL: Object Definition Language
  - What about querying OO databases???
    - OQL: Object Oriented Query Language

- **Second Approach: Object-Relational Model**
  - Conceptual view
  - Data Definition Language (Creating types, tables, and relationships)
  - **Querying object-relational database (SQL-99)**

QUERYING OBJECT-RELATIONAL DATABASE

- Most relational operators work on the object-relational tables
  - E.g., selection, projection, aggregation, set operations
- Some new operators and new syntax for some existing operators
- SQL-99 (SQL3): Extended SQL to operate on object-relational databases

EXAMPLES I

```
1)  CREATE TYPE MovieType AS (
2)      title   CHAR(30),
3)      year    INTEGER,
4)      inColor BOOLEAN
    );
```

```
5)  CREATE TABLE Movie OF MovieType (
6)      REF IS movieID SYSTEM GENERATED,
7)      PRIMARY KEY (title, year)
    );
```

```
CREATE TYPE StarType AS (
    name      CHAR(30),
    address   AddressType,
    bestMovie REF(MovieType) SCOPE Movie
);
```

```
CREATE TABLE MovieStar OF StarType (
    REF IS starID SYSTEM GENERATED
);
```

```
CREATE TABLE StarsIn (
    star    REF(StarType) SCOPE MovieStar,
    movie   REF(MovieType) SCOPE Movie
);
```

Q1: Find the year of movie 'King Kong'

```
Select m.year
From  Movie m
Where m.title = 'King Kong';
```

Variable *m* is important to reference the fields

Q2: Find the title of the best movie 'Jim Carry'

```
Select s.bestMovie->title
From  MovieStar s
Where s.name = 'Jim Carry';
```

Follow a reference (pointer) using → operator

EXAMPLES II: DE-REFERENCING

```
1)  CREATE TYPE MovieType AS (
2)      title   CHAR(30),
3)      year    INTEGER,
4)      inColor BOOLEAN
    );
```

```
5)  CREATE TABLE Movie OF MovieType (
6)      REF IS movieID SYSTEM GENERATED,
7)      PRIMARY KEY (title, year)
    );
```

```
CREATE TYPE StarType AS (
    name      CHAR(30),
    address   AddressType,
    bestMovie REF(MovieType) SCOPE Movie
);
```

```
CREATE TABLE MovieStar OF StarType (
    REF IS starID SYSTEM GENERATED
);
```

```
CREATE TABLE StarsIn (
    star    REF(StarType) SCOPE MovieStar,
    movie   REF(MovieType) SCOPE Movie
);
```

Q3: Find movies starred by ‘Jim Carry’

```
Select DEREF(movie)
From  StarsIn
Where star->name = ‘Jim Carry’;
```

DEREF: Get the tuple pointed to by the given pointer

Q4: Find movies starred by ‘Jim Carry’ (Another way)

```
Select s.movie->title, s.movie->year, s.movie->inColor,
From  StarsIn s
Where s.star->name = ‘Jim Carry’;
```

*** Using a variable for StartsIn (s in Q4) is not necessary because the table is not based on type.

EXAMPLES III: COMPARISON

```
CREATE TYPE MovieType AS (
    title   CHAR(30),
    year    INTEGER,
    inColor BOOLEAN
);
```

```
CREATE TABLE Movie OF MovieType (
    REF IS movieID SYSTEM GENERATED,
    PRIMARY KEY (title, year)
);
```

```
CREATE TYPE StarType AS (
    name      CHAR(30),
    address   AddressType,
    bestMovie REF(MovieType) SCOPE Movie
);
```

```
CREATE TABLE MovieStar OF StarType (
    REF IS starID SYSTEM GENERATED
);
```

```
CREATE TABLE StarsIn (
    star   REF(StarType) SCOPE MovieStar,
    movie  REF(MovieType) SCOPE Movie
);
```

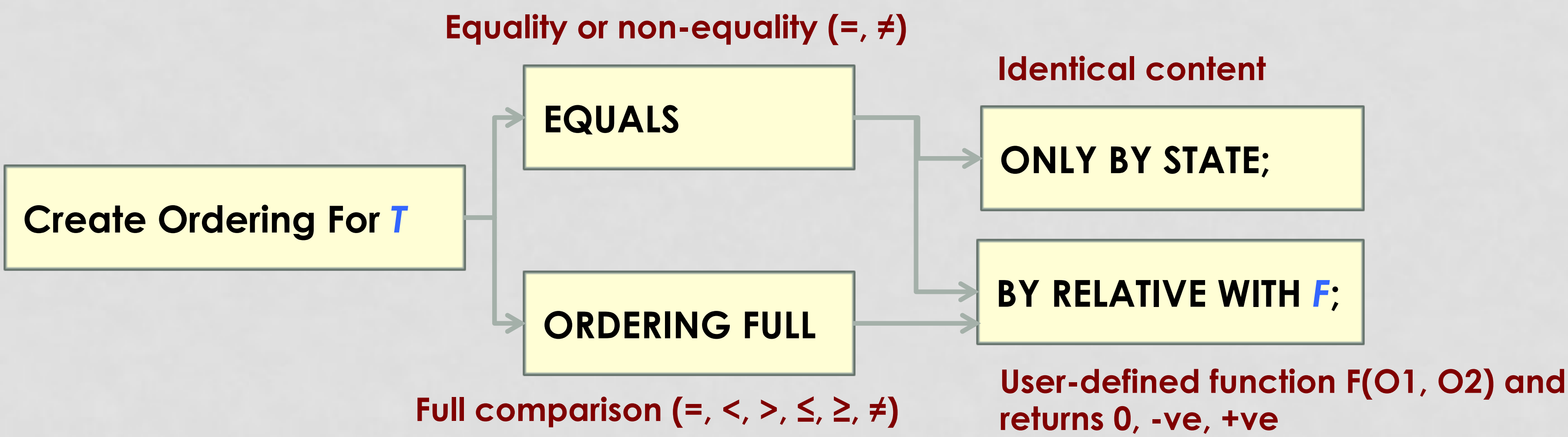
Q5: Find distinct movies starred by ‘Jim Carry’ or ‘Mel Gibson’

```
Select Distinct DEREF(movie)
From  StarsIn
Where star->name = ‘Jim Carry’
Or star->name = ‘Mel Gibson’;
```

- That is wrong because all objects of type MovieType are unique even if they have the same content
- Need a mechanism to define how objects compare to each other
(needed for any comparison, e.g., ordering, duplicate elimination, grouping, etc.)

ORDERING RELATIONSHIPS

- Need to define how to compare objects of a given type *T*



ORDERING FUNCTION

```
CREATE TYPE MovieType AS (
    title   CHAR(30),
    year    INTEGER,
    inColor BOOLEAN
);
```

```
CREATE TABLE Movie OF MovieType (
    REF IS movieID SYSTEM GENERATED,
    PRIMARY KEY (title, year)
);
```

```
CREATE TYPE StarType AS (
    name      CHAR(30),
    address   AddressType,
    bestMovie REF(MovieType) SCOPE Movie
);
```

```
CREATE TABLE MovieStar OF StarType (
    REF IS starID SYSTEM GENERATED
);
```

```
CREATE TABLE StarsIn (
    star    REF(StarType) SCOPE MovieStar,
    movie   REF(MovieType) SCOPE Movie
);
```

```
CREATE ORDERING FOR AddressType
ORDER FULL BY RELATIVE WITH AddrLEG;
```

```
CREATE FUNCTION AddrLEG(
    x1 AddressType,
    x2 AddressType
) RETURNS INTEGER

IF x1.city() < x2.city() THEN RETURN(-1)
ELSEIF x1.city() > x2.city() THEN RETURN(1)
ELSEIF x1.street() < x2.street() THEN RETURN(-1)
ELSEIF x1.street() = x2.street() THEN RETURN(0)
ELSE RETURN(1)
END IF;
```

EXAMPLES IV: COMPARISON

```
CREATE TYPE MovieType AS (
    title   CHAR(30),
    year    INTEGER,
    inColor BOOLEAN
);
```

```
CREATE TABLE Movie OF MovieType (
    REF IS movieID SYSTEM GENERATED,
    PRIMARY KEY (title, year)
);
```

```
CREATE TYPE StarType AS (
    name      CHAR(30),
    address   AddressType,
    bestMovie REF(MovieType) SCOPE Movie
);
```

```
CREATE TABLE MovieStar OF StarType (
    REF IS starID SYSTEM GENERATED
);
```

```
CREATE TABLE StarsIn (
    star    REF(StarType) SCOPE MovieStar,
    movie   REF(MovieType) SCOPE Movie
);
```

Create Ordering For MovieType Equals Only By State;

Q5: Find distinct movies starred by 'Jim Carry' or 'Mel Gibson'

```
Select Distinct DEREF(movie)
From  StarsIn
Where star->name = 'Jim Carry'
Or star->name = 'Mel Gibson';
```

EXAMPLES V: GROUPING & NESTING

```
1)  CREATE TYPE MovieType AS (
2)      title   CHAR(30),
3)      year    INTEGER,
4)      inColor BOOLEAN
    );
```

```
5)  CREATE TABLE Movie OF MovieType (
6)      REF IS movieID SYSTEM GENERATED,
7)      PRIMARY KEY (title, year)
    );
```

```
CREATE TYPE StarType AS (
    name      CHAR(30),
    address   AddressType,
    bestMovie REF(MovieType) SCOPE Movie
);
```

```
CREATE TABLE MovieStar OF StarType (
    REF IS starID SYSTEM GENERATED
);
```

```
CREATE TABLE StarsIn (
    star    REF(StarType) SCOPE MovieStar,
    movie   REF(MovieType) SCOPE Movie
);
```

Q6: Find stars who participated in less than 10 movies

```
Select DEREF(star)
From  StarsIn
Group by DEREF(star)
Having count(movie) < 10;
```

Create at least an equality ordering on StarType

Q7: Find movie titles in 2000 where 'Jim Carry' is not in

```
Select m
From  Movie m
Where m.year = 2000
And   m.title Not In (
              Select movie->title
              From  StarsIn
              Where star->name = 'Jim Carry'
              And movie->year = 2000);
```

QUERYING COLLECTIONS & ARRAYS

```
create type Book as
        (title          varchar(20),
         author-array   varchar(20) array [10],
         pub-date       date,
         publisher      Publisher,
         keyword-set    setof(varchar(20)))
```

To get a relation containing pairs of the form "title, author-name" for each book and each author of the book

```
select B.title, A
    from books as B, unnest (B.author-array) as A
```

find all books that have the word "database" as one of their keywords

```
select title
    from books
    where 'database' in (unnest(keyword-set))
```

Unnest returns a relation

Get 1st and 2nd authors of certain book

```
select author-array[1], author-array[2]
    from books
    where title = `Database System Concepts'
```

GENERATORS AND MUTATORS

- How to insert new new data into tables

- **Generators**
  - Like the constructors in OO programming
  - Create new objects

- **Mutators**
  - Modify the value of an existing object

- For each attribute x in UDT *T*, the system automatically creates:
  - Generator ***T()*** that returns an empty object of T
  - Mutator ***x(v)*** that sets the value of attribute x to value v

EXAMPLE

```
CREATE TYPE MovieType AS (
    title   CHAR(30),
    year    INTEGER,
    inColor BOOLEAN
);
```

```
CREATE TABLE Movie OF MovieType (
    REF IS movieID SYSTEM GENERATED,
    PRIMARY KEY (title, year)
);
```

```
CREATE TYPE StarType AS (
    name      CHAR(30),
    address   AddressType,
    bestMovie REF(MovieType) SCOPE Movie
);
```

```
CREATE TABLE MovieStar OF StarType (
    REF IS starID SYSTEM GENERATED
);
```

```
CREATE TABLE StarsIn (
    star    REF(StarType) SCOPE MovieStar,
    movie   REF(MovieType) SCOPE Movie
);
```

```
CREATE PROCEDURE InsertStar(
    IN s CHAR(50),
    IN c CHAR(20),
    IN n CHAR(30)
)
DECLARE newAddr AddressType;
DECLARE newStar StarType;

BEGIN
    SET newAddr = AddressType();
    SET newStar = StarType();
    newAddr.street(s);
    newAddr.city(c);
    newStar.name(n);
    newStar.address(newAddr);
    INSERT INTO MovieStar VALUES(newStar);
END;
```

```
CALL InsertStar('345 Spruce St.', 'Glendale', 'Gwyneth Paltrow');
```

If DBMS allows creating generators with parameters

```
INSERT INTO MovieStar VALUES(
    StarType('Gwyneth Paltrow',
        AddressType('345 Spruce St.', 'Glendale')));
```

CREATING RECORDS OF COMPLEX TYPES

- **Collection and array types**

```
create type Book as
        (title          varchar(20),
         author-array   varchar(20) array [10],
         pub-date       date,
         publisher      Publisher,
         keyword-set    setof(varchar(20)))
```

Array construction

array [‘Silberschatz’ ,`Korth’ ,`Sudarshan’]

Set value attributes

set(v1, v2, …, vn)

To insert the preceding tuple into the relation *books*

insert into *books* **values**
(`Compilers’ , **array**[`Smith’ ,`Jones’], null,
Publisher(‘McGraw Hill’ ,`New York’),
set(`parsing’ ,`analysis’))

WHAT WE COVERED

- **First Approach: Object-Oriented Model**
  - Concepts from OO programming languages
  - ODL: Object Definition Language
  - What about querying OO databases???
    - OQL: Object Oriented Query Language

- **Second Approach: Object-Relational Model**
  - Conceptual view
  - Data Definition Language (Creating types, tables, and relationships)
  - Querying object-relational database (SQL-99)

Make use of the interesting features of Object-Oriented into database systems → ODBMSs

WHEN TO CONSIDER OODBMS OR ORDBMS

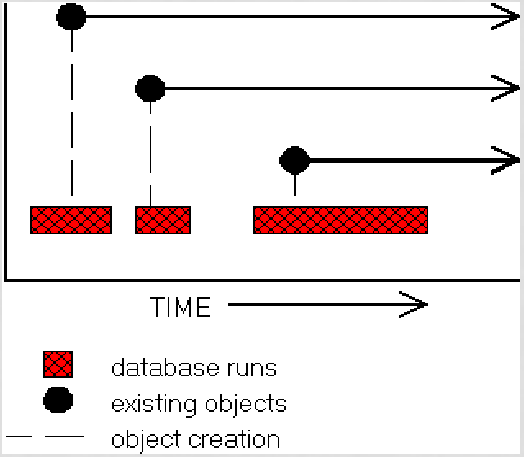
- **Complex Relationships**
  - A lot of many-to-many relationships, tree structures or network (graph) structures.
- **Complex Data**
  - Multi-dimensional arrays, nested structures, or binary data, images, multimedia, etc.
- **Distributed Databases**
  - Need for free objects without the rigid table structure.
- **Repetitive use of Large Working Sets of Objects**
  - To make use of inheritance and reusability
- **Expensive Mapping Layer**
  - Expensive decomposition of objects (normalization) and re-composition at query time

KEY BENEFITS OF ODBMS

- **Persistence & Versioning**
  - Created objects are maintained across different database runs (persistent)
  - Different evolving copies of the same object can be created over time (versioning)

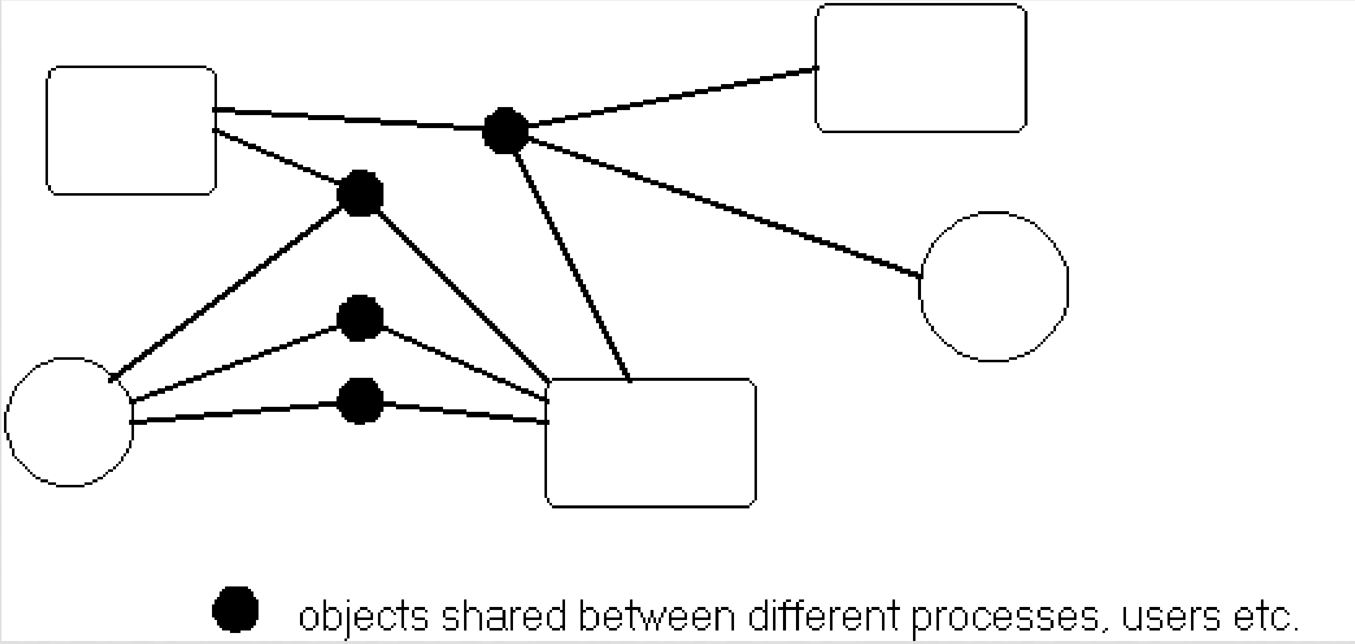
PersistentObject Superclass Approach

- ***Superclass encapsulates any class for storage and retrieval***
- ***This superclass implements all functionalities of read/write operations***



KEY BENEFITS OF ODBMS (CONT'D)

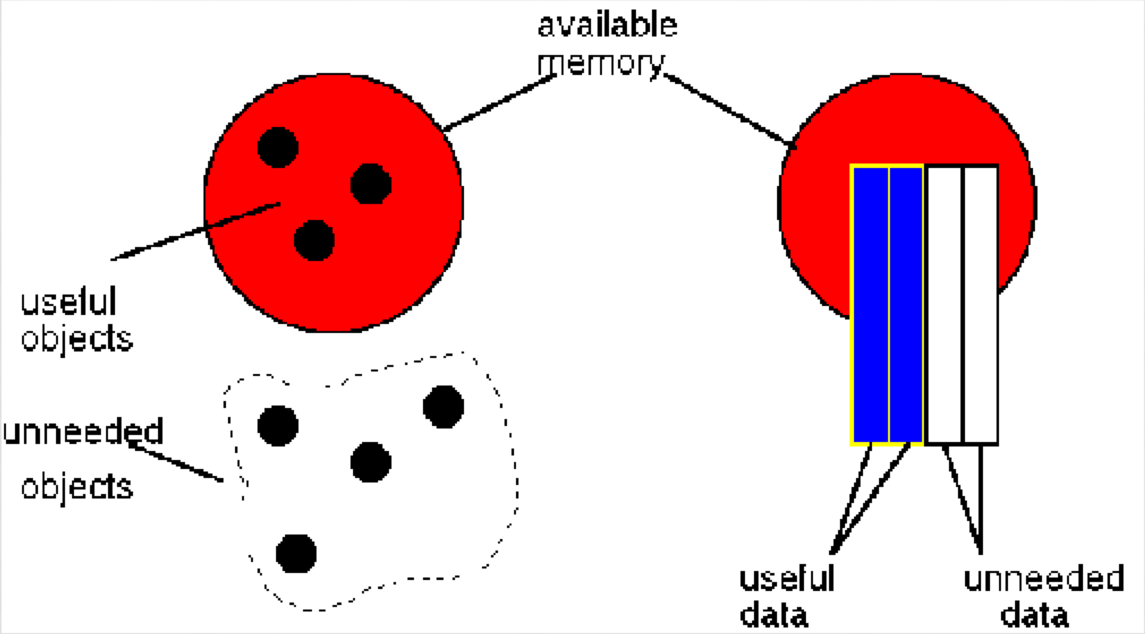
- **Sharing in highly distributed environment**
  - Easier to share and distribute objects than tables



KEY BENEFITS OF ODBMS (CONT'D)

- **Better memory usage and less paging**
  - Bringing only objects of interest

ODBMS **Relational DBMS**



OBJECT-ORIENTED VS. OBJECT-RELATIONAL

- **Object-oriented DBMSs**
  - Did not achieve much success (until now) in the market place
  - No query support (Indexing, optimization)
  - No security layer

- **Object-relational DBMSs**
  - Better support from big vendors
  - Tries to make use of all advances in RDBMSs
    - Indexes, views, triggers, query optimizations, security layer, etc.
    - **Work in progress --- Long way to go**

MODIFICATIONS TO RDBMS

- **Parsing**
  - Type-checking for methods pretty complex
- **Query Rewriting**
  - New rewriting rules including complex types and collections
- **Optimization**
  - New algebra operators needed for complex types.
  - Must know how to integrate them into optimization.
  - WHERE clause exprs can be expensive!
    - Selection pushdown may be a bad idea.

MODIFICATIONS TO RDBMS (CONT'D)

- **Execution**
  - New algebra operators for complex types.
  - OID generation & reference handling.
  - Dynamic linking and overriding.
  - Support objects bigger than 1 page.
  - Caching of expensive methods.

- **Access Methods**
  - Indexes on methods, not just columns.
  - Indexes over collection hierarchies.
  - Need indexes for new WHERE clause exprs (not just <, >, =)

- **Data Layout**
  - Clustering of nested objects.
  - Chunking of arrays.

COMPARISON

Table 2

A Comparison of Database Management Systems

Criteria	**RDBMS**	**ORDBMS**	**ODBMS**
Defining standard	SQL2 (ANSI X3H2)	SQL3/4 (in process)	ODMG-V2.0
Support for object-oriented programming	Poor; programmers spend 25% of coding time mapping the program object to the database	Limited mostly to new data types	Direct and extensive
Simplicity of use	Table structures easy to understand; many end-user tools available	Same as RDBMS, with some confusing extensions	OK for programmers; some SQL access for end users
Simplicity of development	Provides independence of data from application, good for simple relationships	Provides independence of data from application, good for simple relationships	Objects are a natural way to model; can accommodate a wide variety of types and relationships
Extensibility and content	None	Limited mostly to new data types	Can handle arbitrary complexity; users can write methods and on any structure
Complex data relationships	Difficult to model	Difficult to model	Can handle arbitrary complexity; users can write methods and on any structure