# ADVANCED DATABASE SYSTEMS

## COURSE NAME: MTECH – IST SEMESTER

## COURSE CODE: CSE511

## TEACHER INCHARGE: *DR. SHIFAA BASHARAT*

# UNIT 1

## Object-Oriented Concepts and Features:

The term *object-oriented*—abbreviated *OO* or *O-O*—has its origins in OO programming languages, or OOPLs. The programming language Smalltalk, developed at Xerox PARC4 in the 1970s, was one of the first languages to explicitly incorporate additional OO concepts, such as message passing and inheritance. It is known as a *pure* OO programming language, meaning that it was explicitly designed to be object-oriented. This contrasts with *hybrid* OO programming languages, which incorporate OO concepts into an already existing language e.g. C++, which incorporates OO concepts into the popular C programming language.

An **object** typically has two components: state (value) and behavior (operations). It can have a *complex data structure* as well as *specific operations* defined by the programmer. Objects in an OOPL exist only during program execution; therefore, they are called *transient objects.* An OO database can extend the existence of objects so that they are stored permanently in a database, and hence the objects become *persistent objects* that exist beyond program termination and can be retrieved later and shared by other programs. In other words, OO databases store persistent objects permanently in secondary storage and allow the sharing of these objects among multiple programs and applications. This requires the incorporation of other well-known features of database management systems, such as indexing mechanisms to efficiently locate the objects, concurrency control to allow object sharing among concurrent programs, and recovery from failures. An OO database system will typically interface with one or more OO programming languages to provide persistent and shared object capabilities.

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 instance variable is similar to the concept of an *attribute* in the relational model, except that instance variables may be encapsulated within the object and thus are not necessarily visible to external users. Instance variables may also be of arbitrarily complex data types. Object-oriented systems allow definition of the operations or functions (behavior) that can be applied to objects of a particular type. In fact, some OO models insist that all operations a user can apply to an object must be predefined. This forces a *complete encapsulation* of objects. This rigid approach has been relaxed in most OO data models for two reasons:

- First, database users often need to know the attribute names so they can specify selection conditions on the attributes to retrieve specific objects.
- Second, complete encapsulation implies that any simple retrieval requires a predefined operation, thus making ad hoc queries difficult to specify on the fly.

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. This encapsulation permits modification of the internal structure of an object, as well as the implementation of its operations, without the need to disturb the external programs that invoke these operations. Hence, encapsulation provides a form of data and operation independence.

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. 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.
One problem in early OO database systems involved representing *relationships* among objects. The insistence on complete encapsulation in early OO data models led to the argument that relationships should not be explicitly represented, but should instead be described by defining appropriate methods that locate related objects. However, this approach does not work very well for complex databases with many relationships because it is useful to identify these relationships and make them visible to users.
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. This may require the use of *late binding* of the operation name to the appropriate method at runtime, when the type of object to which the operation is applied becomes known.

## Object Identity:

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. The OID can be assigned to program variables of the appropriate type when needed.
The main property required of an OID is that it be **immutable**; that is, the OID value of a particular object should not change. This preserves the identity of the real-world object being

represented. Hence, an ODMS must have some mechanism for generating OIDs and preserving the immutability property. 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.
This can be compared 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 ascertain 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). It is also 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. It is more common to use long integers as OIDs and then to use some form of hash table to map the OID value to the current physical address of the object in storage.

Some early OO data models required that everything—from a simple value to a complex object—was represented as an object; hence, every basic value, such as an integer, string, or Boolean value, has an OID. This allows two identical basic values to have different OIDs, which can be useful in some cases. For example, the integer value 50 can sometimes be used to mean a weight in kilograms and at other times to mean the age of a person. Then, two basic objects with distinct OIDs could be created, but both objects would have the integer 50 as their value. Although useful as a theoretical model, this is not very practical, since it leads to the generation of too many OIDs. Hence, most ODBs allow for the representation of both objects and **literals** (or values). 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. In many systems, complex structured literal values can also be created without having a corresponding OID if needed.

## Complex Type Structures for Objects and Literals:

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. 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.

1. **Atom constructor:** This includes the basic built-in data types of the object model, which are similar to the basic types in many programming languages: integers, strings, floating-point numbers, enumerated types, Booleans, and so on. These basic data types are called **single valued** or **atomic** types, since each value of the type is considered an atomic (indivisible) single value.

2. **Structured type constructor (struct** or **tuple** constructor): This 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. More accurately, the struct constructor is not considered to be a type, but rather a **type generator**, because many different structured types can be created. 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>.
3. **Collection (or *multivalued*) type constructors:** To create complex nested type structures in the object model, the *collection* type constructors are needed. These include the **set(T)**, **list(T)**, **bag(T)**, **array(T)**, and **dictionary(K,T)** type constructors. These allow 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.
   The **set constructor** will create objects or literals that are a set of *distinct* elements {*i*1, *i*2, … , *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* [*i*1, *i*2, … , *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 *j*th 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.
   Finally, 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*.
   The main characteristic of a collection type is that its objects or values will be a *collection of objects or values of the same type* that may be unordered (such as a set or a bag) or ordered (such as a list or an array). The **tuple** type constructor is often called a **structured type**, since it corresponds to the **struct** construct in the C and C++ programming languages.

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 1 shows how we may declare EMPLOYEE and DEPARTMENT types. In Figure 1, the attributes that refer to other objects—such as Dept of EMPLOYEE or Projects of DEPARTMENT—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. A binary relationship can be represented in one direction, or it can have an *inverse reference.* The latter representation makes it easy to traverse the relationship in both directions. For example, in Figure 1 the attribute Employees of DEPARTMENT has as its value a *set of references* (that is, a set of OIDs) to objects of type EMPLOYEE; these are the employees who work for the DEPARTMENT. The inverse is the reference attribute Dept of EMPLOYEE.

## Encapsulation of Operations:

The concept of *encapsulation* is one of the main characteristics of OO languages and systems. It is also related to the concepts of *abstract data types* and *information hiding* in programming languages. In traditional database models and systems this concept was not applied, since it is customary to make the structure of database objects visible to users and external programs. In these traditional models, a number of generic database operations are applicable to objects *of all types*. 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. Still other operations may perform a combination of retrieval, calculation, and update. In general, the **implementation** of an operation can be specified in a *general-purpose programming language* that provides flexibility and power in defining the operations.

The external users of the object are only made aware of the **interface** of the operations, which defines the name and arguments (parameters) of each operation. The implementation is hidden from the external users; it includes the definition of any hidden internal data structures of the object and the implementation of the operations that access these structures. The interface part of an operation is sometimes called the **signature**, and the operation implementation is sometimes called the **method**.

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. Most ODMSs employ high-level query languages for accessing visible attributes The term **class** is often used to refer to a type definition, along with the definitions of the operations for that type. Figure 2 shows how the type definitions in Figure 1 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

```
define type EMPLOYEE
    tuple ( Fname:        string;
            Minit :       char;
            Lname:        string;
            Ssn:          string;
            Birth_date:   DATE;
            Address:      string;
            Sex:          char;
            Salary:       float;
            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);
            Projects:     set(PROJECT); );
```

Figure 1: Specifying the object types EMPLOYEE, DATE and DEPARTMENT using type constructors.

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 only exception is the constructor operation, which returns a reference to a new DEPARTMENT object. Hence, it is customary in some OO models to have a default name for the constructor operation that is the name of the class itself. The dot notation is also used to refer to attributes of an object—for example, by writing d.Dnumber or d.Mgr_Start_date.

```
define class EMPLOYEE
    type tuple (  Fname:          string;
                  Minit:          char;
                  Lname:          string;
                  Ssn:            string;
                  Birth_date:     DATE;
                  Address:        string;
                  Sex:            char;
                  Salary:         float;
                  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);
                  Projects        set(PROJECT); );
    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;
```

Figure 2 Adding operations to the definitions of EMPLOYEE and DEPARTMENT

## Object Persistence:

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 3. The named persistent objects are used as **entry points** to the database through which users and applications can start their database access. However, 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 concept of 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*. If we first create a named persistent object *N*, whose state is a *set* of objects of some class *C*, we can make objects of *C* persistent by *adding them* to the set, thus making them reachable from *N*. Hence, *N* is a named object that defines a **persistent collection** of objects of class *C*. In the object model standard, *N* is called the **extent** of *C*.

For example, we can define a class DEPARTMENT_SET (see Figure 3) whose objects are of type set(DEPARTMENT). We can create an object of type DEPARTMENT_SET, and give it a persistent name ALL_DEPARTMENTS, as shown in Figure 3. Any DEPARTMENT object that is added to the set of ALL_DEPARTMENTS by using the add_dept operation becomes persistent by virtue of it being reachable from ALL_DEPARTMENTS.

## Type Hierarchies and Inheritance

**Simplified Model for Inheritance.** Another main characteristic of ODBs is that they allow type hierarchies and inheritance. Inheritance allowsthe definition of new types based on other predefined types, leading to a **type** (or **class**) **hierarchy**.

A type is defined by assigning it a type name and then defining a number of attributes (instance variables) and operations (methods) for the type. The attributes and operations are together alled *functions*,since attributes resemble functions with zero arguments. A function name can be used to refer to the value of an attribute or to refer to the resulting value of an operation (method).The term **function** here is used to refer to both attributes *and* operations, since they are treated similarly in a basic introduction to inheritance.

A type in its simplest form has a **type name** and a list of visible (*public*) **functions**.
**General Syntax:**

TYPE_NAME: function, function, … , function

```
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 *)
```

Figure 3:Creating Persistent Objects by Naming and Reachability

For example, a type that describes characteristics of a PERSON may be defined as follows:

**PERSON: Name, Address, Birth_date, Age, Ssn**

In the **PERSON** type, the Name, Address, Ssn, and Birth_date functions can be implemented as stored attributes, whereas the Age function can be implemented as an operation that calculates the Age from the value of the Birth_date attribute and the current date.
The concept of **subtype** is useful when the designer or user must create a new type that is similar but not identical to an already defined type. The subtype then inherits all the functions of the predefined type, which is referred to as the **supertype**. For example, suppose that we want to define two new types **EMPLOYEE** and **STUDENT** as follows:

**EMPLOYEE: Name, Address, Birth_date, Age, Ssn, Salary, Hire_date, Seniority**
**STUDENT: Name, Address, Birth_date, Age, Ssn, Major, Gpa**

Since both **STUDENT** and **EMPLOYEE** include all the functions defined for **PERSON** plus some additional functions of their own, we can declare them to be **subtypes** of **PERSON.** Each will inherit the previously defined functions of **PERSON**—namely, Name, Address, Birth_date,

Age, and Ssn. For **STUDENT**, it is only necessary to define the new (local) functions Major and Gpa, which are not inherited. Presumably, Major can be defined as a stored attribute, whereas Gpa may be implemented as an operation that calculates the student's grade point average by accessing the Grade values that are internally stored (hidden) within each **STUDENT** object as *hidden attributes.*
For **EMPLOYEE**, the Salary and Hire_date functions may be stored attributes, whereas Seniority may be an operation that calculates Seniority from the value of Hire_date. Therefore, we can declare **EMPLOYEE** and **STUDENT** as follows:

**EMPLOYEE subtype-of PERSON**: Salary, Hire_date, Seniority
**STUDENT subtype-of PERSON**: Major, Gpa

In general, a subtype includes *all* of the functions that are defined for its supertype plus some additional functions that are *specific* only to the subtype. Hence, it is possible to generate a **type hierarchy** to show the supertype/subtype relationships among all the types declared in the system.
As another example, consider a type that describes objects in plane geometry, which may be defined as follows:

**GEOMETRY_OBJECT:** Shape, Area, Reference_point

For the **GEOMETRY_OBJECT** type, Shape is implemented as an attribute (its domain can be an enumerated type with values 'triangle', 'rectangle', 'circle', and so on), and Area is a method that is applied to calculate the area. Reference_point specifies the coordinates of a point that determines the object location. Suppose you want to define a number of subtypes for the **GEOMETRY_OBJECT** type, as follows:

RECTANGLE **subtype-of** GEOMETRY_OBJECT: Width, Height
TRIANGLE S **subtype-of** GEOMETRY_OBJECT: Side1, Side2, Angle
CIRCLE **subtype-of** GEOMETRY_OBJECT: Radius

Here the Area operation may be implemented by a different method for each subtype, since the procedure for area calculation is different for rectangles, triangles, and circles. Similarly, the attribute Reference_point may have a different meaning for each subtype; it might be the center point for **RECTANGLE** and **CIRCLE** objects, and the vertex point between the two given sides for a **TRIANGLE** object.

**Constraints on Extents Corresponding to a Type Hierarchy.** In most ODBs, an **extent** is defined to store the collection of persistent objects for each type or subtype. In this case, the constraint is that every object in an extent that corresponds to a subtype must also be a member of the *extent* that corresponds to its supertype.
Some OO database systems have a predefined system type (called the ROOT class or the OBJECT class) whose extent contains all the objects in the system. Classification then proceeds

by assigning objects into additional subtypes that are meaningful to the application, creating a **type hierarchy** (or **class hierarchy**) for the system. All extents for system- and user-defined classes are subsets of the extent corresponding to the class OBJECT, directly or indirectly. The user may or may not specify an extent for each class (type), depending on the application.
An extent is a named persistent object whose value is a **persistent collection** that holds a collection of objects of the same type that are stored permanently in the database. The objects can be accessed and shared by multiple programs. It is also possible to create a **transient collection**, which exists temporarily during the execution of a program but is not kept when the program terminates. For example, a transient collection may be created in a program to hold the result of a query that selects some objects from a persistent collection and copies those objects into the transient collection. The program can then manipulate the objects in the transient collection, and once the program terminates, the transient collection ceases to exist. In general, numerous collections—transient or persistent—may contain objects of the same type.

**Multiple Inheritance and Selective Inheritance. Multiple inheritance** occurs when a certain subtype *T* is a subtype of two (or more) types and hence inherits the functions (attributes and methods) of both supertypes. For example, we may create a subtype **ENGINEERING_MANAGER** that is a subtype of both **MANAGER** and **ENGINEER.** This leads to the creation of a **type lattice** rather than a type hierarchy. One problem that can occur with multiple inheritance is that the supertypes from which the subtype inherits may have distinct functions of the same name, creating an ambiguity. For example, both **MANAGER** and **ENGINEER** may have a function called Salary. If the Salary function is implemented by different methods in the **MANAGER** and **ENGINEER** supertypes, an ambiguity exists as to which of the two is inherited by the subtype **ENGINEERING_MANAGER**. It is possible, however, that both **ENGINEER** and **MANAGER** inherit Salary from the same supertype (such as EMPLOYEE) higher up in the lattice. The general rule is that if a function is inherited from some *common supertype,* then it is inherited only once. In such a case, there is no ambiguity;
the problem only arises if the functions are distinct in the two supertypes.
There are several techniques for dealing with ambiguity in multiple inheritance. One solution is to have the system check for ambiguity when the subtype is created, and to let the user explicitly choose which function is to be inherited at this time. A second solution is to use some system default. A third solution is to disallow multiple inheritance altogether if name ambiguity occurs, instead forcing the user to change the name of one of the functions in one of the supertypes. Indeed, some OO systems do not permit multiple inheritance at all.

**Selective inheritance** occurs when a subtype inherits only some of the functions of a supertype. Other functions are not inherited. In this case, an EXCEPT clause may be used to list the functions in a supertype that are *not* to be inherited by the subtype. The mechanism of selective inheritance is not typically provided in ODBs, but it is used more frequently in artificial intelligence applications.

# OBJECT BASED EXTENSIONS TO SQL

## Introduction:

SQL is the standard language for RDBMSs. It was first specified by Chamberlin and Boyce (1974) and underwent enhancements and standardization in 1989 and 1992. The language continued
its evolution with a new standard, initially called SQL3 while being developed and later known as SQL:99 for the parts of SQL3 that were approved into the standard. Starting with the version of SQL known as SQL3, features from object databases were incorporated into the SQL standard. At first, these extensions were known as SQL/Object, but later they were incorporated in the main part of SQL, known as SQL/Foundation in SQL:2008. 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:

- Some **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.
- A mechanism for specifying **object identity** through the use of *reference type* is included.
- **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).
- **Inheritance** mechanisms are provided using the keyword UNDER.

## 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), SQL now provides **user-defined types** (**UDT**s). 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>);

Figure 4 illustrates some of the object concepts in SQL. First, a UDT can be used as either the type for an attribute or as the type for a table. 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. This is similar to

using the *struct* type constructor. For example, in Figure 4(a), the UDT STREET_ADDR_TYPE is used as the type for the STREET_ADDR attribute in the UDT USA_ADDR_TYPE. Similarly, the UDT USA_ADDR_TYPE is in turn used as the type for the ADDR attribute in the UDT PERSON_TYPE in Figure 4(b). If a UDT does
not have any operations, as in the examples in Figure 4(a), it is possible to use the concept of **ROW TYPE** to directly create a structured attribute by using the keyword **ROW**. For example, we could use the following instead of declaring STREET_ADDR_TYPE as a separate type as in Figure 4(a):

```
CREATE TYPE USA_ADDR_TYPE AS (
 STREET_ADDR ROW ( NUMBER VARCHAR (5),
 STREET_NAME VARCHAR (25),
 APT_NO VARCHAR (5),
 SUITE_NO VARCHAR (5) ),
 CITY VARCHAR (25),
 ZIP VARCHAR (10)
 );
```

(a)
```
CREATE TYPE STREET_ADDR_TYPE AS (
    NUMBER          VARCHAR (5),
    STREET          NAME VARCHAR (25),
    APT_NO          VARCHAR (5),
    SUITE_NO        VARCHAR (5)
);
CREATE TYPE USA_ADDR_TYPE AS (
    STREET_ADDR     STREET_ADDR_TYPE,
    CITY            VARCHAR (25),
    ZIP             VARCHAR (10)
);
CREATE TYPE USA_PHONE_TYPE AS (
    PHONE_TYPE      VARCHAR (5),
    AREA_CODE       CHAR (3),
    PHONE_NUM       CHAR (7)
);
```

**Figure 4(a): Using UDTs as types for attributes such as Address and Phone**

```
(b) CREATE TYPE PERSON_TYPE AS (
        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;
    );
```

**Figure 4(b): Specifying UDT for PERSON_TYPE**

To allow for collection types in order to create complex-structured objects, four constructors are now included in SQL: ARRAY, MULTISET, LIST, and SET. In the initial specification of SQL/Object, only the ARRAY type was specified, since it can be used to simulate the other types, but the three additional collection types were included in a later version of the SQL standard. In Figure 4(b), the PHONES attribute of PERSON_TYPE has as its type an array whose elements are of the previously defined
UDT USA_PHONE_TYPE. This array has a maximum of four elements, meaning that we can store up to four phone numbers per person. An array can also have no maximum number of elements if desired. An array type can have its elements referenced using the common notation of square brackets. For example, PHONES[1] refers to the first location value in a PHONES attribute (see Figure 4(b)). A built-in function **CARDINALITY** can return the current number of elements in an array (or any other collection type). For example, PHONES[**CARDINALITY** (PHONES)] refers to the last element in the array. The commonly used dot notation is used to refer to components of a **ROW TYPE** or a UDT. For example, ADDR.CITY refers to the CITY component of an ADDR attribute (see Figure 4(b)).

## Object Identifiers Using Reference Types

Unique system-generated object identifiers can be created via the **reference type** using the keyword **REF**. For example, in Figure 4(b), the phrase:

**REF IS SYSTEM GENERATED**

indicates that whenever a new PERSON_TYPE object is created, the system will assign it a unique system-generated identifier. It is also possible not to have a system generated object identifier and use the traditional keys of the basic relational model if desired.
In general, the user can specify that system-generated object identifiers for the individual rows in a table should be created. By using the syntax:

**REF IS** <OID_ATTRIBUTE> <VALUE_GENERATION_METHOD> ;

the user declares that the attribute named <OID_ATTRIBUTE> will be used to identify individual tuples in the table. The options for <VALUE_GENERATION_METHOD> are SYSTEM GENERATED or DERIVED. In the former case, the system will automatically generate a unique identifier for each tuple. In the latter case, the traditional method of using the user-provided primary key value to identify tuples is applied.

## Creating Tables Based on the UDTs

For each UDT that is specified to be instantiable via the phrase **INSTANTIABLE** (see Figure 4(b)), one or more tables may be created as shown in Figure 4(d), where we create a table PERSON based on the PERSON_TYPE UDT. Also the UDTs in Figure 4(a) are *non instantiable* and hence can only be used as types for attributes, but not as a basis for table creation. In Figure 4(b), the attribute
PERSON_ID will hold the system-generated object identifier whenever a new PERSON record (object) is created and inserted in the table.

## Encapsulation of Operations

In SQL, a **user-defined type** can have its own behavioral specification by specifying methods (or operations) in addition to the attributes. The general form of a UDT specification with methods is as follows:

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

For example, in Figure 4(b), a method Age() was declared that calculates the age of
an individual object of type PERSON_TYPE. The code for implementing the method still has to be written. We can refer to the method implementation by specifying the file that contains the code for the method, or we can write the actual code within the type declaration itself (see Figure 4(b)).

SQL provides certain built-in functions for user-defined types. For a UDT called TYPE_T, the **constructor function** TYPE_T( ) returns a new object of that type. In the new UDT object, every attribute is initialized to its default value. An **observer function** *A* is implicitly created for each attribute *A* to read its value. Hence, *A*(*X*) or *X*.*A* returns the value of attribute *A* of TYPE_T if *X* is a variable that refers to an object/row of type TYPE_T. A **mutator function** for updating an attribute sets the value of the attribute to a new value. SQL allows these functions to be blocked from public use; an EXECUTE privilege is needed to have access to these functions.
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>;

Attributes and functions in UDTs are divided into three categories:

- PUBLIC (visible at the UDT interface)
- PRIVATE (not visible at the UDT interface)
- PROTECTED (visible only to subtypes)

It is also possible to define virtual attributes as part of UDTs, which are computed and updated using functions.

(c)
```
CREATE TYPE GRADE_TYPE AS (
    COURSENO      CHAR (8),
    SEMESTER      VARCHAR (8),
    YEAR          CHAR (4),
    GRADE         CHAR
);
CREATE TYPE STUDENT_TYPE UNDER PERSON_TYPE AS (
    MAJOR_CODE    CHAR (4),
    STUDENT_ID    CHAR (12),
    DEGREE        VARCHAR (5),
    TRANSCRIPT    GRADE_TYPE ARRAY [100]          (continues)
```

**Figure 4(c): Specifying UDTs for STUDENT_TYPE and EMPLOYEE_TYPE as two sub_types of PERSON_TYPE.**

```
    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 AS (
    JOB_CODE        CHAR (4),
    SALARY          FLOAT,
    SSN             CHAR (11)
INSTANTIABLE
NOT FINAL
);
CREATE TYPE MANAGER_TYPE UNDER EMPLOYEE_TYPE AS (
     DEPT_MANAGED CHAR (20)
INSTANTIABLE
);
```

**Figure 4(c): Specifying UDTs for STUDENT_TYPE and EMPLOYEE_TYPE as two sub_types of PERSON_TYPE (contd.)**

## Specifying Inheritance

In SQL, inheritance can be applied to types or to tables. SQL has rules for dealing with **type inheritance** (specified via the **UNDER** keyword). In general, both attributes and instance methods (operations) are inherited. The phrase **NOT FINAL** must be included in a UDT if subtypes are allowed to be created under that UDT (see Figures 4(a) and (b), where PERSON_TYPE, STUDENT_TYPE, and EMPLOYEE_TYPE are declared to be NOT FINAL). There are certain rules associated with type inheritance which can be summarized as follows:

- All attributes are inherited.
- The order of supertypes in the UNDER clause determines the inheritance hierarchy.
- An instance of a subtype can be used in every context in which a supertype

instance is used.

- A subtype can redefine any function that is defined in its supertype, with the restriction that the signature be the same.
- When a function is called, the best match is selected based on the types of all arguments.
- For dynamic linking, the types of the parameters are considered at runtime.

Consider the following examples to illustrate type inheritance, which are illustrated in Figure 4(c). Suppose that we want to create two subtypes of PERSON_TYPE: EMPLOYEE_TYPE and STUDENT_TYPE. In addition, we also create a subtype MANAGER_TYPE that inherits all the attributes (and methods) of EMPLOYEE_TYPE but has an additional attribute DEPT_MANAGED.
In general, we specify the local (specific) attributes and any additional specific methods for the subtype, which inherits the attributes and operations (methods) of its supertype.

Another facility in SQL is **table inheritance** via the supertable/subtable facility. This is also specified using the keyword **UNDER** (see Figure 4(d)). Here, a new record that is inserted into a subtable, say the MANAGER table, is also inserted into its supertables EMPLOYEE and PERSON. Notice that when a record is inserted in MANAGER, we must provide values for all its inherited attributes. INSERT, DELETE, and UPDATE operations are appropriately propagated. The rule is that a tuple in a sub-table must also exist in its super-table to enforce the set/subset constraint on the objects.

(d)
```
CREATE TABLE PERSON OF PERSON_TYPE
    REF IS PERSON_ID SYSTEM GENERATED;
CREATE TABLE EMPLOYEE OF EMPLOYEE_TYPE
    UNDER PERSON;
CREATE TABLE MANAGER OF MANAGER_TYPE
    UNDER EMPLOYEE;
CREATE TABLE STUDENT OF STUDENT_TYPE
    UNDER PERSON;
```

**Figure 4(d): Creating tables based on some of the UDTs and illustrating table inheritance**

## Specifying Relationships via Reference

A component attribute of one tuple may be a **reference** (specified using the keyword **REF**) to a tuple of another (or possibly the same) table as shown in Figure 4(e). The keyword **SCOPE** specifies the name of the table whose tuples can be referenced by the reference attribute. This is similar to a foreign key, except that the system-generated OID value is used rather than the primary key value.
SQL uses a **dot notation** to build **path expressions** that refer to the component attributes of tuples and row types. However, for an attribute whose type is REF, the dereferencing symbol –> is used. For example, the query below retrieves employees working in the company named 'ABCXYZ' by querying the EMPLOYMENT table:

**SELECT** *E*.Employee–>NAME
**FROM** EMPLOYMENT **AS** *E*
**WHERE** *E*.Company–>COMP_NAME = 'ABCXYZ';

In SQL, –> is used for **dereferencing** and has the same meaning assigned to it in the C programming language. Thus, if *r* is a reference to a tuple (object) and *a* is a component attribute in that tuple, then *r* –> *a* is the value of attribute *a* in that tuple. If several relations of the same type exist, SQL provides the SCOPE keyword by which a reference attribute may be made to point to a tuple within a specific table of that type.

```
(e) CREATE TYPE COMPANY_TYPE AS (
        COMP_NAME     VARCHAR (20),
        LOCATION      VARCHAR (20));
    CREATE TYPE EMPLOYMENT_TYPE AS (
        Employee REF (EMPLOYEE_TYPE) SCOPE (EMPLOYEE),
        Company REF (COMPANY_TYPE) SCOPE (COMPANY) );
    CREATE TABLE COMPANY OF COMPANY_TYPE (
        REF IS COMP_ID SYSTEM GENERATED,
        PRIMARY KEY (COMP_NAME) );
    CREATE TABLE EMPLOYMENT OF EMPLOYMENT_TYPE;
```

**Figure 5(e): Specifying relationships using REF and SCOPE**

# ODMG OBJECT MODEL AND OBJECT DEFINITION LANGUAGE

## Introduction

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. Subsequently, 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. The 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.

## Overview of the Object Model of ODMG

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.

**Objects and Literals.** 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*.18 In either case, the value can have a complex structure. The object state can change over time by modifying the object value. A literal is basically a constant value, possibly having a complex structure, but it does not change.

An **object** has five aspects:

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.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. Other important objects in the application may also have unique names, and it is possible to give *more than one* name to an object. 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*. All values of the basic built-in data types are considered to be *literals*.

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:

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 (see Figure 5(b)). 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*.

Figure 5 gives a simplified view of the basic types and type generators of the object model. The notation of ODMG uses three concepts: interface, literal, and class. In the ODMG terminology, the word **behavior** to refer to *operations* and **state** to refer to *properties* (attributes and relationships).

An **interface** specifies only 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.

A **class** specifies both state (attributes) and behavior (operations) of an object type and is **instantiable**. Hence, database and application objects are typically created based on the user-specified class declarations that form a database schema.
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 5 is a simplified version of the object model. In the object model, all objects inherit the basic interface operations of Object, shown in Figure 5(a); these include operations such as copy (creates a new copy of the object), delete (deletes the object), and same_as (compares the object's identity to another object).In general, operations are applied to objects using the **dot notation**. For example, given an object *O*, to compare it with another object *P*, we write

*O*.same_as(*P*)

The result returned by this operation is Boolean and would be true if the identity of *P* is the same as that of *O*, and false otherwise. Similarly, to create a copy *P* of object *O*, we write

*P* = *O*.copy()

An alternative to the dot notation is the **arrow notation:** *O*–>same_as(*P*) or *O*–>copy().

```
(a) interface Object {
        ...
        boolean         same_as(in object other_object);
        object          copy();
        void            delete();
    };
```

**Figure 5(a): The basic Object Interface inherited by all objects.**

```
(b) Class Date : Object {
        enum            Weekday
                            { Sunday, Monday, Tuesday, Wednesday,
                              Thursday, Friday, Saturday };
        enum            Month
                            { January, February, March, April, May, June,
                              July, August, September, October, November,
                              December };
        unsigned short  year();
        unsigned short  month();
        unsigned short  day();
        ...
        boolean         is_equal(in Date other_date);
        boolean         is_greater(in Date other_date);
        ... };
    Class Time : Object {
        ...
        unsigned short  hour();
        unsigned short  minute();
        unsigned short  second();
        unsigned short  millisecond();
        ...
        boolean         is_equal(in Time a_time);
        boolean         is_greater(in Time a_time);
        ...
        Time            add_interval(in Interval an_interval);
        Time            subtract_interval(in Interval an_interval);
        Interval        subtract_time(in Time other_time); };
    class Timestamp : Object {
        ...
        unsigned short  year();
        unsigned short  month();
        unsigned short  day();
        unsigned short  hour();
        unsigned short  minute();
        unsigned short  second();
        unsigned short  millisecond();
        ...
        Timestamp       plus(in Interval an_interval);
                                                    (continues)
```

**Figure 5(b): Some standard interfaces for structured literals.**

```
Timestamp           minus(in Interval an_interval);
boolean             is_equal(in Timestamp a_timestamp);
boolean             is_greater(in Timestamp a_timestamp);
… };
class Interval :    Object {
unsigned short      day();
unsigned short      hour();
unsigned short      minute();
unsigned short      second();
unsigned short      millisecond();
…
Interval            plus(in Interval an_interval);
Interval            minus(in Interval an_interval);
Interval            product(in long a_value);
Interval            quotient(in long a_value);
boolean             is_equal(in interval an_interval);
boolean             is_greater(in interval an_interval);
… };
```

**Figure 5(b): Some standard interfaces for structured literals (contd.)**

```
(c) interface Collection : Object {
        ...
        exception           ElementNotFound{ Object element; };
        unsigned long       cardinality();
        boolean             is_empty();
        ...
        boolean             contains_element(in Object element);
        void                insert_element(in Object element);
        void                remove_element(in Object element)
                                raises(ElementNotFound);
        iterator            create_iterator(in boolean stable);
        ... };
    interface Iterator {
        exception           NoMoreElements();
        ...
        boolean             at_end();
        void                reset();
        Object              get_element() raises(NoMoreElements);
        void                next_position() raises(NoMoreElements);
        ... };
    interface set : Collection {
        set                 create_union(in set other_set);
        ...
        boolean             is_subset_of(in set other_set);
        ... };
    interface bag : Collection {
        unsigned long       occurrences_of(in Object element);
```

**Figure 5(c): Interfaces for collection and Iterators**

```
    bag                 create_union(in Bag other_bag);
    ... };
interface list : Collection {
    exception           Invalid_Index{unsigned_long index; );
    void                remove_element_at(in unsigned long index)
                            raises(InvalidIndex);
    Object              retrieve_element_at(in unsigned long index)
                            raises(InvalidIndex);
    void                replace_element_at(in Object element, in unsigned long index)
                            raises(InvalidIndex);
    void                insert_element_after(in Object element, in unsigned long index)
                            raises(InvalidIndex);
    ...
    void                insert_element_first(in Object element);
    ...
    void                remove_first_element() raises(ElementNotFound);
    ...
    Object              retrieve_first_element() raises(ElementNotFound);
    ...
    list                concat(in list other_list);
    void                append(in list other_list);
};
interface array : Collection {
    exception           Invalid_Index{unsigned_long index; };
    exception           Invalid_Size{unsigned_long size; };
    void                remove_element_at(in unsigned long index)
                            raises(InvalidIndex);
    Object              retrieve_element_at(in unsigned long index)
                            raises(InvalidIndex);
    void                replace_element_at(in unsigned long index, in Object element)
                            raises(InvalidIndex);
    void                resize(in unsigned long new_size)
                            raises(InvalidSize);
};
struct association { Object key; Object value; };
interface dictionary : Collection {
    exception           DuplicateName{string key; };
    exception           KeyNotFound{Object key; };
    void                bind(in Object key, in Object value)
                            raises(DuplicateName);
    void                unbind(in Object key) raises(KeyNotFound);
    Object              lookup(in Object key) raises(KeyNotFound);
    boolean             contains_key(in Object key);
};
```

**Figure 5(c): Interfaces for collection and Iterators (contd.)**

## 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.

## Built-in Interfaces and Classes in the Object Model

Figure 5 shows the built-in interfaces of the object model. All interfaces, such as Collection, Date, and Time, inherit the basic Object interface. In the object model, there is a distinction between collections, whose state contains multiple objects or literals, versus atomic (and structured) objects, whose state is an individual object or literal.

**Collection objects** inherit the basic Collection interface shown in Figure 5(c), which shows the operations for all collection objects. Given a collection object *O*, the *O*.cardinality() operation returns the number of elements in the collection. The operation *O*.is_empty() returns true if the collection *O* is empty, and returns false otherwise. The operations *O*.insert_element(*E*) and *O*.remove_element(*E*)
insert or remove an element *E* from the collection *O*. Finally, the operation *O*.contains_element(*E*) returns true if the collection *O* includes element *E*, and returns false otherwise.

The operation *I* = *O*.create_iterator() creates an **iterator object** *I* for the collection object *O*, which can iterate over each element in the collection. The interface for iterator objects is also shown in Figure 5(c). The *I*.reset() operation sets the iterator at the first element in a collection (for an unordered collection, this would be some arbitrary element), and *I*.next_position() sets the iterator to the next element. The *I*.get_element() retrieves the **current element**, which is the element at which the iterator is currently positioned.

The ODMG object model uses **exceptions** for reporting errors or particular conditions. For example, the ElementNotFound exception in the Collection interface would be raised by the *O*.remove_element(*E*) operation if *E* is not an element in the collection *O*. The NoMoreElements exception in the iterator interface would be raised by the *I*.next_position() operation if the

iterator is currently positioned at the last element in the collection, and hence no more elements exist for the iterator to point to. Collection objects are further specialized into set, list, bag, array, and dictionary, which inherit the operations of the Collection interface.

A **set<*T*> type generator** can be used to create objects such that the value of object *O* is a *set whose elements are of type T.* The Set interface includes the additional operation *P* = *O*.create_union(*S*)
(see Figure 5(c)), which returns a new object *P* of type set<*T*> that is the union of the two sets *O* and *S*. Other operations similar to create_union are create_intersection(*S*) and create_difference(*S*). Operations for set comparison include the *O*.is_subset_of(*S*) operation, which returns true if the set object *O* is a subset of some other set object *S*, and returns false otherwise. Similar operations are is_proper_subset_of(*S*), is_superset_of(*S*), and is_proper_superset_of(*S*).

The **bag<*T*> type generator** allows duplicate elements in the collection and also inherits the Collection interface. It has three operations—create_union(b), create_intersection(b), and create_difference(b)—that all return a new object of type bag<*T*>.

A **list<*T*> type generator** inherits the Collection operations and can be used to create collections of objects of type T where the order of the elements is important. The value of each such object *O* is an *ordered list whose elements are of type T.* Hence, we can refer to the first, last, and *i*th element in the list. Also, when we add an element to the list, we must specify the position in the list where the element is inserted. Some of the list operations are shown in Figure 5(c). If *O* is an object of type list<*T*>, the operation *O*.insert_element_first(*E*) inserts the element *E* before the first element in the
list *O*, so that *E* becomes the first element in the list.The operation *O*.insert_element_after(*E*, *I*) in Figure 5(c) inserts the element *E* after the *i*th element in the list *O* and will raise the exception InvalidIndex if no *i*th element exists in *O*. A similar operation is *O*.insert_element_before(*E*, *I*). To remove elements from the list, the operations are *E* = *O*.remove_first_element(), *E* = *O*.remove_last_element(), and *E* = *O*.remove_element _at(*I*); these operations remove the indicated element from the list *and* return the element as the operation's result. Other operations retrieve an element without removing it from
the list. These are *E* = *O*.retrieve_first_element(), *E* = *O*.retrieve _last_element(), and
*E* = *O*.retrieve_element_at(*I*). Also, two operations to manipulate lists are defined. They
are *P* = *O*.concat(*I*), which creates a new list *P* that is the concatenation of lists *O* and *I* (the elements in list *O* followed by those in list *I*), and *O*.append(*I*), which appends the elements of list *I* to the end of list *O* (without creating a new list object).

The **array<*T*> type generator** also inherits the Collection operations and is similar to list. Specific operations for an array object *O* are *O*.replace_element_at(*I*, *E*), which replaces the array element at position *I* with element *E*; *E* = *O*.remove_element_at(*I*), which retrieves the *i*th element and replaces it with a NULL value; and *E* = *O*.retrieve_element_at(*I*), which simply retrieves the *i*th element of the array. Any of these operations can raise the exception

InvalidIndex if *I* is greater than the array's size. The operation *O*.resize(*N*) changes the number of array elements to *N*.

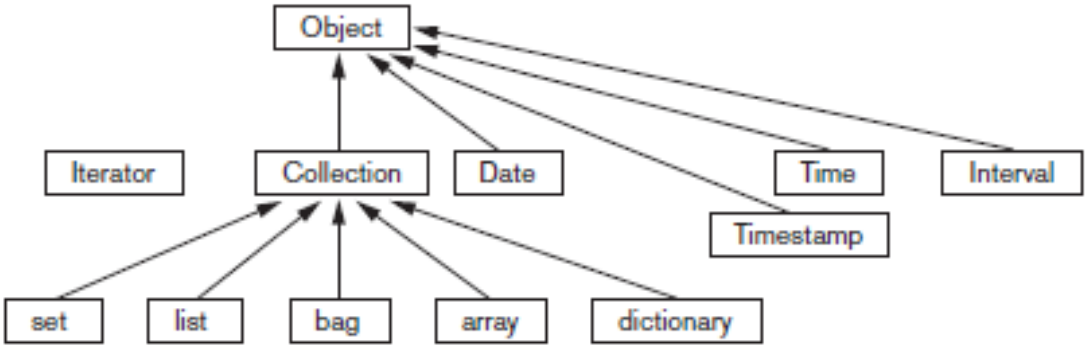



**Figure 6: Inheritance hierarchy for the built-in interfaces of the object model**

The last type of collection objects are of type **dictionary<*K*,*V*>.** This allows the creation of a collection of association pairs <*K*,*V*>, where all *K* (key) values are unique. Making the key values unique allows for associative retrieval of a particular pair given its key value (similar to an index). If *O* is a collection object of type dictionary<*K*,*V*>, then *O*.bind(*K*,*V*) binds value *V* to the key *K* as an association <*K*,*V*> in the collection, whereas *O*.unbind(*K*) removes the association with key *K*
from *O*, and *V* = *O*.lookup(*K*) returns the value *V* associated with key *K* in *O*. The latter two operations can raise the exception KeyNotFound. Finally, *O*.contains_key(*K*) returns true if key *K* exists in *O*, and returns false otherwise.
Figure 6 illustrates the inheritance hierarchy of the built-in constructs of the object model. Operations are inherited from the supertype to the subtype. The collection interfaces described above are *not directly instantiable*; that is, one cannot directly create objects based on these interfaces. Rather, the interfaces can be used to generate user-defined collection types—of type set, bag, list, array, or dictionary—for a particular database application. If an attribute or class has a collection type, say a set, then it will inherit the operations of the set interface.
For example, in a UNIVERSITY database application, the user can specify a type for set<STUDENT>, whose state would be sets of STUDENT objects. The programmer can then use the operations for set<*T*> to manipulate an object of type set<STUDENT>.

## Atomic (User-Defined) Objects

Object types for *atomic objects* can be constructed using the keyword class in ODL. In the object model, any user-defined object that is not a collection object is called an **atomic object**. For example, in a UNIVERSITY database application, the user can specify an object type (class) for

STUDENT objects. Most such objects will be **structured objects**; for example, a STUDENT object will have a complex structure, with many attributes, relationships, and operations, but it is still considered atomic because it is not a collection. Such a user-defined atomic object type is defined as a class by specifying its **properties** and **operations**. The properties define the state of the object and are further distinguished into **attributes** and **relationships**. Thus, a user-defined object type for atomic (structured) objects can include three types of components—attributes, relationships, and
operations.

We illustrate our discussion with the two classes EMPLOYEE and DEPARTMENT shown in Figure 7.

An **attribute** is a property that describes some aspect of an object. Attributes have values (which are typically literals having a simple or complex structure) that are stored within the object. However, attribute values can also be Object_ids of other objects. Attribute values can even be specified via methods that are used to calculate the attribute value. In Figure 7 the attributes for EMPLOYEE are Name, Ssn, Birth_date, Sex, and Age, and those for DEPARTMENT are Dname, Dnumber, Mgr, Locations, and Projs. The Mgr and Projs attributes of DEPARTMENT have complex structure and are defined via **struct**, which corresponds to the *tuple constructor* . Hence, the value of Mgr in each DEPARTMENT object will have two components: Manager, whose value is an Object_id that references the EMPLOYEE
object that manages the DEPARTMENT, and Start_date, whose value is a date. The locations attribute of DEPARTMENT is defined via the set constructor, since each DEPARTMENT object can have a set of locations.

A **relationship** is a property that specifies that two objects in the database are related.

In the object model of ODMG, only binary relationships are explicitly represented, and each binary relationship is represented by a *pair of inverse references* specified via the keyword relationship. In Figure 7, one relationship exists that relates each EMPLOYEE to the DEPARTMENT in which he or she
works—the Works_for relationship of EMPLOYEE. In the inverse direction, each DEPARTMENT is related to the set of EMPLOYEES that work in the DEPARTMENT— the Has_emps relationship of DEPARTMENT. The keyword **inverse** specifies that these two properties define a single conceptual relationship in inverse directions. By specifying inverses, the database system can maintain the referential integrity of the relationship automatically. That is, if the value of Works_for for a particular EMPLOYEE *E* refers to DEPARTMENT *D*, then the value of Has_emps for DEPARTMENT *D* must include a reference to *E* in its set of EMPLOYEE references. If the database designer desires to have a relationship to be represented in *only one direction*, then it has to be modeled as an attribute (or operation). An example is the Manager component of the Mgr attribute in DEPARTMENT.

In addition to attributes and relationships, the designer can include **operations** in object type (class) specifications. Each object type can have a number of **operation signatures**, which specify the operation name, its argument types, and its returned value, if applicable. Operation

names are unique within each object type, but they can be overloaded by having the same operation name appear in distinct object types. The operation signature can also specify the names of **exceptions** that can occur during operation execution. The implementation of the operation will include the code to raise these exceptions. In Figure 7 the EMPLOYEE class has one operation: reassign_emp, and the DEPARTMENT class has two operations: add_emp and change_manager.

```
class EMPLOYEE
(     extent              ALL_EMPLOYEES
      key                 Ssn   )
{
      attribute           string                  Name;
      attribute           string                  Ssn;
      attribute           date Birth_date;
      attribute           enum Gender(M, F)       Sex;
      attribute           short                   Age;
      relationship        DEPARTMENT              Works_for
                              inverse DEPARTMENT::Has_emps;
      void                reassign_emp(in string New_dname)
                              raises(dname_not_valid);
};
class DEPARTMENT
(     extent              ALL_DEPARTMENTS
      key                 Dname, Dnumber )
{
      attribute           string                  Dname;
      attribute           short                   Dnumber;
      attribute           struct Dept_mgr {EMPLOYEE Manager, date Start_date}
                              Mgr;
      attribute           set<string>             Locations;
      attribute           struct Projs {string Proj_name, time Weekly_hours)
                              Projs;
      relationship        set<EMPLOYEE>           Has_emps inverse EMPLOYEE::Works_for;
      void                add_emp(in string New_ename) raises(ename_not_valid);
      void                change_manager(in string New_mgr_name; in date
                              Start_date);
};
```

**Figure 7: The attributes, relationships and operations in a class definition.**

## Extents, Keys, and Factory Objects

In the ODMG object model, the database designer can declare an *extent* (using the keyword **extent**) for any object type that is defined via a **class** declaration. The extent is given a name, and it will contain all persistent objects of that class. Hence, the extent behaves as a *set object* that holds all persistent objects of the class. In Figure 7 the EMPLOYEE and DEPARTMENT classes have extents called ALL_EMPLOYEES and ALL_DEPARTMENTS, respectively. This is similar to creating two objects—one of type set<EMPLOYEE> and the second of type set<DEPARTMENT>—and making them persistent by naming them ALL_EMPLOYEES and ALL_DEPARTMENTS.

Extents are also used to automatically enforce the set/subset relationship between the extents of a supertype and its subtype. If two classes A and B have extents ALL_A and ALL_B, and class B is a subtype of class A (that is, class B extends class A), then the collection of objects in ALL_B must be a subset of those in ALL_A at any point. This constraint is automatically enforced by the database system.

A class with an extent can have one or more keys. A **key** consists of one or more properties (attributes or relationships) whose values are constrained to be unique for each object in the extent. For example, in Figure 7 the EMPLOYEE class has the Ssn attribute as key (each EMPLOYEE object in the extent must have a unique Ssn value), and the DEPARTMENT class has two distinct keys: Dname and Dnumber (each DEPARTMENT must have a unique Dname and a unique Dnumber). For a composite key that is made of several properties, the properties that form the key are contained in parentheses. For example, if a class VEHICLE with an extent ALL_VEHICLES has a key made up of a combination of two attributes State and License_number, they would be placed in parentheses as (State, License_number) in the key declaration.

F**actory object is** an object that can be used to generate or create individual objects via its operations. Some of the interfaces of factory objects that are part of the ODMG object model are shown in Figure 8. The interface ObjectFactory has a single operation, new(), which returns a new object with an Object_id. By inheriting this interface, users can create their own factory interfaces for each user-defined (atomic) object type, and the programmer can implement the operation *new* differently for each type of object. Figure 8 also shows a DateFactory interface, which has additional operations for creating a new calendar_date and for creating an object whose value is the current_date, among other operations. A factory object basically provides the **constructor operations** for new objects.

Because an ODB system can create many different databases, each with its own schema, the ODMG object model has interfaces for DatabaseFactory and Database objects, as shown in Figure 8. Each database has its own *database name*, and the **bind** operation can be used to assign individual unique names to persistent objects in a particular database. The **lookup** operation returns an object from the database that has the specified persistent object_name, and the **unbind** operation removes the name of a persistent named object from the database.

```
interface ObjectFactory {
    Object          new();
};
interface SetFactory : ObjectFactory {
    Set             new_of_size(in long size);
};
interface ListFactory : ObjectFactory {
    List            new_of_size(in long size);
};
interface ArrayFactory : ObjectFactory {
    Array           new_of_size(in long size);
};
interface DictionaryFactory : ObjectFactory {
    Dictionary      new_of_size(in long size);
};
interface DateFactory : ObjectFactory {
    exception       InvalidDate{};
    ...
    Date            calendar_date(    in unsigned short year,
                                      in unsigned short month,
                                      in unsigned short day )
                    raises(InvalidDate);
    ...
    Date            current();
};
interface DatabaseFactory {
    Database        new();
};
interface Database {
    ...
    void            open(in string database_name)
                        raises(DatabaseNotFound, DatabaseOpen);
    void            close() raises(DatabaseClosed, ...);
    void            bind(in Object an_object, in string name)
                        raises(DatabaseClosed, ObjectNameNotUnique, ...);
    Object          unbind(in string name)
                        raises(DatabaseClosed, ObjectNameNotFound, ...);
    Object          lookup(in string object_name)
                        raises(DatabaseClosed, ObjectNameNotFound, ...);
    ... };
```

**Figure 8:Interfaces to illustrate factory objects and database objects.**

## The Object Definition Language ODL

The ODL is designed to support the semantic constructs of the ODMG object model and is independent of any particular 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.

Figure 9(b) shows a possible object schema for part of the UNIVERSITY database. The graphical notation for Figure 9(b) is shown in Figure 9(a) and can be considered as a variation of EER

diagrams with the added concept of interface inheritance but without several EER concepts, such as categories (union types) and attributes of relationships.

Figure 10 shows the straightforward way of mapping part of the UNIVERSITY database. Entity types are mapped into ODL classes, and inheritance is done using **extends**. However, there is no direct way to map categories (union types) or to do multiple inheritance. In Figure 10 the classes PERSON, FACULTY, STUDENT, and GRAD_STUDENT have the extents PERSONS, FACULTY, STUDENTS, and GRAD_STUDENTS, respectively. Both FACULTY and STUDENT **extends** PERSON and GRAD_STUDENT **extends** STUDENT. Hence, the collection of STUDENTS (and the collection of FACULTY) will be constrained to be a subset of the collection of PERSONs at any time. Similarly, the collection of GRAD_STUDENTs will be a subset of STUDENTs. At the same time, individual STUDENT and FACULTY objects will inherit the properties (attributes and relationships) and operations of PERSON, and individual GRAD_STUDENT objects will inherit those of STUDENT.

The classes DEPARTMENT, COURSE, SECTION, and CURR_SECTION in Figure 10 are straightforward mappings of the corresponding entity types in Figure 9(b). However,the GRADE class corresponds

to the M:N relationship between STUDENT and SECTION in Figure 9(b).The reason it was made into a separate class (rather than as a pair of inverse relationships) is because it includes the relationship attribute Grade. Hence, the M:N relationship is mapped to the class GRADE, and a pair of 1:N relationships, one between STUDENT and GRADE and the other between SECTION and GRADE.These relationships are represented by the following relationship properties:

Completed_sections of STUDENT; Section and Student of GRADE; and Students of

SECTION (see Figure 10). Finally, the class DEGREE is used to represent the composite, multivalued attribute.

We consider another example to illustrate interfaces and interface (behavior) inheritance.

Figure 11(a) is part of a database schema for storing geometric objects. An interface GeometryObject is specified, with operations to calculate the perimeter and area of a geometric object, plus operations to translate (move) and rotate an object. Several classes (RECTANGLE, TRIANGLE, CIRCLE, …) inherit the GeometryObject interface. Since GeometryObject is an interface, it is *noninstantiable*—that is, no

objects can be created based on this interface directly. However, objects of type RECTANGLE, TRIANGLE, CIRCLE, … can be created, and these objects inherit all the operations of the GeometryObject interface. Note that with interface inheritance, only operations are inherited, not properties (attributes, relationships). Hence, if a property is needed in the inheriting class, it must be repeated in the class definition, as with the Reference_point attribute in Figure 11(b). Notice that the inherited operations can have different implementations in each class. For example, the implementations of the area and perimeter operations may be different for RECTANGLE, TRIANGLE, and CIRCLE.

*Multiple inheritance* of interfaces by a class is allowed, as is multiple inheritance of interfaces by another interface. However, with **extends** (class) inheritance, multiple inheritance is *not permitted*. Hence, a class can inherit via **extends** from at most one class (in addition to inheriting from zero or more interfaces).

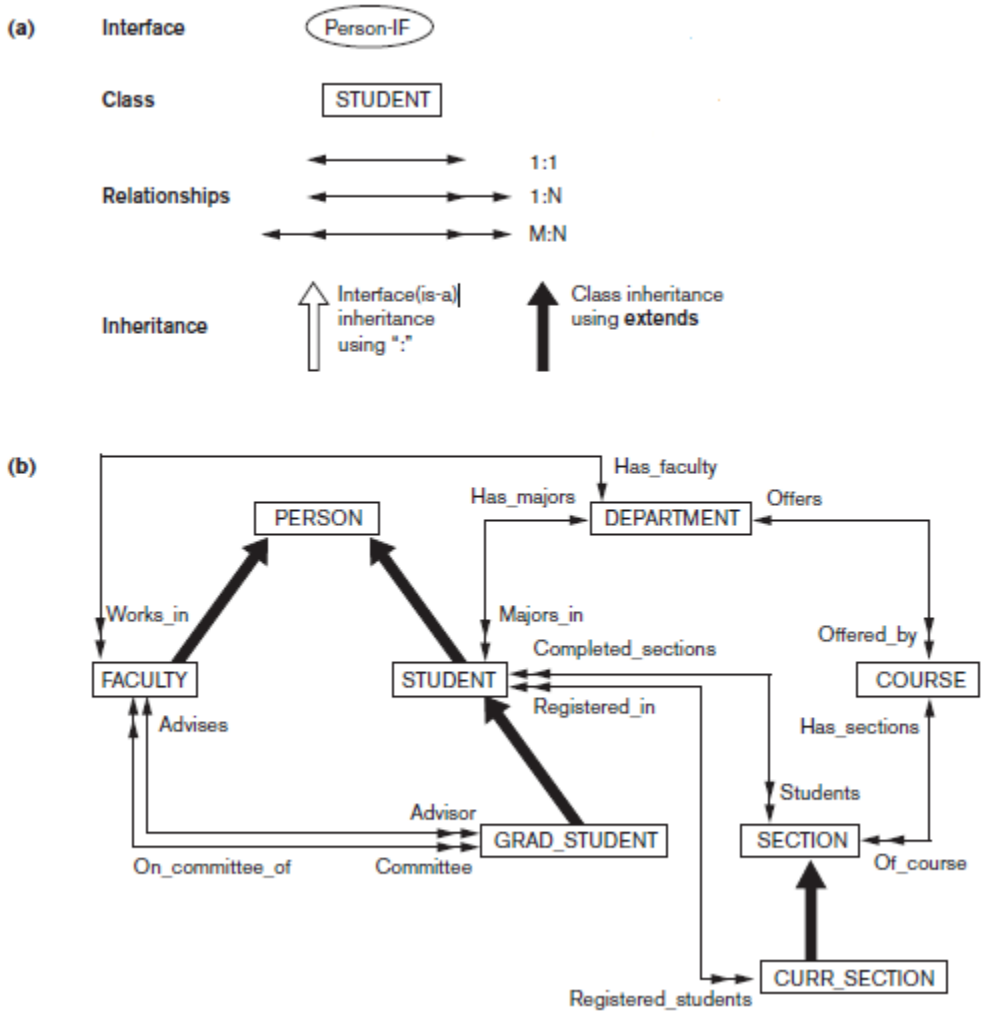


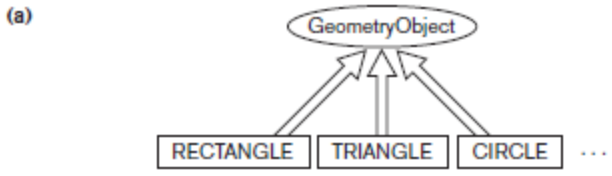

**Figure 9: An example of a database schema. (a) Graphical notation for representing ODL schemas. (b) A graphical object database schema for part of the UNIVERSITY database.**

```
class PERSON
(   extent      PERSONS
    key         Ssn )
{   attribute   struct Pname {  string  Fname,
                                string  Mname,
                                string  Lname }     Name;
    attribute   string                              Ssn;
    attribute   date                                Birth_date;
    attribute   enum Gender{M, F}                   Sex;
    attribute   struct Address {  short   No,
                                  string  Street,
                                  short   Apt_no,
                                  string  City,
                                  string  State,
                                  short   Zip }     Address;
    short       Age();  };
class FACULTY extends PERSON
(   extent      FACULTY )
{   attribute   string          Rank;
    attribute   float           Salary;
    attribute   string          Office;
    attribute   string          Phone;
    relationship    DEPARTMENT  Works_in inverse DEPARTMENT::Has faculty;
    relationship    set<GRAD_STUDENT> Advises inverse GRAD_STUDENT::Advisor;
    relationship    set<GRAD_STUDENT> On_committee_of inverse GRAD_STUDENT::Committee;
    void        give_raise(in float raise);
    void        promote(in string new rank); };
class GRADE
(   extent GRADES )
{
    attribute       enum GradeValues{A,B,C,D,F,I, P} Grade;
    relationship    SECTION Section inverse SECTION::Students;
    relationship STUDENT Student inverse STUDENT::Completed_sections; };
class STUDENT extends PERSON
(   extent      STUDENTS )
{   attribute   string          Class;
    attribute   Department      Minors_in;
    relationship    Department Majors_in inverse DEPARTMENT::Has_majors;
    relationship    set<GRADE> Completed_sections inverse GRADE::Student;
    relationship    set<CURR_SECTION> Registered_in INVERSE CURR_SECTION::Registered_students;
    void        change_major(in string dname) raises(dname_not_valid);
    float       gpa();
    void        register(in short secno) raises(section_not_valid);
    void        assign_grade(in short secno; IN GradeValue grade)
                    raises(section_not_valid,grade_not_valid); };
```

**Figure 10:Possible ODL schema for the UNIVERSITY database in figure 8(b).**

```
class DEGREE
{   attribute       string              College;
    attribute       string              Degree;
    attribute       string              Year; };
class GRAD_STUDENT extends STUDENT
(   extent          GRAD_STUDENTS )
{   attribute       set<Degree>         Degrees;
    relationship    Faculty advisor inverse FACULTY::Advises;
    relationship    set<FACULTY>  Committee inverse FACULTY::On_committee_of;
    void            assign_advisor(in string Lname; in string Fname)
                        raises(faculty_not_valid);
    void            assign_committee_member(in string Lname; in string Fname)
                        raises(faculty_not_valid); };
class DEPARTMENT
(   extent          DEPARTMENTS
    key             Dname )
{   attribute       string              Dname;
    attribute       string              Dphone;
    attribute       string              Doffice;
    attribute       string              College;
    attribute       FACULTY             Chair;
    relationship    set<FACULTY> Has_faculty inverse FACULTY::Works_in;
    relationship    set<STUDENT> Has_majors inverse STUDENT::Majors_in;
    relationship    set<COURSE> Offers inverse COURSE::Offered_by; };
class COURSE
(   extent          COURSES
    key             Cno )
{   attribute       string              Cname;
    attribute       string              Cno;
    attribute       string              Description;
    relationship    set<SECTION> Has_sections inverse SECTION::Of_course;
    relationship    <DEPARTMENT> Offered_by inverse DEPARTMENT::Offers; };
class SECTION
(   extent          SECTIONS )
{   attribute       short               Sec_no;
    attribute       string              Year;
    attribute       enum Quarter{Fall, Winter, Spring, Summer}
                        Qtr;
    relationship    set<Grade> Students inverse Grade::Section;
    relationship    COURSE Of_course inverse COURSE::Has_sections; };
class CURR_SECTION extends SECTION
(   extent          CURRENT_SECTIONS )
{   relationship    set<STUDENT> Registered_students
                        inverse STUDENT::Registered_in
    void            register_student(in string Ssn)
                        raises(student_not_valid, section_full); };
```

**Figure 11:Possible ODL schema for the UNIVERSITY database in figure 8(b) contd**




```
(b) interface GeometryObject
    {   attribute    enum      Shape{RECTANGLE, TRIANGLE, CIRCLE, ... }
                                   Shape;
        attribute    struct    Point {short x, short y} Reference_point;
        float        perimeter();
        float        area();
        void         translate(in short x_translation; in short y_translation);
        void         rotate(in float angle_of_rotation); };
    class RECTANGLE : GeometryObject
    (   extent       RECTANGLES )
    {   attribute    struct    Point {short x, short y} Reference_point;
        attribute    short     Length;
        attribute    short     Height;
        attribute    float     Orientation_angle; };
    class TRIANGLE : GeometryObject
    (   extent       TRIANGLES )
    {   attribute    struct    Point {short x, short y} Reference_point;
        attribute    short     Side_1;
        attribute    short     Side_2;
        attribute    float     Side1_side2_angle;
        attribute    float     Side1_orientation_angle; };
    class CIRCLE : GeometryObject
    (   extent       CIRCLES )
    {   attribute    struct    Point {short x, short y} Reference_point;
        attribute    short     Radius; };
    ...
```

**Figure 11:An illustration of interface inheritance via ":". (a) Graphical Schema Representation (b) Corresponding interface and class definitions in ODL.**

**For any queries contact: fazilishifaa@gmail.com**