Chapter 2
Object and Object Relational Databases

- Object Database Concepts
- Object Database Extensions to SQL
- The ODMG Object Model and the Object Definition Language ODL
- Object Database Conceptual Design
- Object Query Language OQL
- Language Binding in the ODMG Standard

Introduction

- Database systems that were based on the **object data model** were known originally as **object-oriented databases (OODBs) or simply object databases (ODBs).**
- **In object-oriented design, the data and behavior of a system are encapsulated into objects.** These objects have attributes (data) and methods (behavior) that define their properties and actions. The object data model defines how these objects are structured, organized, and interconnected within a system.
- **Traditional data models and systems, such as network, hierarchical, and relational** have been quite successful in developing the database technologies required for many traditional business database applications. However, they have certain shortcomings when more complex database applications must be designed and implemented—for example, databases for engineering design and manufacturing (CAD/CAM and CIM1), biological and other sciences, telecommunications, geographic information systems, and multimedia.

  Feature of Object Database

  1. A key feature of object databases is the power they give the designer to specify both the **structure of complex objects and the operations t**hat can be applied to these objects.
  2. Another reason for the creation of object-oriented databases is **the vast increase in the use of object-oriented programming languages** for developing software applications.

- Unlike other databases like relational, hierarchical, network, Object databases are designed so they can be directly—or seamlessly—integrated with software that is developed using object-oriented programming languages like java, c#, python etc.

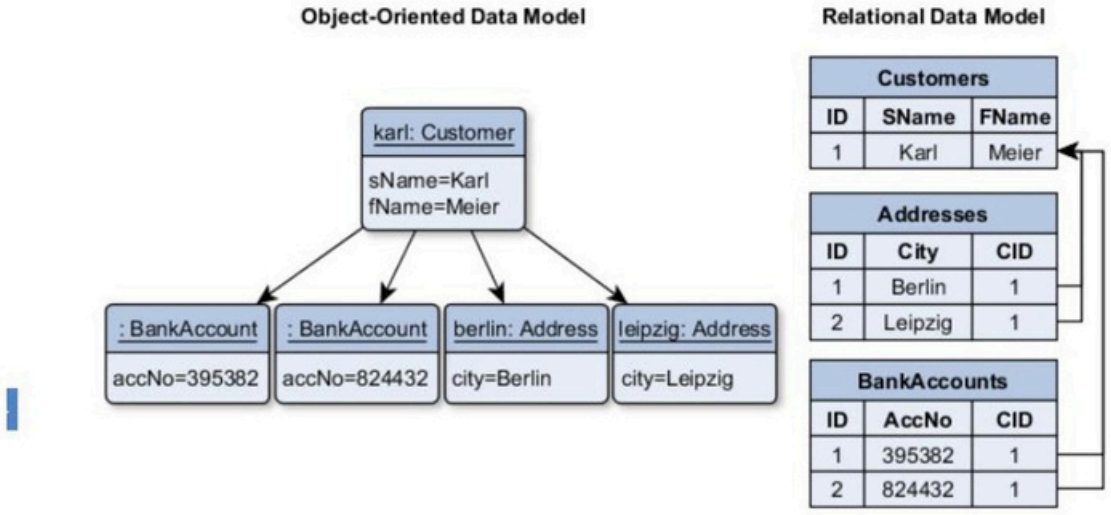


Fig: Data Representation in OO Data model vs Relation Data Model

- **Relational DBMS (RDBMS)** vendors have also recognized the need for incorporating features that were proposed for object databases, and newer versions of relational systems have incorporated many of these features. This has led to database systems that are characterized as **object relational or ORDBMSs.**

- As commercial object DBMSs became available, the need for a standard model and language was recognized. DBMS vendors and users, called **ODMG(Object Data Management Group),** proposed **a standard** whose current specification is known as the ODMG 3.0 standard.
- Object-oriented databases have adopted many of the concepts that were developed originally for object-oriented programming languages. **Concepts include object identity, object structure and type constructors, encapsulation, inheritance etc.**

Object Database Concept

Introduction to object-oriented concept and feature

- The term object-oriented—abbreviated OO or O-O—has its origins in OO programming languages, or OOPLs.
- An object typically has two components: state (attributes) and behavior (operations or methods). It can have a complex data structure as well as specific operations defined by the programmer.

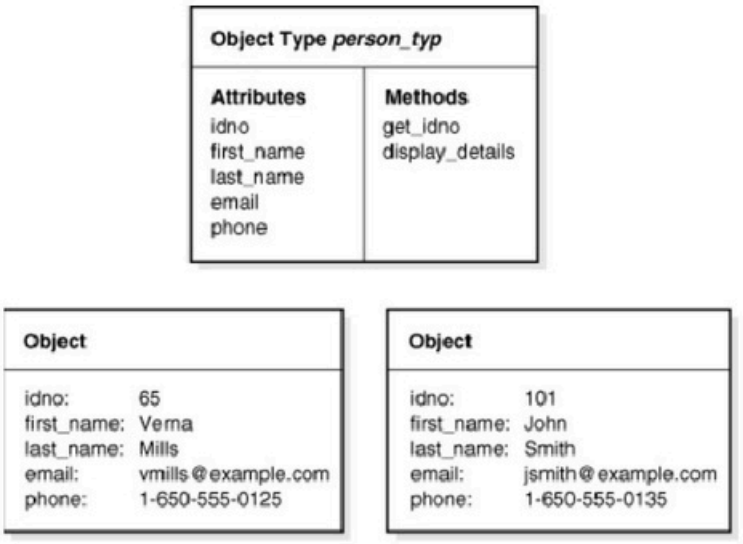


Fig: Representation of Student Object

- The internal structure of an object in OOPLs includes the specification of instance variables, which hold the values that define the internal state of the object.
- An attribute or state variable is similar to the concept of an attribute in the relational model, except that attribute variables may be encapsulated within the object and thus are not necessarily visible to external users.'
- Object types sometime also called as class serve as blueprints or templates that define both structure and behavior. Object types are database schema and application code can retrieve and manipulate these objects.
- A variable of an object type is an instance of the type, or an object.
- We can use the **CREATE TYPE** SQL statement to define object types.

Key Features in OO System

- **Encapsulation:** To encourage **encapsulation,** an operation is defined in two parts. The first part, called the signature or interface of the operation, specifies the operation name and arguments (or parameters). The second part, called the method or body, specifies the implementation of the operation, usually written in some general-purpose programming language. Operations can be invoked by passing a message to an object, which includes the operation name and the parameters. The object then executes the method for that operation.

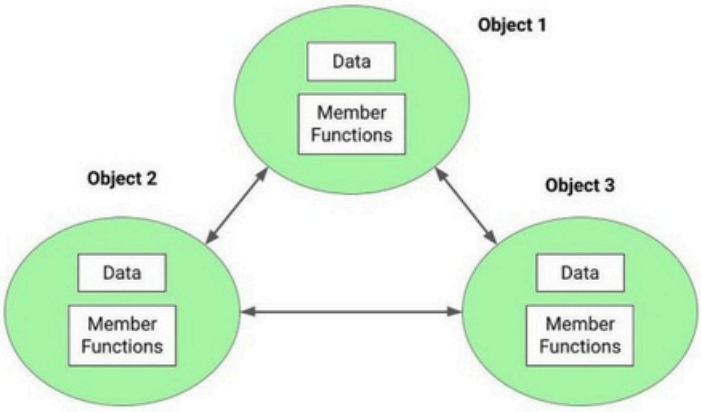


Fig: Encapsulation (Data and Function) and Message Passing

- **Inheritance:** Another key concept in OO systems is that of type and class hierarchies and **inheritance.** This permits specification of new types or classes that inherit much of their structure and/or operations from previously defined types or classes.

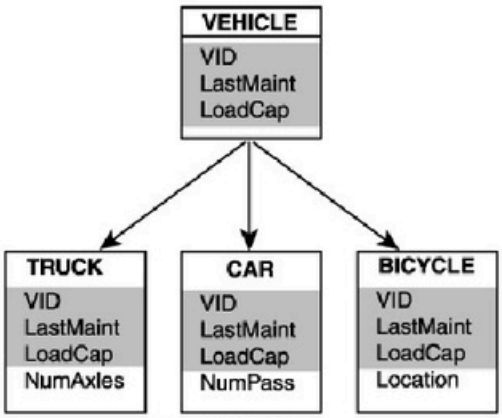


Fig: Inheritance

This makes it easier to develop the data types of a system incrementally and to reuse existing type definitions when creating new types of objects.

- **Overloading:** Another OO concept is operator **overloading**, which refers to an operation's ability to be applied to different types of objects; in such a situation, an operation name may refer to several distinct implementations, depending on the type of object it is applied to. This feature is also called **operator polymorphism.** For example, an operation to calculate the area of a geometric object may differ in its method (implementation), depending on whether the object is of type triangle, circle, or rectangle as shown in figure below.

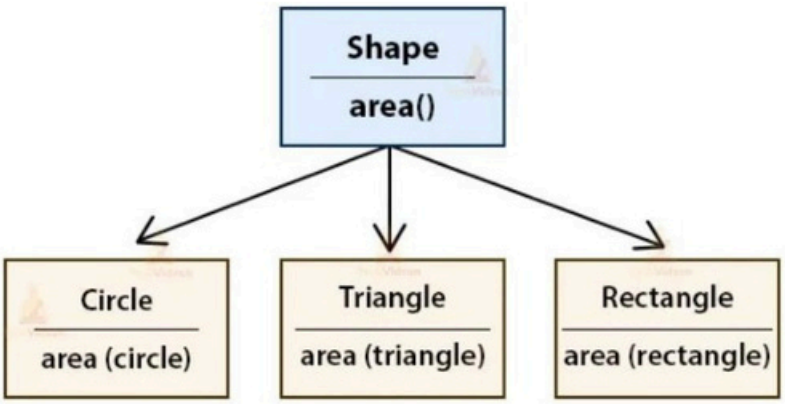


Fig: Polymorphism

Object Identity, and Objects

- **Object identity** is a property of an object which **distinguishes it from all other objects.** One goal of an ODB is to maintain a direct correspondence between real-world and database objects so that objects do not lose their integrity and identity and can easily be identified and operated upon. Hence, **a unique identity is assigned to**

each independent object stored in the database. **This unique identity is typically implemented via a unique, system-generated object identifier (OID)**. The value of an OID may not be visible to the external user but is used internally by the system to **identify each object uniquely and to create and manage inter object references.**

Two main Properties of OID

1. The main property required of an OID is that it be **immutable;** that is, the OID value of a particular object should not change Hence, an ODMS (Object Database Management System) must have some mechanism for generating OIDs and preserving the immutability property.
2. It is also desirable that each OID be used **only once;** that is, even if an object is removed from the database, its OID should not be assigned to another object.

- These two properties imply that the OID should not depend on any attribute values of the object, since the value of an attribute may be changed or corrected.
- We can compare this with the **relational model, where each relation must have a primary key attribute, whose value identifies each tuple uniquely**. If the value of the primary key is changed, the tuple will have a new identity, even though it may still represent the same real-world object. Alternatively, a real-world object may have different names for key attributes in different relations, making it difficult to establish that the keys represent the same real-world object (for example, using the Emp_id of an EMPLOYEE in one relation and the Ssn in another).
- Some early ODMSs have used the physical address as the OID to increase the efficiency of object retrieval. However, it is sometime inappropriate to base the OID on the physical address of the object in storage, since the physical address can change after a physical reorganization of the database. So, if the physical address of the object changes, an indirect pointer can be placed at the former address, which gives the new physical location of the object.

Complex type structure for object and literal

- OO data models may need to represent a simple value (integer, string, Boolean value etc.) or literals to complex object. Every object must have an immutable OID, whereas a literal value has no OID and its value just stands for itself. Thus, a literal value is typically stored within an object and cannot be referenced from other objects.
- Another feature of ODBs is that objects and literals may have a **type structure of arbitrary complexity** in order to contain all of the necessary information that describes the object or literal.
- In contrast, in traditional database systems, information about a complex object is often scattered over many relations or records, leading to loss of direct correspondence between a real-world object and its database representation.
- In ODBs, a complex type may be constructed from other types by nesting of type constructors.
- The three most basic constructors are atom, struct (or tuple), and collection.

a. Atom Constructor

- One type of constructor is atom constructor which includes the basic built-in data types of the object model, which are similar to the basic types in many programming languages.
- The atom constructor is used to represent all basic atomic values, such as integers, real numbers, character strings, Booleans, and any other basic data types that the system supports directly.
- These basic data types are called single valued or atomic types, since each value of the type is considered an atomic (indivisible) single value.

a. Struct(tuple) Constructor.

- A second type of constructor is referred to as the struct (or tuple) constructor which can create standard structured types, such as the tuples (record types) in the basic relational model.
- A structured type is made up of several components and is also sometimes referred to as a compound or composite type.
- The struct constructor is not considered to be a type, but rather a type generator, because many different structured types can be created.
- The tuple constructor can create structured values and objects of the form <a1:i1, a2:i2, … , an:in>, where each aj is an attribute name and each ij is a value or OID.

- For example, two different structured types that can be created are:
  struct Name<FirstName: string, MiddleInitial: char, LastName: string>, and
  struct CollegeDegree<Major: string, Degree: string, Year: date>.
- To create complex nested type structures in the object model, the collection type constructors are needed.

b. Collection Constructor

- Collection type constructors include **the set(T), list(T), bag(T), array(T), and dictionary(K,T)** type constructors.
- The **set constructor** will create objects or literals that are a set of distinct elements {i1, i2, … , in}, all of the same type.
- The **bag constructor** (also called a multiset) is similar to a set except that the elements in a bag need not be distinct.
- The **list constructor** will create an ordered list [i1, i2, … , in] of OIDs or values of the same type. A list is similar to a bag except that the elements in a list are ordered, and hence we can refer to the first, second, or jth element.
- The **array constructor** creates a single-dimensional array of elements of the same type. The main difference between array and list is that a list can have an arbitrary number of elements whereas an array typically has a maximum size.
- The **dictionary constructor** creates a collection of key-value pairs (K, V), where the value of a key K can be used to retrieve the corresponding value V.
- Collection constructors allows part of an object or literal value to include a collection of other objects or values when needed. These constructors are also considered to be type generators because many different types can be created. For example, set(string), set(integer), and set(Employee) are three different types that can be created from the set type constructor. All the elements in a particular collection value must be of the same type. For example, all values in a collection of type set(string) must be string values.

- An **object definition language (ODL)** that incorporates the preceding type constructors can be used to define the object types for a particular database application.
- **The type constructors can be used to define the data structures for an OO database schema.** Figure below shows how we may declare EMPLOYEE and DEPARTMENT types.

```
define type EMPLOYEE
        tuple ( full_name: string;
                Birth_date: DATE;
                Address: string;
                Supervisor: EMPLOYEE;
                Dept: DEPARTMENT;
              );

define type DATE
        tuple ( Year: integer;
                Month: integer;
                Day: integer;
              );

define type DEPARTMENT
        tuple ( Dname: string;
                Dnumber: integer;
                Mgr: tuple ( Manager: EMPLOYEE;Start_date: DATE; );
                Locations: set(string);
                Employees: set(EMPLOYEE);
              );
```

Fig: Specifying the object types EMPLOYEE, DATE, and DEPARTMENT using type constructor

- In the above figure, the attributes that refer to other objects—such as Dept of EMPLOYEE are basically OIDs that serve as references to other objects to represent relationships among the objects. For example, the attribute Dept of EMPLOYEE is of type DEPARTMENT and hence is used to refer to a specific DEPARTMENT object (the DEPARTMENT object where the employee works). The value of such an attribute would be an OID for a specific DEPARTMENT object. The attribute Employees of DEPARTMENT has as its value a set of references (that is, a set of OIDs) to objects of type EMPLOYEE.

Encapsulation of Object

- The concept of encapsulation is one of the main characteristics of OO languages and systems. Encapsulation is related to abstract data types and information hiding in programming language. In traditional database models and systems this concept was not applied.
- It defines behavior of a type of object based on operations that can be externally applied. External users only aware of interface of the operations. Dot notation is used to apply operations to object.
- For database applications, the requirement that all objects be completely encapsulated is too stringent. One way to relax this requirement is to divide the structure of an object into visible and hidden attributes (instance variables). Visible attributes can be seen by and are directly accessible to the database users and programmers via the query language. The hidden attributes of an object are completely encapsulated and can be accessed only through predefined operations.
- For example, in the relational model, the operations for selecting, inserting, deleting, and modifying tuples are generic and may be applied to any relation in the database. The relation and its attributes are visible to users and to external programs that access the relation by using these operations. The concept of encapsulation is applied to database objects in ODBs by defining the behavior of a type of object based on the operations that can be externally applied to objects of that type. Some operations may be used to create (insert) or destroy (delete) objects; other operations may update the object state; and others may be used to retrieve parts of the object state or to apply some calculations.

```
define class EMPLOYEE
type tuple ( fullname: string;
        Birth_date: DATE;
        Address: string;
        Supervisor: EMPLOYEE;
        Dept: DEPARTMENT;
        );

operations
      age: integer;
      create_emp: EMPLOYEE;
      destroy_emp: boolean;
end EMPLOYEE;

define class DEPARTMENT
type tuple ( Dname: string;
        Dnumber: integer;
        Mgr: tuple ( Manager: EMPLOYEE;Start_date: DATE; );
        Locations: set (string);
        Employees: set (EMPLOYEE);
       );
```

```
operations
    no_of_emps: integer;
    create_dept: DEPARTMENT;
    destroy_dept: boolean;
    assign_emp(e: EMPLOYEE): boolean;(* adds an employee to the department *)
    remove_emp(e: EMPLOYEE): boolean;(* removes an employee from the department *)
end DEPARTMENT;
```

Fig: Adding Operation to object Type Employee and Department

- Figure above shows how the type definitions in previous figure can be extended with operations to define classes. A number of operations are declared for each class, and the signature (interface) of each operation is included in the class definition. A method (implementation) for each operation must be defined elsewhere using a programming language. Typical operations include the object constructor operation (often called new), which is used to create a new object, and the destructor operation, which is used to destroy (delete) an object. A number of object modifier operations can also be declared to modify the states (values) of various attributes of an object. Additional operations can retrieve information about the object.
- An operation is typically applied to an object by using the dot notation. For example, if d is a reference to a DEPARTMENT object, we can invoke an operation such as no_of_emps by writing d.no_of_emps. Similarly, by writing d.destroy_dept, the object referenced by d is destroyed (deleted). The dot notation is also used to refer to attributes of an object. for example, by writing d.Dnumber or d.Mgr_Start_date.

Persistence of Object

- An ODBS is often closely coupled with an object-oriented programming language (OOPL).
- The OOPL is used to specify the method (operation) implementations as well as other application code. Not all objects are meant to be stored permanently in the database**. Transient objects exist** in the executing program and disappear once the program terminates. **Persistent objects are stored** in the database and persist after program termination.
- The typical mechanisms for making an object persistent are **naming and reachability.**
- The naming mechanism involves giving an object a unique persistent name within a particular database. This persistent object name can be given via a specific statement or operation in the program, as shown in Figure below.
- The **named persistent objects** are used as entry points to the database through which users and applications can start their database access. Obviously, it is not practical to give names to all objects in a large database that includes thousands of objects, so most objects are made persistent by using the second mechanism, called **reachability.** The reachability mechanism works by making the object reachable from some other persistent object. An object B is said to be reachable from an object A if a sequence of references in the database lead from object A to object B.

```
define class DEPARTMENT_SET
type set (DEPARTMENT);

operations
    add_dept(d: DEPARTMENT): boolean;(* adds a department to the DEPARTMENT_SET object *)
    remove_dept(d: DEPARTMENT): boolean; (* removes a department from the DEPARTMENT_SET
                                                    object *)
    create_dept_set: DEPARTMENT_SET;
    destroy_dept_set: boolean;
end Department_Set;

persistent name
```

```
ALL_DEPARTMENTS: DEPARTMENT_SET;(* ALL_DEPARTMENTS is a persistent named
                                   object of type DEPARTMENT_SET *)
```

```
d:= create_dept; (* create a new DEPARTMENT object in the variable d *)
b:= ALL_DEPARTMENTS.add_dept(d);(* make d persistent by adding it to the persistent set
ALL_DEPARTMENTS *)
```

Fig: Creating Persistent object by naming and recheability

- In above figure, we have defined a class DEPARTMENT_SET whose objects are of type set(DEPARTMENT). We can create an object of type DEPARTMENT_SET and give it a persistent name i.e., ALL_DEPARTMENTS, as shown in Figure above. Any DEPARTMENT object that is added to the set of ALL_DEPARTMENTS by using the add_dept operation becomes persistent by virtue of its being reachable from ALL_DEPARTMENTS.
- Notice the difference between traditional database models and ODBs in this respect. In traditional database models, such as the relational model, all objects are assumed to be persistent. Hence, when a table such as EMPLOYEE is created in a relational database, it represents both the type declaration for EMPLOYEE and a persistent set of all EMPLOYEE records (tuples). In the OO approach, a class declaration of EMPLOYEE specifies only the type and operations for a class of objects. The user must separately define a persistent object of type set (EMPLOYEE) whose value is the collection of references (OIDs) to all persistent EMPLOYEE objects, if this is desired, as shown in Figure above.

Advantage of Object Database

- **Objects Can Encapsulate Operations Along with Data:** Database tables contain only data. Objects can include the ability to perform operations that are likely to be performed on that data. An application can simply call the methods to retrieve the information.
- **Increased Developer Productivity:** Object types and their methods are stored with the data in the database, so they are available for any application to use. Developers do not need to re-create similar structures and methods in every application. This also ensures that developers are using consistent standards.
- **Enhanced Modeling Capabilities:** Object databases allow for the direct representation of complex real-world objects and relationships, making it easier to model and represent data accurately.
- **Persistence of Complex Objects:** Object databases enable the persistence of complex data structures, including objects with nested relationships. This allows for the storage and retrieval of entire object graphs, preserving the integrity and structure of the data.
- **Increased Developer Productivity:** Object databases align closely with object-oriented programming paradigms, allowing developers to work with a consistent model throughout the application development process. This can lead to increased developer productivity and reduced development time.
- **Improved Performance:** Object databases can provide better performance for certain types of applications. They eliminate the need for complex joins and provide faster access to data by directly accessing objects and their attributes.

Disadvantages of Object Databases:

- **Limited Industry Support:** Object databases have historically received less industry support compared to relational databases. As a result, there may be a smaller ecosystem of tools, frameworks, and community support available for object databases.
- **Integration Challenges:** Object databases may face challenges when integrating with existing systems and tools that are designed for relational databases. Migrating existing data and applications to an object database can require additional effort and may introduce compatibility issues.

- **Lack of Standardization:** Unlike relational databases, which have widely adopted SQL standards, object databases lack a standardized query language. This can make it more challenging to interchange data between different systems or migrate to a different database technology in the future.

Object Database Extension to SQL

- SQL was first specified by Chamberlin and Boyce (1974) and underwent enhancements and standardization in 1989 and 1992. Starting with the version of SQL known as SQL3, features from object databases were incorporated into the SQL standard.
- The relational model with object database enhancements is sometimes referred to as the **object-relational model.**
- The following are some of the **object database features that have been included in SQL:**
    1. **type constructors** have been added to specify complex objects. These include the **row type**, which corresponds to the **tuple (or struct) constructor. An array type** for specifying **collections** is also provided. Other collection type constructors, such as **set, list, and bag constructors,** were not part of the original SQL/Object specifications in SQL:99 but were later included in the standard in SQL:2008. This will be discussed in a subsequent topic.
    2. A mechanism for specifying **object identity** through the use of reference type is included. We can specify system-generated object identifiers or alternatively can use primary key as OID as in traditional relational model. Examples:

        REF IS SYSTEM GENERATED
        REF IS <OID_ATTRIBUTE> <VALUE_GENERATION_METHOD> ;

    3. **Encapsulation** of operations is provided through the mechanism of user-defined types (UDTs) that may include operations as part of their declaration. These are somewhat similar to the concept of abstract data types that were developed in programming languages. In addition, the concept of user-defined routines (UDRs) allows the definition of general methods (operations). We can specify methods (or operations) in addition to the attributes .

        CREATE TYPE <TYPE-NAME>
        (
            <LIST OF COMPONENT ATTRIBUTES AND THEIR TYPES>
            <DECLARATION OF FUNCTIONS (METHODS)>
        );

        In general, a UDT can have a number of user-defined functions associated with it. The syntax is

        INSTANCE METHOD <NAME> (<ARGUMENT_LIST>) RETURNS <RETURN_TYPE>;

        Two types of functions can be defined: internal SQL and external. Internal functions are written in the extended PSM(Persistent Stored Module) language of SQL . External functions are written in a host language, with only their signature (interface) appearing in the UDT definition

    4. **Inheritance** mechanisms are provided using the keyword UNDER. All attributes/operations are inherit. Order of supertypes in UNDER clause determines inheritance hierarchy. Instance (object) of a subtype can be used in every context in which a supertype instance used. Subtype can redefine any function defined in supertype

        CREATE TYPE <Sub_Class_Type> UNDER <Base_Class_Type> AS
        (
            <LIST OF COMPONENT ATTRIBUTES AND THEIR TYPES>
            <DECLARATION OF FUNCTIONS (METHODS)>
)

User-Defined Types Using CREATE TYPE and Complex Objects

- To allow the creation of complex-structured objects and to separate the declaration of a class/type from the creation of a table (which is the collection of objects/rows and hence corresponds to the extent), SQL now provides **user-defined types (UDTs).**
- In addition, four collection types have been included to allow for collections (multivalued types and attributes) in order to specify complex-structured objects rather than just simple (flat) records. The user will create the UDTs for a particular application as part of the database schema.
- A UDT may be specified in its simplest form using the following syntax:

CREATE TYPE TYPE_NAME AS (<component declarations>);

- By using a UDT as the type for an attribute within another UDT, a complex structure for objects (tuples) in a table can be created, much like that achieved by nesting type constructors/ generators as discussed earlier.
- To drop any TYPE created we can use following syntax

DROP TYPE <TYPE_NAME> FORCE;

Example: Drop type STREET_ADDR_TYPE FORCE

- We must specify FORCE to drop the created type, as the database invalidates all subtypes depending on this supertype while attempting to perform drop operation.
- Further we can use following command to see the structure TYPE we have created.

DESC <TYPE_NAME>

Example: DESC STREET_ADDR_TYPE

- To create table from some TYPE the following syntax can be used

CREATE TABLE <TABLE_NAME> OF <TYPE_NAME>

Example: CREATE TABLE tbl_street_address OF STREET_ADDR_TYPE

Example:

1. **Using UDTs as types for attributes such as Address and Phone**

```
CREATE TYPE STREET_ADDR_TYPE AS OBJECT
(
        Street_NUMBER VARCHAR(5),
        STREET_NAME VARCHAR(25),
        APT_NO VARCHAR(5),
        SUITE_NO VARCHAR(5)
);

CREATE TYPE USA_ADDR_TYPE AS OBJECT
(
        STREET_ADDR STREET_ADDR_TYPE,
        CITY VARCHAR (25),
        ZIP VARCHAR (10)
);



CREATE TYPE USA_PHONE_TYPE AS OBJECT
 (
        PHONE_TYPE VARCHAR (5),
        AREA_CODE CHAR (3),
        PHONE_NUM CHAR (7)
);
```

2. Specifying UDTs for PERSON_TYPE

```
CREATE TYPE PERSON_TYPE AS OBJECT
(
        NAME VARCHAR (35),
        SEX CHAR,
        BIRTH_DATE DATE,
        PHONES USA_PHONE_TYPE ARRAY [4],
        ADDR USA_ADDR_TYPE
        INSTANTIABLE
        NOT FINAL
        REF IS SYSTEM GENERATED
        INSTANCE METHOD AGE() RETURNS INTEGER;

        CREATE INSTANCE METHOD AGE() RETURNS INTEGER
        FOR PERSON_TYPE
        BEGIN
                RETURN /* CODE TO CALCULATE A PERSON'S AGE FROM
                                        TODAY'S DATE AND SELF.BIRTH_DATE */
        END;
);
```

- In above UDT's Declaration of Person_Type, we have specified that UDT is instantiable which means the user can then create one or more tables based on the UDT. However, if keyword INSTANTIABLE is left out, can use UDT only as attribute data type – not as a basis for a table of objects.
- A mechanism for specifying object identity through the use of reference type is included using "REF IS SYSTEM GENERATED", meaning that Object identity is automatically generated.
- The keyword **NOT FINAL** indicates that subtypes can be created for that type.

3. Specifying UDTs for STUDENT_TYPE, EMPLOYEE_TYPE and MANAGER_TYPE as two subtypes of PERSON_TYPE.

```
CREATE TYPE PERSON_TYPE AS OBJECT
(
        NAME VARCHAR (35),
        SEX CHAR,
        BIRTH_DATE DATE,
        PHONES USA_PHONE_TYPE ARRAY [4],
        ADDR USA_ADDR_TYPE
        INSTANTIABLE
        NOT FINAL
        REF IS SYSTEM GENERATED
        INSTANCE METHOD AGE() RETURNS INTEGER;

        CREATE INSTANCE METHOD AGE() RETURNS INTEGER
        FOR PERSON_TYPE
        BEGIN
                RETURN /* CODE TO CALCULATE A PERSON'S AGE FROM
                        TODAY'S DATE AND SELF.BIRTH_DATE */
        END;
);
```

```
CREATE TYPE GRADE_TYPE AS OBJECT
 (
        COURSENO CHAR (8),
        SEMESTER VARCHAR (8),
        YEAR CHAR (4),
        GRADE CHAR
);

CREATE TYPE STUDENT_TYPE UNDER PERSON_TYPE
(
        MAJOR_CODE CHAR (4),
        STUDENT_ID CHAR (12),
        DEGREE VARCHAR (5),
        TRANSCRIPT GRADE_TYPE ARRAY [100]
        INSTANTIABLE
        NOT FINAL
        INSTANCE METHOD GPA( ) RETURNS FLOAT;

        CREATE INSTANCE METHOD GPA( ) RETURNS FLOAT
        FOR STUDENT_TYPE
        BEGIN
                RETURN /* CODE TO CALCULATE A STUDENT'S GPA FROM
                                SELF.TRANSCRIPT */
        END;
);

CREATE TYPE EMPLOYEE_TYPE UNDER PERSON_TYPE
(
        JOB_CODE CHAR (4),
        SALARY FLOAT,
        SSN CHAR (11)
        INSTANTIABLE
        NOT FINAL
);

CREATE TYPE MANAGER_TYPE UNDER EMPLOYEE_TYPE
(
        DEPT_MANAGED CHAR (20)
        INSTANTIABLE
);
```

4. **Creating tables based on some of the UDTs and illustrating table inheritance.**

```
CREATE TABLE PERSON OF PERSON_TYPE

CREATE TABLE EMPLOYEE OF EMPLOYEE_TYPE
UNDER PERSON;

CREATE TABLE MANAGER OF MANAGER_TYPE
UNDER EMPLOYEE;
```

```
CREATE TABLE STUDENT OF STUDENT_TYPE
UNDER PERSON;
```

Solved Example

1. **WRITE QUERY To perform the following operation.**
   a. **Create Person Object with attribute firstname,lastname & dob and method getAge() to return the person age.**

```
CREATE OR REPLACE TYPE Person AS OBJECT
(
        first_name  VARCHAR(50),
        last_name   VARCHAR(50),
        date_of_birth  DATE,
        MEMBER FUNCTION getAge RETURN NUMBER
);
```

   b. **Define the getAge Member Fucntion**

```
CREATE OR REPLACE TYPE BODY Person AS
  MEMBER FUNCTION getAge RETURN NUMBER AS
  BEGIN
    RETURN Trunc(Months_Between(Sysdate, date_of_birth)/12);
  END getAge;
END;
```

Note: The TRUNC (date) function returns date. The value returned is always of datatype.

   c. **Create Table named tbl_people with attribute id and person of Person Type.**

```
CREATE TABLE tbl_people
(
        id     NUMBER(10) NOT NULL,
        person  Person
);
```

   d. **Write SQL DML command to insert data into tbl_people table using PersonObj() constructor.**

```
INSERT INTO tbl_people VALUES
(1, PersonObj('John','Doe',   TO_DATE('01/01/1999','DD/MM/YYYY')));
COMMIT;
```

   e. **Retrieve all the record stored in table tbl_people**

```
SELECT p.id,p.person.first_name,p.person.getAge() age FROM   tbl_people p;
```

2. **Write Query for the following operation.**
   a. **Create Teacher Object with attribute firstname,lastname and mobilenumber. Each person can have up to two mobile number.[Collection Type]**

Note: Data structure called the VARRAY, can store a fixed-size sequential collection of elements of the same type. A varray is used to store an ordered collection of data, however it is often better to think of an array as a collection of variables of the same type.

Syntax for creating varray type:

CREATE OR REPLACE TYPE varray_type_name IS VARRAY(n) of <element_type>

Where,

- varray_type_name is a valid attribute name,
- n is the number of elements (maximum) in the varray,
- element_type is the data type of the elements of the array.

```
CREATE TYPE Mobile_Number_Type  AS VARRAY(2) OF VARCHAR(10);


CREATE OR REPLACE TYPE Teacher AS OBJECT
(
  first_name  VARCHAR(50),
  last_name   VARCHAR(50),
  mobile_number Mobile_Number_Type
);
```

b. Create table tbl_teacher of Teacher Type.

```
CREATE TABLE tbl_teacher OF  Teacher
```

c. Insert one record of teacher

```
insert into tbl_teacher values('ram','thapa',new Mobile_Number_Type('9856013267','9846513267'))
```

d. Display all the records

```
select * from tbl_teacher
```

e. Display first name and phonenumber of each teacher.

```
select t.first_name, mb.column_value from tbl_teacher t, Table(t.mobile_number) mb
```

Note:

- The TABLE keyword can be used to unnest a collection (such as a nested table or a VARRAY) and treat it as a table. This allows you to query the elements of the collection as if they were rows in a table.
- The COLUMN_VALUE keyword is used to access the values of elements within a nested table or collection column when performing an unnest operation.

The ODMG Object Model and the Object Definition Language ODL

- One of the reasons for the success of commercial relational DBMSs is the SQL standard. The lack of a standard for ODBs for several years may have caused some potential users to shy away from converting to this new technology.
- So a consortium of ODB vendors and users, called ODMG (Object Data Management Group), proposed a standard that is known as the ODMG-93 or ODMG 1.0 standard. This was revised into ODMG 2.0, and later to ODMG 3.0.
- This standard is made up of several parts, including the **object model, the object definition language (ODL), the object query language** (OQL), and the bindings to object-oriented programming languages.

Object Definition Language(ODL)

- The properties of a class arc specified using ODL and are of three kinds: attributes, relationships, and methods.
- The ODL is designed to support the semantic constructs of the ODMG object model and is independent of any programming language.
- Its main use is to create object specifications—that is, classes and interfaces.
- Hence, ODL is not a programming language. A user can specify a database schema in ODL independently of any programming language, and then use the specific language bindings to specify how ODL constructs can be mapped to constructs in specific programming languages, such as C++, Smalltalk, and Java.

Object Model of ODMG Model

- The ODMG object model is the data model upon which **the object definition language (ODL) and object query language (OQL) are based.**
- It is meant to provide a standard data model for object databases, just as SQL describes a standard data model for relational databases.
- The ODMG data model is the basis for an OODBMS, just like the relational data model is the basis for an RDBMS.
- **In ODMG,** Objects and literals are the basic building blocks of the object model. The main difference between the two is that an **object has both an object identifier and a state (or current value),** whereas **a literal has a value (state) but no object identifier**. In either case, the value can have a complex structure. The object state can change over time by modifying the object value.
- An object has five aspects: **identifier, name, lifetime, structure, and creation.**
  1. The **object identifier** is a unique system-wide identifier (or Object_id). Every object must have an object identifier.
  2. Some objects may optionally be given a **unique name** within a particular ODMS—this name can be used to locate the object, and the system should return the object given that name. Obviously, not all individual objects will have unique names. Typically, a few objects, mainly those that hold **collections of objects** of a particular object class/type—such as extents—will have a name. These names are used as entry points to the database; that is, by locating these objects by their unique name, the user can then locate other objects that are referenced from these objects. All names within a particular ODB must be unique.
  3. The **lifetime** of an object specifies whether it is a persistent object (that is, a database object) or transient object (that is, an object in an executing program that disappears after the program terminates). Lifetimes are independent of classes/types—that is, some objects of a particular class may be transient whereas others may be persistent.
  4. The **structure** of an object specifies how the object is constructed by using the type constructors. The structure specifies whether an object is atomic or not. An atomic object refers to a single object that follows a user-defined type, such as Employee or Department. If an object is not atomic, then it will be composed of other objects. For example, a collection object is not an atomic object, since its state will be a collection of other objects. In the ODMG model, an atomic object is any individual user-defined object.
  5. **Object creation** refers to the manner in which an object can be created. This is typically accomplished via an operation **new** for a special Object_Factory interface.

- In the object model, a literal is a value that does not have an object identifier. However, the value may have a simple or complex structure. There are three types of **literals: atomic, structured, and collection.**
  1. **Atomic literals** correspond to the values of **basic data types and are predefined.** The basic data types of the object model include long, short, and unsigned integer numbers (these are specified by the keywords long, short, unsigned long, and unsigned short in ODL), regular and double precision floating-point numbers (float, double), Boolean values (boolean), single characters (char), character strings (string), and enumeration types (enum), among others.
  2. **Structured literals** correspond roughly to values that are constructed using the **tuple constructor**. The built-in structured literals include Date, Interval, Time, and Timestamp . Additional user-defined structured literals

can be defined as needed by each application. User-defined structures are created using the **STRUCT** keyword in ODL, as in the C and C++ programming languages.

3. **Collection literals specify** a literal value that is **a collection of objects or values** but the **collection itself does not have an Object_id.** The collections in the object model can be defined by the type generators set<T>, bag<T>, list<T>, and array<T>, where T is the type of objects or values in the collection. Another collection type is dictionary<K, V>, which is a collection of associations <K, V>, where each K is a key (a unique search value) associated with a value V; this can be used to create an index on a collection of values V.

- The notation of ODMG uses three main concepts: **interface, literal, and class**.
- Following the ODMG terminology, we use the word **behavior to refer to operations** and **state to refer to properties (attributes and relationships)**.
- An interface specifies only the behavior of an object type and is typically non-instantiable (that is, no objects are created corresponding to an interface). Although an interface may have **state properties** (attributes and relationships) as part of its specifications, these cannot be inherited from the interface. Hence, an interface serves to define operations that can be inherited by other interfaces, as well as by classes that define the user-defined objects for a particular application. For each interface, we can declare **an extent**, which is the same for the current set of objects of that class. The extent is analogous to the instance of a relation and the interface is analogous to the schema.
- **A class specifies both state (attributes) and behavior (operations) of an object type** and is **instantiable. Also a class can defines relationship which specifies references to other object or collection of other objects. A relationship has a corresponding inverse relationship.**
- Attribute may be atomic type, structure type or also a collection. Hence, database and application objects are typically created based on the user-specified class declarations that form a database schema.
- Finally, a **literal declaration specifies state but no behavior**. Thus, a **literal instance holds a simple or complex structured value but has neither an object identifier nor encapsulated operations.**
- Figure below is a simplified version of the object model.

```
interface Object
{
        boolean add(object);
         …
        object copy();
        void delete();
        …….
};
```

Fig: The basic Object interface

- The following ODL definitions of the Movie and Threater classes illustrate these concepts.

```
Interface Movie (extent Movies key movieNarne)
{
        attribute Date date_start;
        attribute Date date_end;
        attribute string moviename;
        attribute struct directorname{string fname, string mname, string lname};
        relationship Set(Theater) shownAt inverse Theater::nowShowing;
}
```

- The collection of database objects whose class is Movie is called Movies. No two objects in Movies have the same movieName value, as the key declaration indicates.

- Each movie is shown at a set of theaters and is shown during the specified period.
- A theater is an object of class Theater, defined as

```
Interface Theater (extent Theaters key threaterName)
{
        attribute string threaterName;
        attribute string address;
        attribute integer ticketPrice;
        relationship Set (Movie) nowShowing inverse .Mlovie::shownAt;
        float numshowing();
}
```

- Each theater shows several movies. Observe that the shownAt relationship of Movie and the nowShowing relationship of Theater are declared to be inverses of each other. Theater also has a method numshowing() that can be applied to a theater object to find the number of movies being shown at that theater.

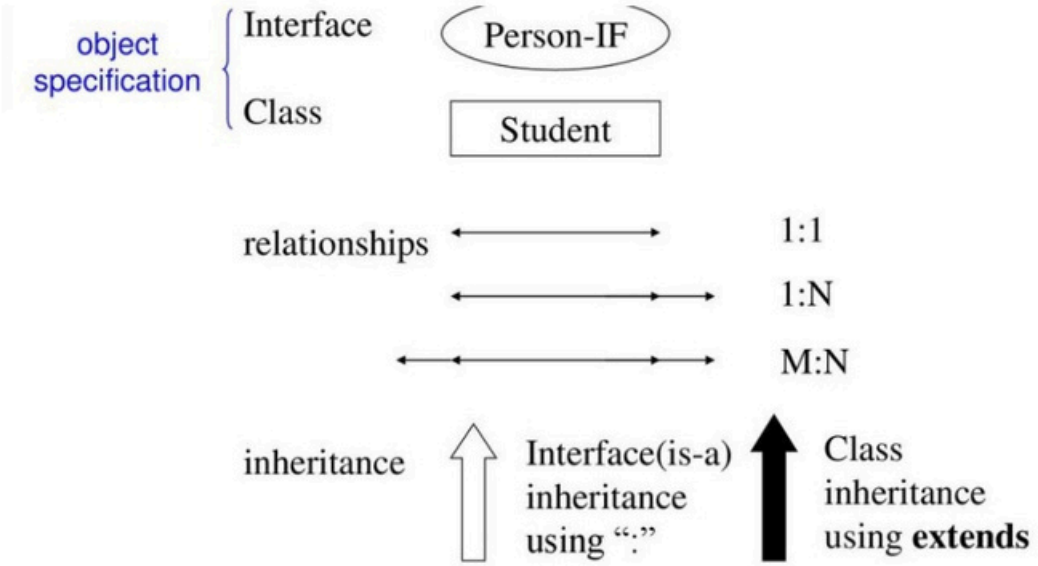
Inheritance in the Object Model of ODMG

- In the ODMG object model, two types of inheritance relationships exist: **behavior only inheritance** and **state plus behavior inheritance.**
- Behavior inheritance is also known as ISA or interface inheritance and is specified by the colon (:) notation. Hence, in the ODMG object model, **behavior inheritance requires the supertype to be an interface**, whereas the **subtype could be either a class or another interface.**
- The other inheritance relationship, called EXTENDS inheritance, is specified by the keyword **extends. It is used to inherit both state and behavior strictly among classes**, so both the **supertype and the subtype must be classes.**
- **Multiple inheritance via extends is not permitted**. However, multiple inheritance is allowed for **behavior inheritance via the colon (:) notation**. Hence, an interface may inherit behavior from several other interfaces.
- A class may also inherit behavior from several interfaces via colon (:) notation, in addition to inheriting behavior and state from at most one other class via extends.
- The following ODL definitions of the Movie and Theater classes illustrate these concepts.
- ODL also allows us to specify inheritance hierarchies, as the following class definition illustrates:

```
interface SpecialShow extends Movie (extent SpecialShows)
{
        attribute integer MaximumAttendees;
        attribute string benefitCharity;
}
```

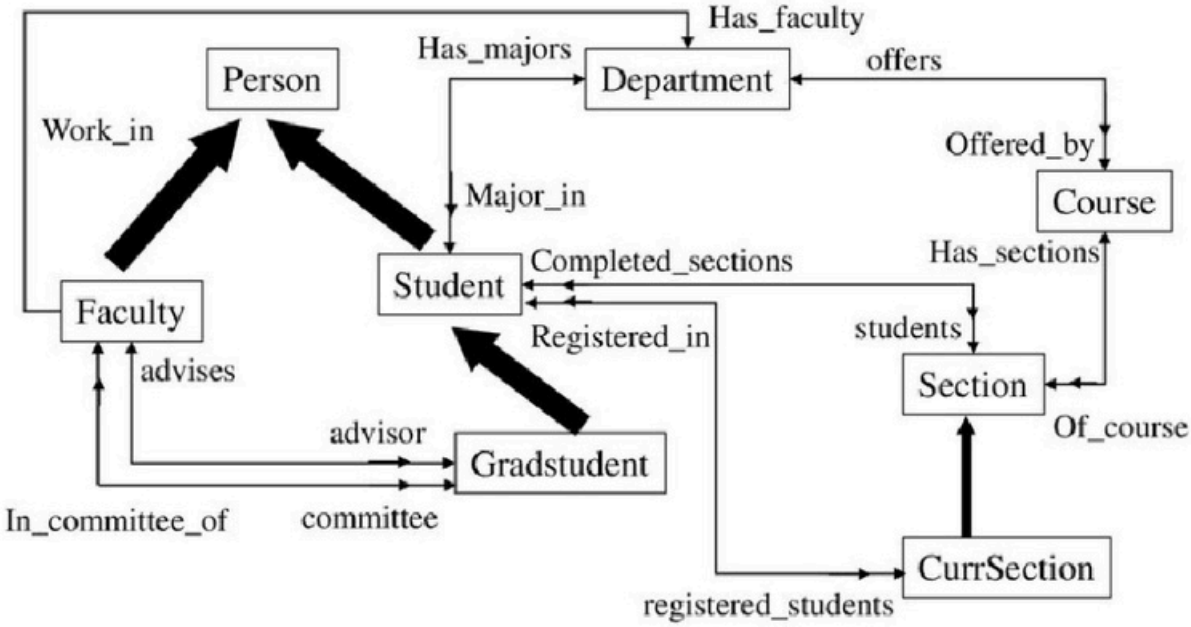
In the above example. an object of class SpecialShow is an object of class Movie, with Special additional properties like MaximumAttendees and benefitCharity.

Graphical Representation of ODL Schemas



Example:

- **Draw graphical ODB schema for university database.**



Object Database Conceptual Design

Differences between Conceptual Design of ODB and RDB

- One of the main differences between ODB and RDB design is how relationships are handled. In ODB, relationships are typically handled by having **relationship properties or reference attributes** that include OID(s) of the related objects. Both single references and collections of references are allowed. References for a binary relationship can be declared in a single direction, or in both directions, depending on the types of access expected. If declared in both directions, they may be specified as inverses of one another, thus enforcing the ODB equivalent of the relational referential integrity constraint. However, In RDB, relationships among tuples (records) are specified by attributes with matching values. These can be considered as value references and are specified via **foreign keys,** which are values of primary key attributes repeated in tuples of the referencing relation. These are limited to being single valued in each record because multivalued attributes are not permitted in the basic relational model. Thus, M:N relationships must be represented not directly, but as a separate relation or table.

- Another major area of difference between ODB and RDB design is how inheritance is handled. In ODB, these structures are built into the model, so the mapping is achieved by using the inheritance constructs, such as derived (:) and extends. In relational design, there are several options to choose from since no built-in construct exists for inheritance in the basic relational model.
- The third major difference is that in ODB design, it is necessary to specify the operations early on in the design since they are part of the class specifications. Although it is important to specify operations during the design phase for all types of databases, the design of operations may be delayed in RDB design as it is not strictly required until the implementation phase.

Mapping an EER Schema to an ODB Schema

- It is relatively straightforward to design the type declarations of object classes for an ODBMS from an EER schema that contains neither categories nor n-ary relationships with n > 2.
- However, the operations of classes are not specified in the EER diagram and must be added to the class declarations after the structural mapping is completed.
- The major steps involved in the mapping from EER to ODL is as follows:

Step 1: Create an ODL class for each EER entity type or subclass. The type of the ODL class should include all the attributes of the EER class. Multivalued attributes are typically declared by using the set, bag, or list constructors. If the values of the multivalued attribute for an object should be ordered, the list constructor is chosen; if duplicates are allowed, the bag constructor should be chosen; otherwise, the set constructor is chosen. Composite attributes are mapped into a tuple constructor (by using a struct declaration in ODL).

Step 2: Add relationship properties or reference attributes for each binary relationship into the ODL classes that participate in the relationship. These may be created in one or both directions. If a binary relationship is represented by references in both directions, declare the references to be relationship properties that are inverses of one another, if such a facility exists. If a binary relationship is represented by a reference in only one direction, declare the reference to be an attribute in the referencing class whose type is the referenced class name.
The relationship properties or reference attributes may be single-valued or collection types. They will be single-valued for binary relationships in the 1:1 or N:1 directions; they will be collection types (set-valued or list-valued) for relationships in the 1:N or M:N direction.

Step 3: Include appropriate operations for each class. These are not available from the EER schema and must be added to the database design by referring to the original requirements. A constructor method should include program code that checks any constraints that must hold when a new object is created. A destructor method should check any constraints that may be violated when an object is deleted. Other methods should include any further constraint checks that are relevant.

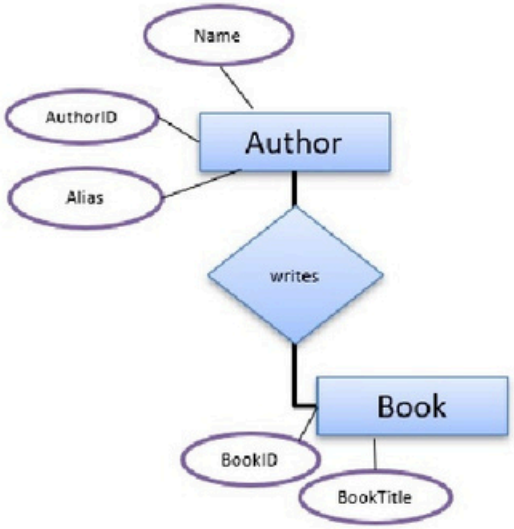
Step 4: An ODL class that corresponds to a subclass in the EER schema inherits (via extends) the attributes, relationships, and methods of its superclass in the ODL schema. Its specific (local) attributes, relationship references, and operations are also specified.

Step 5: Weak entity types can be mapped in the same way as regular entity types. An alternative mapping is possible for weak entity types that do not participate in any relationships except their identifying relationship; these can be mapped as though they were composite multivalued attributes of the owner entity type, by using the set<struct<…>> constructors. **The attributes of the weak entity are included in the struct<…> construct, which corresponds to a tuple constructor.**

Step 7: An n-ary relationship with degree n > 2 can be mapped into a separate class, with appropriate references to each participating class

Example:

Write ODL schema for the below ER Diagram



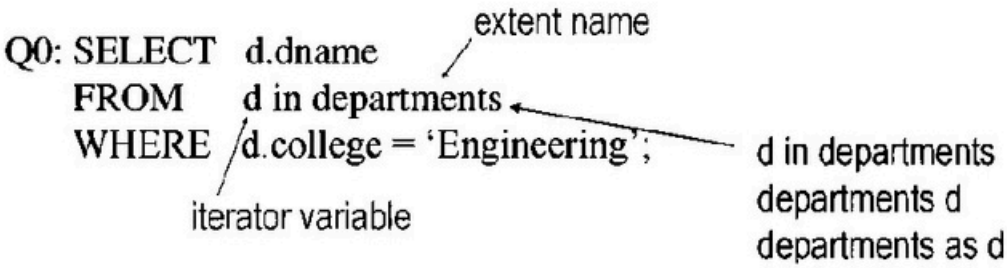

```
Interface Author (extent Authors key AuthorID)
{
        Attribute string name;
        Attribute string AuthorId;
        Attribute string Alias;
        Relationship set<Book> writes inverse Book::writtenby;
        int booknumberswrittenbythisauthor();
}

Interface Book (extent Books key BookID)
{
        Attribute string BookID;
        Attribute string BookTitle;
        Relationship set<Author> writtenby inverse Book::writes;
        int authornumbersforthisbook();

}
```

The Object Query Language (OQL)

- The object query language OQL is the query language proposed for the ODMG object model.
- It is designed to work closely with the programming languages for which an ODMG binding is defined, such as C++, Smalltalk, and Java.
- Hence, an OQL query embedded into one of these programming languages can return objects that match the type system of that language.
- Additionally, the implementations of class operations in an ODMG schema can have their code written in these programming languages.
- The OQL syntax for queries is similar to the syntax of the relational standard query language SQL, with additional features for ODMG concepts, such as object identity, complex objects, operations, inheritance, polymorphism, and relationships.

- basic OQL syntax
  - select ... from ... where ...
  - Retrieve the names of all departments in the college of 'Engineering'

How to refer to a persistent object?
Entry point (named persistent object; or name of the extent of a class)



- The data type of query result can be any type defined in the ODMG model. A query doesn't have to follow the select…from…where format. A persistent name on its own can server as query whose result is a reference to the persistent objects.
  For example:

  departments;

  the above query outputs all the department objects i.e. Set<Department>
- A path expression is used to specify a path to attributes and object in an entry point. A path expression starts at a persistent object name or its iterator variable. The name will be followed by zero or more dot connected relationship or attribute name.
  For example:

  Departments.dname;

Language Binding in the ODMG Standard

- Language binding specifies how ODL/OML constructs are mapped to some programming language(say C++) constructs.
- Basic design principle is that programmers should think there is only one language being used.
- Language binding includes
  i. Class libraries
  ii. Object Definition Language (ODL) and Object Manipulation language(OML)
  iii. A set of constructs called physical pragmas that allows programmers some control over physical storage concerns.
- Class library provided containing classes and functions that implement ODL constructs. Also, OML is used to specify how database objects are retrieved and manipulated within application program.

C++ Language Binding

- The class library added to C++ for the ODMG standards uses the prefix d_ for class declarations
- d_Ref<T> is defined for each database class T
- To utilize ODMG's collection types, various templates are defined, e.g., d_Object<T> specifies the operations to be inherited by all objects
- A template class is provided for each type of ODMG collections:
  - **d_Set<T>**
  - **d_List<T>**
  - **d_Bag<t>**
  - **d_Varray<t>**

- **d_Dictionary<T>**

- Thus a programmer can declare:
  - **d_Set<d_Ref<Student>>**
- The data types of ODMG database attributes are also available to the C++ programmers via the d_ prefix, e.g., d_Short, d_Long, d_Float
- To specify relationships, the prefix Rel_ is used within the prefix of type names

  E.g., d_Rel_Ref<Dept, has_majors> majors_in;
- The C++ binding also allows the creation of extents via using the library class d_Extent:
  d_Extent<Person> All_Persons(CS101)