part 5

Object, Object-Relational, and XML: Concepts, Models, Languages, and Standards

This page intentionally left blank

chapter 12

Object and Object-Relational Databases

In this chapter, we discuss the features of object-oriented data models and show how some of these features have been incorporated in relational database systems and the SQL standard. Some features of object data models have also been incorporated into the data models of newer types of database systems, known as NOSQL systems (see Chapter 24). In addition, the XML model (see Chapter 13) has similarities to the object model. So an introduction to the object model will give a good perspective on many of the recent advances in database technology. Database systems that were based on the object data model were known originally as object-oriented databases (OODBs) but are now referred to as **object databases** (**ODBs**).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 CIM[1]), biological and other sciences, telecommunications, geographic information systems, and multimedia.[2] These ODBs were developed for applications that have requirements requiring more complex structures for stored objects. A key feature of object databases is the power they give the designer to specify both the *structure* of complex objects and the *operations* that can be applied to these objects.

[1]Computer-aided design/computer-aided manufacturing and computer-integrated manufacturing.

[2]Multimedia databases must store various types of multimedia objects, such as video, audio, images, graphics, and documents (see Chapter 26).

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. Databases are fundamental components in many software systems, and traditional databases are sometimes difficult to use with software applications that are developed in an object-oriented programming language such as C++ or Java. Object databases are designed so they can be directly—or *seamlessly*—integrated with software that is developed using object-oriented programming languages.

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. A recent version of the SQL standard (2008) for RDBMSs, known as SQL/Foundation, includes many of these features, which were originally known as SQL/Object and have now been merged into the main SQL specification.

Although many experimental prototypes and commercial object-oriented database systems have been created, they have not found widespread use because of the popularity of relational and object-relational systems. The experimental prototypes included the Orion system developed at MCC, OpenOODB at Texas Instruments, the Iris system at Hewlett-Packard laboratories, the Ode system at AT&T Bell Labs, and the ENCORE/ObServer project at Brown University. Commercially available systems included GemStone Object Server of GemStone Systems, ONTOS DB of Ontos, Objectivity/DB of Objectivity Inc., Versant Object Database and FastObjects by Versant Corporation (and Poet), ObjectStore of Object Design, and Ardent Database of Ardent.

As commercial object DBMSs became available, the need for a standard model and language was recognized. Because the formal procedure for approval of standards normally takes a number of years, a consortium of object DBMS vendors and users, called ODMG, 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.[3] In Section 12.1, we describe the key concepts utilized in many object database systems and that were later incorporated into object-relational systems and the SQL standard. These include *object identity*, *object structure* and *type constructors*, *encapsulation of operations*, and the definition of *methods* as part of class declarations, mechanisms for storing objects in a database by making them *persistent,* and *type and class hierarchies* and *inheritance*. Then, in Section 12.2 we see how these concepts have been incorporated into the latest SQL standards, leading to object-relational databases. Object features were originally introduced in SQL:1999, and then updated in SQL:2008. In Section 12.3, we turn our attention to "pure" object database standards by presenting features of the object database standard ODMG 3.0 and the object definition

[3]Similar concepts were also developed in the fields of semantic data modeling and knowledge representation.

language ODL. Section 12.4 presents an overview of the database design process for object databases. Section 12.5 discusses the object query language (OQL), which is part of the ODMG 3.0 standard. In Section 12.6, we discuss programming language bindings, which specify how to extend object-oriented programming languages to include the features of the object database standard. Section 12.7 summarizes the chapter. Sections 12.3 through 12.6 may be left out if a less thorough introduction to object databases is desired.

12.1 Overview of Object Database Concepts

12.1.1 Introduction to Object-Oriented Concepts and Features

The term *object-oriented*—abbreviated *OO* or *O-O*—has its origins in OO programming languages, or OOPLs. Today OO concepts are applied in the areas of databases, software engineering, knowledge bases, artificial intelligence, and computer systems in general. OOPLs have their roots in the SIMULA language, which was proposed in the late 1960s. The programming language Smalltalk, developed at Xerox PARC[4] 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. An example of the latter is 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.[5] 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,

[4]Palo Alto Research Center, Palo Alto, California.

[5]Objects have many other characteristics, as we discuss in the rest of this chapter.

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 (see Chapter 2).

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. The ODMG object database standard has recognized this need and it explicitly represents binary relationships via a pair of *inverse references*, as we will describe in Section 12.3.

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.

In the next several sections, we discuss in some detail the main characteristics of object databases. Section 12.1.2 discusses object identity; Section 12.1.3 shows how the types for complex-structured objects are specified via type constructors; Section 12.1.4 discusses encapsulation and persistence; and Section 12.1.5 presents inheritance concepts. Section 12.1.6 discusses some additional OO concepts, and Section 12.1.7 gives a summary of all the OO concepts that we introduced. In Section 12.2, we show how some of these concepts have been incorporated into the SQL:2008 standard for relational databases. Then in Section 12.3, we show how these concepts are realized in the ODMG 3.0 object database standard.

12.1.2 Object Identity, and Objects versus Literals

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 interobject 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. 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 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. However, some early ODMSs have used the physical address as the OID to increase the efficiency of object retrieval. 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. 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.

12.1.3 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. 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.

1. One type constructor has been called the **atom** constructor, although this term is not used in the latest object standard. 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. A second type constructor is referred to as the **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>. To create complex nested type structures in the object model, the *collection* type constructors are needed, which we discuss next. Notice that the type constructors *atom* and *struct* are the only ones available in the original (basic) relational model.

3. **Collection** (or *multivalued*) type constructors 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 *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. The *tuple constructor* can create structured values and objects of the form $<a_1:i_1, a_2:i_2, \ldots, a_n:i_n>$, where each a_j is an attribute name[6] and each i_j is a value or an OID.

The other commonly used constructors are collectively referred to as collection types but have individual differences among them. The **set constructor** will create objects or literals that are a set of *distinct* elements $\{i_1, i_2, \ldots, i_n\}$, 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* $[i_1, i_2, \ldots, i_n]$ 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**)[7] that incorporates the preceding type constructors can be used to define the object types for a particular database application. In Section 12.3 we will describe the standard ODL of ODMG, but first we introduce the concepts gradually in this section using a simpler notation. The type constructors can be used to define the *data structures* for an OO *database schema.* Figure 12.1 shows how we may declare EMPLOYEE and DEPARTMENT types.

In Figure 12.1, 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

[6]Also called an *instance variable name* in OO terminology.

[7]This corresponds to the DDL (data definition language) of the database system (see Chapter 2).

```
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 12.1
Specifying the object types EMPLOYEE, DATE, and DEPARTMENT using type constructors.

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 12.1 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. We will see in Section 12.3 how the ODMG standard allows inverses to be explicitly declared as relationship attributes to ensure that inverse references are consistent.

12.1.4 Encapsulation of Operations and Persistence of Objects

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. In Section 12.5 we will describe the OQL query language that is proposed as a standard query language for ODBs.

The term **class** is often used to refer to a type definition, along with the definitions of the operations for that type.[8] Figure 12.2 shows how the type definitions in Figure 12.1 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 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.

[8]This definition of *class* is similar to how it is used in the popular C++ programming language. The ODMG standard uses the word *interface* in addition to *class* (see Section 12.3). In the EER model, the term *class* was used to refer to an object type, along with the set of all objects of that type (see Chapter 8).

```
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 12.2
Adding operations to the definitions of EMPLOYEE and DEPARTMENT.

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, although this was not used in Figure 12.2.[9] The dot notation is also used to refer to attributes of an object—for example, by writing d.Dnumber or d.Mgr_Start_date.

[9]Default names for the constructor and destructor operations exist in the C++ programming language. For example, for class EMPLOYEE, the *default constructor name* is EMPLOYEE and the *default destructor name* is ~EMPLOYEE. It is also common to use the *new* operation to create *new* objects.

Specifying Object Persistence via Naming and Reachability. 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 12.3. 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*.

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* (see Section 12.3).

For example, we can define a class DEPARTMENT_SET (see Figure 12.3) whose objects are of type set(DEPARTMENT).[10] We can create an object of type DEPARTMENT_SET, and give it a persistent name ALL_DEPARTMENTS, as shown in Figure 12.3. 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. As we will see in Section 12.3, the ODMG ODL standard gives the schema designer the option of naming an extent as part of class definition.

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 12.3.[11] This allows transient and persistent objects to follow the

[10] As we will see in Section 12.3, the ODMG ODL syntax uses **set**<DEPARTMENT> instead of **set**(DEPARTMENT).

[11] Some systems, such as POET, automatically create the extent for a class.

```
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 12.3
Creating persistent objects by naming and reachability.

same type and class declarations of the ODL and the OOPL. In general, it is possible to define several persistent collections for the same class definition, if desired.

12.1.5 Type Hierarchies and Inheritance

Simplified Model for Inheritance. Another main characteristic of ODBs is that they allow type hierarchies and inheritance. We use a simple OO model in this section—a model in which attributes and operations are treated uniformly—since both attributes and operations can be inherited. In Section 12.3, we will discuss the inheritance model of the ODMG standard, which differs from the model discussed here because it distinguishes between *two types of inheritance*. Inheritance allows the 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.[12] In the simplified model we use in this section, the attributes and operations are together called *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). We use the term **function** to refer to both attributes *and* operations, since they are treated similarly in a basic introduction to inheritance.[13]

[12]In this section we will use the terms *type* and *class* as meaning the same thing—namely, the attributes *and* operations of some type of object.

[13]We will see in Section 12.3 that types with functions are similar to the concept of interfaces as used in ODMG ODL.

A type in its simplest form has a **type name** and a list of visible (*public*) **functions**. When specifying a type in this section, we use the following format, which does not specify arguments of functions, to simplify the discussion:

TYPE_NAME: function, function, … , function

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. Now suppose that we 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

Notice that 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.

Notice that type definitions describe objects but *do not* generate objects on their own. When an object is created, typically it belongs to one or more of these types that have been declared. For example, a circle object is of type CIRCLE and GEOMETRY_OBJECT (by inheritance). Each object also becomes a member of one or more persistent collections of objects (or extents), which are used to group together collections of objects that are persistently stored in the database.

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.[14]

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. In the ODMG model (see Section 12.3), 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.

[14]This is called OBJECT in the ODMG model (see Section 12.3).

The inheritance model discussed in this section is very simple. As we will see in Section 12.3, the ODMG model distinguishes between type inheritance—called *interface inheritance* and denoted by a colon (:)—and the *extent inheritance* constraint—denoted by the keyword EXTEND.

12.1.6 Other Object-Oriented Concepts

Polymorphism of Operations (Operator Overloading). Another characteristic of OO systems in general is that they provide for **polymorphism** of operations, which is also known as **operator overloading**. This concept allows the same *operator name* or *symbol* to be bound to two or more different *implementations* of the operator, depending on the type of objects to which the operator is applied. A simple example from programming languages can illustrate this concept. In some languages, the operator symbol "+" can mean different things when applied to operands (objects) of different types. If the operands of "+" are of type *integer*, the operation invoked is integer addition. If the operands of "+" are of type *floating point*, the operation invoked is floating-point addition. If the operands of "+" are of type *set*, the operation invoked is set union. The compiler can determine which operation to execute based on the types of operands supplied.

In OO databases, a similar situation may occur. We can use the GEOMETRY_OBJECT example presented in Section 12.1.5 to illustrate operation polymorphism[15] in ODB. In this example, the function Area is declared for all objects of type GEOMETRY_OBJECT. However, the implementation of the method for Area may differ for each subtype of GEOMETRY_OBJECT. One possibility is to have a general implementation for calculating the area of a generalized GEOMETRY_OBJECT (for example, by writing a general algorithm to calculate the area of a polygon) and then to rewrite more efficient algorithms to calculate the areas of specific types of geometric objects, such as a circle, a rectangle, a triangle, and so on. In this case, the Area function is *overloaded* by different implementations.

The ODMS must now select the appropriate method for the Area function based on the type of geometric object to which it is applied. In strongly typed systems, this can be done at compile time, since the object types must be known. This is termed **early** (or **static**) **binding**. However, in systems with weak typing or no typing (such as Smalltalk, LISP, PHP, and most scripting languages), the type of the object to which a function is applied may not be known until runtime. In this case, the function must check the type of object at runtime and then invoke the appropriate method. This is often referred to as **late** (or **dynamic**) **binding**.

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

[15]In programming languages, there are several kinds of polymorphism. The interested reader is referred to the Selected Bibliography at the end of this chapter for works that include a more thorough discussion.

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. In the object database standard (see Section 12.3), multiple inheritance is allowed for operation inheritance of interfaces, but is not allowed for EXTENDS inheritance of classes.

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.[16]

12.1.7 Summary of Object Database Concepts

To conclude this section, we give a summary of the main concepts used in ODBs and object-relational systems:

- **Object identity.** Objects have unique identities that are independent of their attribute values and are generated by the ODB system.
- **Type constructors.** Complex object structures can be constructed by applying in a nested manner a set of basic type generators/constructors, such as tuple, set, list, array, and bag.
- **Encapsulation of operations.** Both the object structure and the operations that can be applied to individual objects are included in the class/type definitions.
- **Programming language compatibility.** Both persistent and transient objects are handled seamlessly. Objects are made persistent by being reachable from

[16]In the ODMG model, type inheritance refers to inheritance of operations only, not attributes (see Section 12.3).

a persistent collection (extent) or by explicit naming (assigning a unique name by which the object can be referenced/retrieved).

- **Type hierarchies and inheritance.** Object types can be specified by using a type hierarchy, which allows the inheritance of both attributes and methods (operations) of previously defined types. Multiple inheritance is allowed in some models.
- **Extents.** All persistent objects of a particular class/type C can be stored in an extent, which is a named persistent object of type set(C). Extents corresponding to a type hierarchy have set/subset constraints enforced on their collections of persistent objects.
- **Polymorphism and operator overloading.** Operations and method names can be overloaded to apply to different object types with different implementations.

In the following sections we show how these concepts are realized, first in the SQL standard (Section 12.2) and then in the ODMG standard (Section 12.3).

12.2 Object Database Extensions to SQL

We introduced SQL as the standard language for RDBMSs in Chapters 6 and 7. As we discussed, SQL 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**. Additional revisions were made to SQL in 2003 and 2006 to add features related to XML (see Chapter 13).

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.

We now discuss each of these concepts in more detail. In our discussion, we will refer to the example in Figure 12.4.

12.2.1 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 discussed in Section 12.1), 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 12.4 illustrates some of the object concepts in SQL. We will explain the examples in this figure gradually as we explain the concepts. 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, much like that achieved by nesting type constructors/generators as discussed in Section 12.1. This is similar to using the *struct* type constructor of Section 12.1.3. For example, in Figure 12.4(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 12.4(b). If a UDT does not have any operations, as in the examples in Figure 12.4(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 12.4(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)
    );
```

To allow for collection types in order to create complex-structured objects, four constructors are now included in SQL: ARRAY, MULTISET, LIST, and SET. These are

Figure 12.4
Illustrating some of the object features of SQL. (a) Using UDTs as types for attributes such as Address and Phone, (b) specifying UDT for PERSON_TYPE, (c) specifying UDTs for STUDENT_TYPE and EMPLOYEE_TYPE as two subtypes of PERSON_TYPE.

```
(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)
    );

(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;
    );

(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 12.4 (continued) Illustrating some of the object features of SQL. (c) (continued) Specifying UDTs for STUDENT_TYPE and EMPLOYEE_TYPE as two subtypes of PERSON_TYPE, (d) Creating tables based on some of the UDTs, and illustrating table inheritance, (e) Specifying relationships using REF and SCOPE.

```
    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
    );

(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;

(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;
```

similar to the type constructors discussed in Section 12.1.3. 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 12.4(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 12.4(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 12.4(b)).

12.2.2 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 12.4(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.

12.2.3 Creating Tables Based on the UDTs

For each UDT that is specified to be instantiable via the phrase **INSTANTIABLE** (see Figure 12.4(b)), one or more tables may be created. This is illustrated in Figure 12.4(d), where we create a table PERSON based on the PERSON_TYPE UDT. Notice that the UDTs in Figure 12.4(a) are *noninstantiable* and hence can only be used as

types for attributes, but not as a basis for table creation. In Figure 12.4(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.

12.2.4 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 12.4(b), we declared a method Age() 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 12.4(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>;
```

Two types of functions can be defined: internal SQL and external. Internal functions are written in the extended PSM language of SQL (see Chapter 10). External functions are written in a host language, with only their signature (interface) appearing in the UDT definition. An external function definition can be declared as follows:

```
DECLARE EXTERNAL <FUNCTION_NAME> <SIGNATURE>
LANGUAGE <LANGUAGE_NAME>;
```

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.

12.2.5 Specifying Inheritance and Overloading of Functions

In SQL, inheritance can be applied to types or to tables; we will discuss the meaning of each in this section. Recall that we already discussed many of the principles of inheritance in Section 12.1.5. 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 12.4(a) and (b), where PERSON_TYPE, STUDENT_TYPE, and EMPLOYEE_TYPE are declared to be NOT FINAL). Associated with type inheritance are the rules for overloading of function implementations and for resolution of function names. These inheritance rules 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 12.4(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. These subtypes are shown in Figure 12.4(c).

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 12.4(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. Basically, table inheritance corresponds to the *extent inheritance* discussed in Section 12.1.5. 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.

12.2.6 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. An example is shown in Figure 12.4(e).

The keyword **SCOPE** specifies the name of the table whose tuples can be referenced by the reference attribute. Notice that 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.

12.3 The ODMG Object Model and the Object Definition Language ODL

As we discussed in the introduction to Chapter 6, 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. 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.

In this section, we describe the ODMG object model and the ODL. In Section 12.4, we discuss how to design an ODB from an EER conceptual schema. We will give an overview of OQL in Section 12.5, and the C++ language binding in Section 12.6. Examples of how to use ODL, OQL, and the C++ language binding will use the UNIVERSITY database example introduced in Chapter 4. In our description, we will follow the ODMG 3.0 object model as described in Cattell et al. (2000).[17] It is

[17]The earlier versions of the object model were published in 1993 and 1997.

important to note that many of the ideas embodied in the ODMG object model are based on two decades of research into conceptual modeling and object databases by many researchers.

The incorporation of object concepts into the SQL relational database standard, leading to object-relational technology, was presented in Section 12.2.

12.3.1 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. It also provides a standard terminology in a field where the same terms were sometimes used to describe different concepts. We will try to adhere to the ODMG terminology in this chapter. Many of the concepts in the ODMG model have already been discussed in Section 12.1, and we assume the reader has read this section. We will point out whenever the ODMG terminology differs from that used in Section 12.1.

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: identifier, name, lifetime, structure, and creation.

1. The **object identifier** is a unique system-wide identifier (or **Object_id**).[19] 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.[20] 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. 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.

[18]We will use the terms *value* and *state* interchangeably here.

[19]This corresponds to the OID of Section 12.1.2.

[20]This corresponds to the naming mechanism for persistence, described in Section 12.1.4.

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.[21] The term *atomic object* is different from how we defined the *atom constructor* in Section 12.1.3, which referred to all values of built-in data types. 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. We shall describe this in more detail later in this section.

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**[22] 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 described in Section 12.1.3. The built-in structured literals include Date, Interval, Time, and Timestamp (see Figure 12.5(b)). Additional user-defined structured literals can be defined as needed by each application.[23] User-defined structures are created using the **STRUCT** keyword in ODL, as in the C and C++ programming languages.

[21]In the ODMG model, *atomic objects* do not correspond to objects whose values are basic data types. All basic values (integers, reals, and so on) are considered *literals*.

[22]The use of the word *atomic* in *atomic literal* corresponds to the way we used atom constructor in Section 12.1.3.

[23]The structures for Date, Interval, Time, and Timestamp can be used to create either literal values or objects with identifiers.

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

(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 12.5
Overview of the interface definitions for part of the ODMG object model. (a) The basic Object interface, inherited by all objects, (b) Some standard interfaces for structured literals.

Figure 12.5 (continued) Overview of the interface definitions for part of the ODMG object model. (b) (continued) Some standard interfaces for structured literals, (c) Interfaces for collections and iterators.

```
        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);
        … };
```

(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);
```

```
    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 12.5 (continued)
Overview of the interface definitions for part of the ODMG object model.
(c) (continued) Interfaces for collections and iterators.

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.[24] 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 12.5 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. 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 behavior of an object type and is typically **noninstantiable** (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. 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 12.5 is a simplified version of the object model. For the full specifications, see Cattell et al. (2000). We will describe some of the constructs shown in Figure 12.5 as we describe the object model. In the object model, all objects inherit the basic interface operations of Object, shown in Figure 12.5(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).[25] 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().

[24]These are similar to the corresponding type constructors described in Section 12.1.3.

[25]Additional operations are defined on objects for *locking* purposes, which are not shown in Figure 12.5. We discuss locking concepts for databases in Chapter 22.

12.3.2 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.[26] 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. In Section 12.3.4 we will give examples of how these two inheritance relationships—":" and extends—may be used.

12.3.3 Built-in Interfaces and Classes in the Object Model

Figure 12.5 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 12.5(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 12.5(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*.

[26]The ODMG report also calls interface inheritance as type/subtype, is-a, and generalization/specialization relationships, although in the literature these terms have been used to describe inheritance of both state and operations (see Chapter 8 and Section 12.1).

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 12.5(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 (not shown in Figure 12.5(c)) 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 (not shown in Figure 12.5(c)) 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 12.5(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. A similar operation (not shown) is *O*.insert_element_last(*E*). The operation *O*.insert_element_after(*E*, *I*) in Figure 12.5(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 (not shown) 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*.

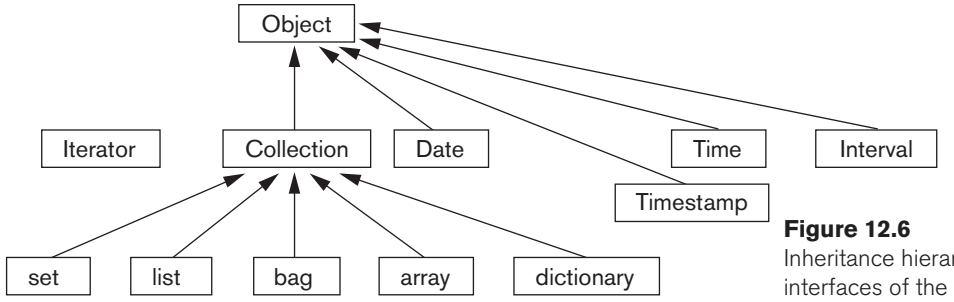


Figure 12.6
Inheritance hierarchy for the built-in interfaces of the object model.

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 12.6 is a diagram that 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>. Creating application classes is typically done by utilizing the object definition language ODL (see Section 12.3.6).

It is important to note that all objects in a particular collection *must be of the same type.* Hence, although the keyword any appears in the specifications of collection interfaces in Figure 12.5(c), this does not mean that objects of any type can be intermixed within the same collection. Rather, it means that any type can be used when specifying the type of elements for a particular collection (including other collection types!).

12.3.4 Atomic (User-Defined) Objects

The previous section described the built-in collection types of the object model. Now we discuss how object types for *atomic objects* can be constructed. These are

specified 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**.[27]

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**. In this subsection, we elaborate on the three types of components—attributes, relationships, and operations—that a user-defined object type for atomic (structured) objects can include. We illustrate our discussion with the two classes EMPLOYEE and DEPARTMENT shown in Figure 12.7.

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 12.7[28] 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* of Section 12.1.3. 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 (see Section 3.4) are explicitly represented, and each binary relationship is represented by a *pair of inverse references* specified via the keyword relationship. In Figure 12.7, 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.[29]

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

---

[27]As mentioned earlier, this definition of *atomic object* in the ODMG object model is different from the definition of atom constructor given in Section 12.1.3, which is the definition used in much of the object-oriented database literature.

[28]We are using the Object Definition Language (ODL) notation in Figure 12.7, which will be discussed in more detail in Section 12.3.6.

[29]Section 7.4 discusses how a relationship can be represented by two attributes in inverse directions.

```
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 12.7
The attributes, relationships, and operations in a class definition.

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 12.7 the EMPLOYEE class

has one operation: reassign_emp, and the DEPARTMENT class has two operations: add_emp and change_manager.

12.3.5 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 12.7 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 12.7 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[30] 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.

Next, we present the concept of **factory object**—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 12.8. 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 12.8 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 (not shown in Figure 12.8). As we can see, a factory object basically provides the **constructor operations** for new objects.

Finally, we discuss the concept of a **database**. Because an ODB system can create many different databases, each with its own schema, the ODMG object model has

[30]A composite key is called a *compound key* in the ODMG report.

```
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 12.8
Interfaces to illustrate factory objects and database objects.

interfaces for DatabaseFactory and Database objects, as shown in Figure 12.8. 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.

12.3.6 The Object Definition Language ODL

After our overview of the ODMG object model in the previous section, we now show how these concepts can be utilized to create an object database schema using the object definition language ODL.[31]

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. We will give an overview of the C++ binding in Section 12.6.

Figure 12.9(b) shows a possible object schema for part of the UNIVERSITY database, which was presented in Chapter 4. We will describe the concepts of ODL using this example, and the one in Figure 12.11. The graphical notation for Figure 12.9(b) is shown in Figure 12.9(a) and can be considered as a variation of EER diagrams (see Chapter 4) with the added concept of interface inheritance but without several EER concepts, such as categories (union types) and attributes of relationships.

Figure 12.10 shows one possible set of ODL class definitions for the UNIVERSITY database. In general, there may be several possible mappings from an object schema diagram (or EER schema diagram) into ODL classes. We will discuss these options further in Section 12.4.

Figure 12.10 shows the straightforward way of mapping part of the UNIVERSITY database from Chapter 4. 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 12.10 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

[31]The ODL syntax and data types are meant to be compatible with the Interface Definition language (IDL) of CORBA (Common Object Request Broker Architecture), with extensions for relationships and other database concepts.

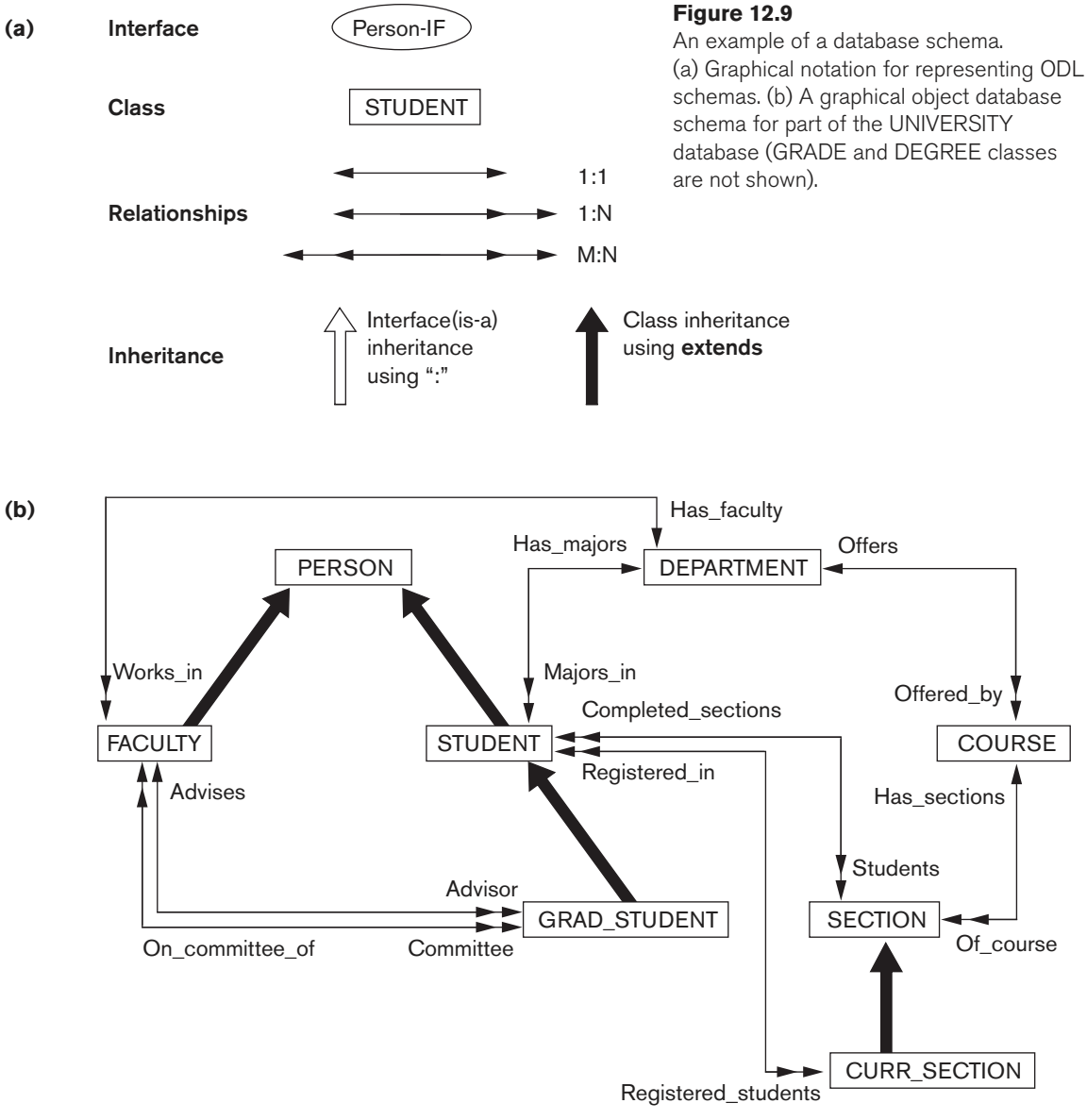


Figure 12.9
An example of a database schema. (a) Graphical notation for representing ODL schemas. (b) A graphical object database schema for part of the UNIVERSITY database (GRADE and DEGREE classes are not shown).

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 12.10 are straightforward mappings of the corresponding entity types in Figure 12.9(b).

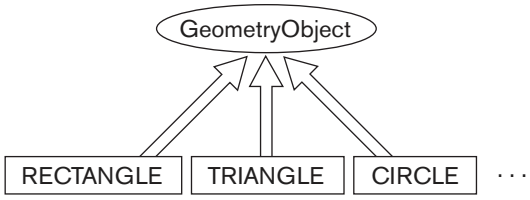
Figure 12.10
Possible ODL schema for the UNIVERSITY database in Figure 12.8(b).

```
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 12.10 (continued)
Possible ODL schema for the UNIVERSITY database in Figure 12.8(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); };
```

(a)



(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 12.11
An illustration of interface inheritance via ":". (a) Graphical schema representation, (b) Corresponding interface and class definitions in ODL.

However, the class GRADE requires some explanation. The GRADE class corresponds to the M:N relationship between STUDENT and SECTION in Figure 12.9(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.[32]

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

[32]We will discuss alternative mappings for attributes of relationships in Section 12.4.

GRADE.[33] These relationships are represented by the following relationship properties: Completed_sections of STUDENT; Section and Student of GRADE; and Students of SECTION (see Figure 12.10). Finally, the class DEGREE is used to represent the composite, multivalued attribute degrees of GRAD_STUDENT (see Figure 8.10).

Because the previous example does not include any interfaces, only classes, we now utilize a different example to illustrate interfaces and interface (behavior) inheritance. Figure 12.11(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 12.11(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).

12.4 Object Database Conceptual Design

Section 12.4.1 discusses how object database (ODB) design differs from relational database (RDB) design. Section 12.4.2 outlines a mapping algorithm that can be used to create an ODB schema, made of ODMG ODL class definitions, from a conceptual EER schema.

12.4.1 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. These can be considered as *OID references* to the related objects. Both single references and collections of references are allowed. References for a binary relationship can be

[33]This is similar to how an M:N relationship is mapped in the relational model (see Section 9.1) and in the legacy network model (see Appendix E).

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.

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 (table), as discussed in Section 9.1.

Mapping binary relationships that contain attributes is not straightforward in ODBs, since the designer must choose in which direction the attributes should be included. If the attributes are included in both directions, then redundancy in storage will exist and may lead to inconsistent data. Hence, it is sometimes preferable to use the relational approach of creating a separate table by creating a separate class to represent the relationship. This approach can also be used for n-ary relationships, with degree $n > 2$.

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, as we discussed in Section 9.2, there are several options to choose from since no built-in construct exists for inheritance in the basic relational model. It is important to note, though, that object-relational and extended-relational systems are adding features to model these constructs directly as well as to include operation specifications in abstract data types (see Section 12.2).

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.

There is a philosophical difference between the relational model and the object model of data in terms of behavioral specification. The relational model does *not* mandate the database designers to predefine a set of valid behaviors or operations, whereas this is a tacit requirement in the object model. One of the claimed advantages of the relational model is the support of ad hoc queries and transactions, whereas these are against the principle of encapsulation.

In practice, it is becoming commonplace to have database design teams apply object-based methodologies at early stages of conceptual design so that both the structure and the use or operations of the data are considered, and a complete specification is developed during conceptual design. These specifications are then mapped into relational schemas, constraints, and behavioral artifacts such as triggers or stored procedures (see Sections 5.2 and 13.4).

12.4.2 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 outline of 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.[34] *Multivalued attributes* are typically declared by using the set, bag, or list constructors.[35] 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).

Declare an extent for each class, and specify any key attributes as keys of the extent.

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.[36] 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.

Depending on the cardinality ratio of the binary relationship, 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[37]) for relationships in the 1:N or M:N direction. An alternative way to map binary M:N relationships is discussed in step 7.

If relationship attributes exist, a tuple constructor (struct) can be used to create a structure of the form <reference, relationship attributes>, which may be included instead of the reference attribute. However, this *does not allow the use of the inverse constraint*. Additionally, if this choice is represented in *both directions,* the attribute values will be represented twice, creating redundancy.

---

[34]This implicitly uses a tuple constructor at the top level of the type declaration, but in general, the tuple constructor is not explicitly shown in the ODL class declarations.

[35]Further analysis of the application domain is needed to decide which constructor to use because this information is not available from the EER schema.

[36]The ODL standard provides for the explicit definition of inverse relationships. Some ODBMS products may not provide this support; in such cases, programmers must maintain every relationship explicitly by coding the methods that update the objects appropriately.

[37]The decision whether to use set or list is not available from the EER schema and must be determined from the requirements.

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 specified, as discussed in steps 1, 2, and 3.

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<…>> or list<struct<…>> constructors. The attributes of the weak entity are included in the struct<…> construct, which corresponds to a tuple constructor. Attributes are mapped as discussed in steps 1 and 2.

Step 6. Categories (union types) in an EER schema are difficult to map to ODL. It is possible to create a mapping similar to the EER-to-relational mapping (see Section 9.2) by declaring a class to represent the category and defining 1:1 relationships between the category and each of its superclasses.

Step 7. An n-ary relationship with degree $n > 2$ can be mapped into a separate class, with appropriate references to each participating class. These references are based on mapping a 1:N relationship from each class that represents a participating entity type to the class that represents the n-ary relationship. An M:N binary relationship, especially if it contains relationship attributes, may also use this mapping option, if desired.

The mapping has been applied to a subset of the UNIVERSITY database schema in Figure 4.10 in the context of the ODMG object database standard. The mapped object schema using the ODL notation is shown in Figure 12.10.

12.5 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.

In Section 12.5.1 we will discuss the syntax of simple OQL queries and the concept of using named objects or extents as database entry points. Then, in Section 12.5.2 we will discuss the structure of query results and the use of path expressions to traverse relationships among objects. Other OQL features for handling object identity, inheritance, polymorphism, and other object-oriented concepts are discussed in Section 12.5.3. The examples to illustrate OQL queries are based on the UNIVERSITY database schema given in Figure 12.10.

12.5.1 Simple OQL Queries, Database Entry Points, and Iterator Variables

The basic OQL syntax is a **select … from … where …** structure, as it is for SQL. For example, the query to retrieve the names of all departments in the college of 'Engineering' can be written as follows:

```
Q0: select   D.Dname
    from     D in DEPARTMENTS
    where    D.College = 'Engineering';
```

In general, an **entry point** to the database is needed for each query, which can be any *named persistent object*. For many queries, the entry point is the name of the extent of a class. Recall that the extent name is considered to be the name of a persistent object whose type is a collection (in most cases, a set) of objects from the class. Looking at the extent names in Figure 12.10, the named object DEPARTMENTS is of type set<DEPARTMENT>; PERSONS is of type set<PERSON>; FACULTY is of type set<FACULTY>; and so on.

The use of an extent name—DEPARTMENTS in Q0—as an entry point refers to a persistent collection of objects. Whenever a collection is referenced in an OQL query, we should define an **iterator variable**[38]—*D* in Q0—that ranges over each object in the collection. In many cases, as in Q0, the query will select certain objects from the collection, based on the conditions specified in the where clause. In Q0, only persistent objects *D* in the collection of DEPARTMENTS that satisfy the condition *D*.College = 'Engineering' are selected for the query result. For each selected object *D*, the value of *D*.Dname is retrieved in the query result. Hence, the *type of the result* for Q0 is bag<string> because the type of each Dname value is string (even though the actual result is a set because Dname is a key attribute). In general, the result of a query would be of type bag for select … from … and of type set for select distinct … from … , as in SQL (adding the keyword distinct eliminates duplicates).

Using the example in Q0, there are three syntactic options for specifying iterator variables:

```
D in DEPARTMENTS
DEPARTMENTS D
DEPARTMENTS AS D
```

[38]This is similar to the tuple variables that range over tuples in SQL queries.

We will use the first construct in our examples.[39]

The named objects used as database entry points for OQL queries are not limited to the names of extents. Any named persistent object, whether it refers to an atomic (single) object or to a collection object, can be used as a database entry point.

12.5.2 Query Results and Path Expressions

In general, the result of a query can be of any type that can be expressed in the ODMG object model. A query does not have to follow the select … from … where … structure; in the simplest case, any persistent name on its own is a query, whose result is a reference to that persistent object. For example, the query

Q1: DEPARTMENTS;

returns a reference to the collection of all persistent DEPARTMENT objects, whose type is set<DEPARTMENT>. Similarly, suppose we had given (via the database bind operation, see Figure 12.8) a persistent name CS_DEPARTMENT to a single DEPARTMENT object (the Computer Science department); then, the query

Q1A: CS_DEPARTMENT;

returns a reference to that individual object of type DEPARTMENT. Once an entry point is specified, the concept of a **path expression** can be used to specify a *path* to related attributes and objects. A path expression typically starts at a *persistent object name,* or at the iterator variable that ranges over individual objects in a collection. This name will be followed by zero or more relationship names or attribute names connected using the *dot notation.* For example, referring to the UNIVERSITY database in Figure 12.10, the following are examples of path expressions, which are also valid queries in OQL:

Q2: CS_DEPARTMENT.Chair;
Q2A: CS_DEPARTMENT.Chair.Rank;
Q2B: CS_DEPARTMENT.Has_faculty;

The first expression Q2 returns an object of type FACULTY, because that is the type of the attribute Chair of the DEPARTMENT class. This will be a reference to the FACULTY object that is related to the DEPARTMENT object whose persistent name is CS_DEPARTMENT via the attribute Chair; that is, a reference to the FACULTY object who is chairperson of the Computer Science department. The second expression Q2A is similar, except that it returns the Rank of this FACULTY object (the Computer Science chair) rather than the object reference; hence, the type returned by Q2A is string, which is the data type for the Rank attribute of the FACULTY class.

Path expressions Q2 and Q2A return single values, because the attributes Chair (of DEPARTMENT) and Rank (of FACULTY) are both single-valued and they are applied to a single object. The third expression, Q2B, is different; it returns an object of type set<FACULTY> even when applied to a single object, because that is the type of the relationship Has_faculty of the DEPARTMENT class. The collection returned will include

[39]Note that the latter two options are similar to the syntax for specifying tuple variables in SQL queries.

a set of references to all FACULTY objects that are related to the DEPARTMENT object whose persistent name is CS_DEPARTMENT via the relationship Has_faculty; that is, a set of references to all FACULTY objects who are working in the Computer Science department. Now, to return the ranks of Computer Science faculty, we *cannot* write

```
Q3′:    CS_DEPARTMENT.Has_faculty.Rank;
```

because it is not clear whether the object returned would be of type set<string> or bag<string> (the latter being more likely, since multiple faculty may share the same rank). Because of this type of ambiguity problem, OQL does not allow expressions such as Q3′. Rather, one must use an iterator variable over any collections, as in Q3A or Q3B below:

```
Q3A:    select    F.Rank
        from      F in CS_DEPARTMENT.Has_faculty;

Q3B:    select    distinct F.Rank
        from      F in CS_DEPARTMENT.Has_faculty;
```

Here, Q3A returns bag<string> (duplicate rank values appear in the result), whereas Q3B returns set<string> (duplicates are eliminated via the distinct keyword). Both Q3A and Q3B illustrate how an iterator variable can be defined in the from clause to range over a restricted collection specified in the query. The variable *F* in Q3A and Q3B ranges over the elements of the collection CS_DEPARTMENT.Has_faculty, which is of type set<FACULTY>, and includes only those faculty who are members of the Computer Science department.

In general, an OQL query can return a result with a complex structure specified in the query itself by utilizing the struct keyword. Consider the following examples:

```
Q4:     CS_DEPARTMENT.Chair.Advises;

Q4A:    select struct ( name: struct (last_name: S.name.Lname, first_name:
                                      S.name.Fname),
                        degrees:( select struct (deg: D.Degree,
                                                 yr: D.Year,
                                                 college: D.College)
                                  from D in S.Degrees ))
        from S in CS_DEPARTMENT.Chair.Advises;
```

Here, Q4 is straightforward, returning an object of type set<GRAD_STUDENT> as its result; this is the collection of graduate students who are advised by the chair of the Computer Science department. Now, suppose that a query is needed to retrieve the last and first names of these graduate students, plus the list of previous degrees of each. This can be written as in Q4A, where the variable *S* ranges over the collection of graduate students advised by the chairperson, and the variable *D* ranges over the degrees of each such student *S*. The type of the result of Q4A is a collection of (first-level) structs where each struct has two components: name and degrees.[40]

[40]As mentioned earlier, struct corresponds to the tuple constructor discussed in Section 12.1.3.

The name component is a further struct made up of last_name and first_name, each being a single string. The degrees component is defined by an embedded query and is itself a collection of further (second level) structs, each with three string components: deg, yr, and college.

Note that OQL is *orthogonal* with respect to specifying path expressions. That is, attributes, relationships, and operation names (methods) can be used interchangeably within the path expressions, as long as the type system of OQL is not compromised. For example, one can write the following queries to retrieve the grade point average of all senior students majoring in Computer Science, with the result ordered by GPA, and within that by last and first name:

```
Q5A:  select struct ( last_name: S.name.Lname, first_name: S.name.Fname,
                      gpa: S.gpa )
      from          S in CS_DEPARTMENT.Has_majors
      where         S.Class = 'senior'
      order by      gpa desc, last_name asc, first_name asc;

Q5B:  select struct ( last_name: S.name.Lname, first_name: S.name.Fname,
                      gpa: S.gpa )
      from          S in STUDENTS
      where         S.Majors_in.Dname = 'Computer Science' and
                    S.Class = 'senior'
      order by      gpa desc, last_name asc, first_name asc;
```

Q5A used the named entry point CS_DEPARTMENT to directly locate the reference to the Computer Science department and then locate the students via the relationship Has_majors, whereas Q5B searches the STUDENTS extent to locate all students majoring in that department. Notice how attribute names, relationship names, and operation (method) names are all used interchangeably (in an orthogonal manner) in the path expressions: gpa is an operation; Majors_in and Has_majors are relationships; and Class, Name, Dname, Lname, and Fname are attributes. The implementation of the gpa operation computes the grade point average and returns its value as a float type for each selected STUDENT.

The order by clause is similar to the corresponding SQL construct, and specifies in which order the query result is to be displayed. Hence, the collection returned by a query with an order by clause is of type *list*.

12.5.3 Other Features of OQL

Specifying Views as Named Queries. The view mechanism in OQL uses the concept of a **named query**. The **define** keyword is used to specify an identifier of the named query, which must be a unique name among all named objects, class names, method names, and function names in the schema. If the identifier has the same name as an existing named query, then the new definition replaces the previous definition. Once defined, a query definition is persistent until it is redefined or deleted. A view can also have parameters (arguments) in its definition.

For example, the following view V1 defines a named query Has_minors to retrieve the set of objects for students minoring in a given department:

```
V1:  define  Has_minors(Dept_name) as
     select  S
     from    S in STUDENTS
     where   S.Minors_in.Dname = Dept_name;
```

Because the ODL schema in Figure 12.10 only provided a unidirectional Minors_in attribute for a STUDENT, we can use the above view to represent its inverse without having to explicitly define a relationship. This type of view can be used to represent inverse relationships that are not expected to be used frequently. The user can now utilize the above view to write queries such as

```
Has_minors('Computer Science');
```

which would return a bag of students minoring in the Computer Science department. Note that in Figure 12.10, we defined Has_majors as an explicit relationship, presumably because it is expected to be used more often.

Extracting Single Elements from Singleton Collections. An OQL query will, in general, return a collection as its result, such as a bag, set (if distinct is specified), or list (if the order by clause is used). If the user requires that a query only return a single element, there is an **element** operator in OQL that is guaranteed to return a single element *E* from a singleton collection *C* that contains only one element. If *C* contains more than one element or if *C* is empty, then the element operator *raises an exception.* For example, Q6 returns the single object reference to the Computer Science department:

```
Q6:  element ( select  D
               from    D in DEPARTMENTS
               where   D.Dname = 'Computer Science' );
```

Since a department name is unique across all departments, the result should be one department. The type of the result is *D*:DEPARTMENT.

Collection Operators (Aggregate Functions, Quantifiers). Because many query expressions specify collections as their result, a number of operators have been defined that are applied to such collections. These include aggregate operators as well as membership and quantification (universal and existential) over a collection.

The aggregate operators (min, max, count, sum, avg) operate over a collection.[41] The operator count returns an integer type. The remaining aggregate operators (min, max, sum, avg) return the same type as the type of the operand collection. Two examples follow. The query Q7 returns the number of students minoring in Computer Science and Q8 returns the average GPA of all seniors majoring in Computer Science.

[41]These correspond to aggregate functions in SQL.

```
Q7: count ( S in Has_minors('Computer Science'));

Q8: avg ( select   S.Gpa
          from     S in STUDENTS
          where    S.Majors_in.Dname = 'Computer Science' and
                   S.Class = 'Senior');
```

Notice that aggregate operations can be applied to any collection of the appropriate type and can be used in any part of a query. For example, the query to retrieve all department names that have more than 100 majors can be written as in Q9:

```
Q9: select   D.Dname
    from     D in DEPARTMENTS
    where    count (D.Has_majors) > 100;
```

The *membership* and *quantification* expressions return a Boolean type—that is, true or false. Let *V* be a variable, *C* a collection expression, *B* an expression of type Boolean (that is, a Boolean condition), and *E* an element of the type of elements in collection *C*. Then:

(*E* **in** *C*) returns true if element *E* is a member of collection *C*.
(**for all** *V* **in** *C* : *B*) returns true if *all* the elements of collection *C* satisfy *B*.
(**exists** *V* **in** *C* : *B*) returns true if there is at least one element in *C* satisfying *B*.

To illustrate the membership condition, suppose we want to retrieve the names of all students who completed the course called 'Database Systems I'. This can be written as in Q10, where the nested query returns the collection of course names that each STUDENT *S* has completed, and the membership condition returns true if 'Database Systems I' is in the collection for a particular STUDENT *S*:

```
Q10: select   S.name.Lname, S.name.Fname
     from     S in STUDENTS
     where    'Database Systems I' in
              ( select   C.Section.Of_course.Cname
                from     C in S.Completed_sections);
```

Q10 also illustrates a simpler way to specify the select clause of queries that return a collection of structs; the type returned by Q10 is bag<struct(string, string)>.

One can also write queries that return true/false results. As an example, let us assume that there is a named object called JEREMY of type STUDENT. Then, query Q11 answers the following question: *Is Jeremy a Computer Science minor?* Similarly, Q12 answers the question *Are all Computer Science graduate students advised by Computer Science faculty?* Both Q11 and Q12 return true or false, which are interpreted as yes or no answers to the above questions:

```
Q11: JEREMY in Has_minors('Computer Science');

Q12: for all G in
          ( select S
            from   S in GRAD_STUDENTS
            where S.Majors_in.Dname = 'Computer Science' )
          : G.Advisor in CS_DEPARTMENT.Has_faculty;
```

Note that query Q12 also illustrates how attribute, relationship, and operation inheritance applies to queries. Although *S* is an iterator that ranges over the extent GRAD_STUDENTS, we can write *S*.Majors_in because the Majors_in relationship is inherited by GRAD_STUDENT from STUDENT via extends (see Figure 12.10). Finally, to illustrate the **exists** quantifier, query Q13 answers the following question: *Does any graduate Computer Science major have a 4.0 GPA?* Here, again, the operation gpa is inherited by GRAD_STUDENT from STUDENT via extends.

```
Q13: exists   G in
     ( select S
       from   S in GRAD_STUDENTS
       where  S.Majors_in.Dname = 'Computer Science' )
     : G.Gpa = 4;
```

Ordered (Indexed) Collection Expressions. As we discussed in Section 12.3.3, collections that are lists and arrays have additional operations, such as retrieving the *i*th, first, and last elements. Additionally, operations exist for extracting a subcollection and concatenating two lists. Hence, query expressions that involve lists or arrays can invoke these operations. We will illustrate a few of these operations using sample queries. Q14 retrieves the last name of the faculty member who earns the highest salary:

```
Q14: first ( select    struct(facname: F.name.Lname, salary: F.Salary)
             from      F in FACULTY
             order by  salary desc );
```

Q14 illustrates the use of the **first** operator on a list collection that contains the salaries of faculty members sorted in descending order by salary. Thus, the first element in this sorted list contains the faculty member with the highest salary. This query assumes that only one faculty member earns the maximum salary. The next query, Q15, retrieves the top three Computer Science majors based on GPA.

```
Q15: ( select   struct( last_name: S.name.Lname, first_name: S.name.Fname,
                        gpa: S.Gpa )
       from     S in CS_DEPARTMENT.Has_majors
       order by gpa desc ) [0:2];
```

The select-from-order-by query returns a list of Computer Science students ordered by GPA in descending order. The first element of an ordered collection has an index position of 0, so the expression [0:2] returns a list containing the first, second, and third elements of the select … from … order by … result.

The Grouping Operator. The **group by** clause in OQL, although similar to the corresponding clause in SQL, provides explicit reference to the collection of objects within each *group* or *partition*. First we give an example, and then we describe the general form of these queries.

Q16 retrieves the number of majors in each department. In this query, the students are grouped into the same partition (group) if they have the same major; that is, the same value for *S*.Majors_in.Dname:

```
Q16: ( select     struct( dept_name, number_of_majors: count (partition) )
       from       S in STUDENTS
       group by   dept_name: S.Majors_in.Dname;
```

The result of the grouping specification is of type set<struct(dept_name: string, partition: bag<struct(*S*:STUDENT>)>), which contains a struct for each group (partition) that has two components: the grouping attribute value (dept_name) and the bag of the STUDENT objects in the group (partition). The select clause returns the grouping attribute (name of the department), and a count of the number of elements in each partition (that is, the number of students in each department), where **partition** is the keyword used to refer to each partition. The result type of the select clause is set<struct(dept_name: string, number_of_majors: integer)>. In general, the syntax for the group by clause is

group by F_1: E_1, F_2: E_2, … , F_k: E_k

where F_1: E_1, F_2: E_2, … , F_k: E_k is a list of partitioning (grouping) attributes and each partitioning attribute specification F_i: E_i defines an attribute (field) name F_i and an expression E_i. The result of applying the grouping (specified in the group by clause) is a set of structures:

set<struct(F_1: T_1, F_2: T_2, … , F_k: T_k, partition: bag)>

where T_i is the type returned by the expression E_i, partition is a distinguished field name (a keyword), and B is a structure whose fields are the iterator variables (S in Q16) declared in the from clause having the appropriate type.

Just as in SQL, a **having** clause can be used to filter the partitioned sets (that is, select only some of the groups based on group conditions). In Q17, the previous query is modified to illustrate the having clause (and also shows the simplified syntax for the select clause). Q17 retrieves for each department having more than 100 majors, the average GPA of its majors. The having clause in Q17 selects only those partitions (groups) that have more than 100 elements (that is, departments with more than 100 students).

```
Q17: select     dept_name, avg_gpa: avg ( select P.gpa from P in partition)
     from       S in STUDENTS
     group by   dept_name: S.Majors_in.Dname
     having     count (partition) > 100;
```

Note that the select clause of Q17 returns the average GPA of the students in the partition. The expression

select *P*.Gpa **from** *P* **in partition**

returns a bag of student GPAs for that partition. The from clause declares an iterator variable P over the partition collection, which is of type bag<struct(*S*: STUDENT)>. Then the path expression *P*.gpa is used to access the GPA of each student in the partition.

12.6 Overview of the C++ Language Binding in the ODMG Standard

The C++ language binding specifies how ODL constructs are mapped to C++ constructs. This is done via a C++ class library that provides classes and operations that implement the ODL constructs. An object manipulation language (OML) is needed to specify how database objects are retrieved and manipulated within a C++ program, and this is based on the C++ programming language syntax and semantics. In addition to the ODL/OML bindings, a set of constructs called *physical pragmas* are defined to allow the programmer some control over physical storage issues, such as clustering of objects, utilizing indexes, and memory management.

The class library added to C++ for the ODMG standard uses the prefix d_ for class declarations that deal with database concepts.[42] The goal is that the programmer should think that only one language is being used, not two separate languages. For the programmer to refer to database objects in a program, a class D_Ref<*T*> is defined for each database class *T* in the schema. Hence, program variables of type D_Ref<*T*> can refer to both persistent and transient objects of class *T*.

In order to utilize the various built-in types in the ODMG object model such as collection types, various template classes are specified in the library. For example, an abstract class D_Object<*T*> specifies the operations to be inherited by all objects. Similarly, an abstract class D_Collection<*T*> specifies the operations of collections. These classes are not instantiable, but only specify the operations that can be inherited by all objects and by collection objects, respectively. A template class is specified for each type of collection; these include D_Set<*T*>, D_List<*T*>, D_Bag<*T*>, D_Varray<*T*>, and D_Dictionary<*T*>, and they correspond to the collection types in the object model (see Section 12.3.1). Hence, the programmer can create classes of types such as D_Set<D_Ref<STUDENT>> whose instances would be sets of references to STUDENT objects, or D_Set<string> whose instances would be sets of strings. Additionally, a class d_Iterator corresponds to the Iterator class of the object model.

The C++ ODL allows a user to specify the classes of a database schema using the constructs of C++ as well as the constructs provided by the object database library. For specifying the data types of attributes,[43] basic types such as d_Short (short integer), d_Ushort (unsigned short integer), d_Long (long integer), and d_Float (floating-point number) are provided. In addition to the basic data types, several structured literal types are provided to correspond to the structured literal types of the ODMG object model. These include d_String, d_Interval, d_Date, d_Time, and d_Timestamp (see Figure 12.5(b)).

[42]Presumably, d_ stands for *database* classes.

[43]That is, *member variables* in object-oriented programming terminology.

To specify relationships, the keyword rel_ is used within the prefix of type names; for example, by writing

```
d_Rel_Ref<DEPARTMENT, Has_majors> Majors_in;
```

in the STUDENT class, and

```
d_Rel_Set<STUDENT, Majors_in> Has_majors;
```

in the DEPARTMENT class, we are declaring that Majors_in and Has_majors are relationship properties that are inverses of one another and hence represent a 1:N binary relationship between DEPARTMENT and STUDENT.

For the OML, the binding overloads the operation *new* so that it can be used to create either persistent or transient objects. To create persistent objects, one must provide the database name and the persistent name of the object. For example, by writing

```
D_Ref<STUDENT> S = new(DB1, 'John_Smith') STUDENT;
```

the programmer creates a named persistent object of type STUDENT in database DB1 with persistent name John_Smith. Another operation, delete_object() can be used to delete objects. Object modification is done by the operations (methods) defined in each class by the programmer.

The C++ binding also allows the creation of extents by using the library class d_Extent. For example, by writing

```
D_Extent<PERSON> ALL_PERSONS(DB1);
```

the programmer would create a named collection object ALL_PERSONS—whose type would be D_Set<PERSON>—in the database DB1 that would hold persistent objects of type PERSON. However, key constraints are not supported in the C++ binding, and any key checks must be programmed in the class methods.[44] Also, the C++ binding does not support persistence via reachability; the object must be statically declared to be persistent at the time it is created.

12.7 Summary

In this chapter, we started in Section 12.1 with an overview of the concepts utilized in object databases, and we discussed how these concepts were derived from general object-oriented principles. The main concepts we discussed were: object identity and identifiers; encapsulation of operations; inheritance; complex structure of objects through nesting of type constructors; and how objects are made persistent.

[44]We have only provided a brief overview of the C++ binding. For full details, see Cattell et al. (2000), Chapter 5.

Then, in Section 12.2, we showed how many of these concepts were incorporated into the relational model and the SQL standard; we showed that this incorporation leads to expanded relational database functionality. These systems have been called object-relational databases.

We then discussed the ODMG 3.0 standard for object databases. We started by describing the various constructs of the object model in Sction 12.3. The various built-in types, such as Object, Collection, Iterator, set, list, and so on, were described by their interfaces, which specify the built-in operations of each type. These built-in types are the foundation upon which the object definition language (ODL) and object query language (OQL) are based. We also described the difference between objects, which have an ObjectId, and literals, which are values with no OID. Users can declare classes for their application that inherit operations from the appropriate built-in interfaces. Two types of properties can be specified in a user-defined class—attributes and relationships—in addition to the operations that can be applied to objects of the class. The ODL allows users to specify both interfaces and classes, and permits two different types of inheritance—interface inheritance via ":" and class inheritance via **extends**. A class can have an extent and keys. A description of ODL followed, and an example database schema for the UNIVERSITY database was used to illustrate the ODL constructs.

Following the description of the ODMG object model, we described a general technique for designing object database schemas in Section 12.4. We discussed how object databases differ from relational databases in three main areas: references to represent relationships, inclusion of operations, and inheritance. Finally, we showed how to map a conceptual database design in the EER model to the constructs of object databases.

In Section 12.5, we presented an overview of the object query language (OQL). The OQL follows the concept of orthogonality in constructing queries, meaning that an operation can be applied to the result of another operation as long as the type of the result is of the correct input type for the operation. The OQL syntax follows many of the constructs of SQL but includes additional concepts such as path expressions, inheritance, methods, relationships, and collections. Examples of how to use OQL over the UNIVERSITY database were given.

Next we gave an overview of the C++ language binding in Section 12.6, which extends C++ class declarations with the ODL type constructors but permits seamless integration of C++ with the ODBMS.

In 1997 Sun endorsed the ODMG API (Application Program Interface). O2 technologies was the first corporation to deliver an ODMG-compliant DBMS. Many ODBMS vendors, including Object Design (now eXcelon), Gemstone Systems, POET Software, and Versant Corporation[45], have endorsed the ODMG standard.

[45]The Versant Object Technology product now belongs to Actian Corporation.

Review Questions

12.1. What are the origins of the object-oriented approach?

12.2. What primary characteristics should an OID possess?

12.3. Discuss the various type constructors. How are they used to create complex object structures?

12.4. Discuss the concept of encapsulation, and tell how it is used to create abstract data types.

12.5. Explain what the following terms mean in object-oriented database terminology: *method, signature, message, collection, extent*.

12.6. What is the relationship between a type and its subtype in a type hierarchy? What is the constraint that is enforced on extents corresponding to types in the type hierarchy?

12.7. What is the difference between persistent and transient objects? How is persistence handled in typical OO database systems?

12.8. How do regular inheritance, multiple inheritance, and selective inheritance differ?

12.9. Discuss the concept of polymorphism/operator overloading.

12.10. Discuss how each of the following features is realized in SQL 2008: *object identifier, type inheritance, encapsulation of operations, and complex object structures*.

12.11. In the traditional relational model, creating a table defined both the table type (schema or attributes) and the table itself (extension or set of current tuples). How can these two concepts be separated in SQL 2008?

12.12. Describe the rules of inheritance in SQL 2008.

12.13. What are the differences and similarities between objects and literals in the ODMG object model?

12.14. List the basic operations of the following built-in interfaces of the ODMG object model: Object, Collection, Iterator, Set, List, Bag, Array, and Dictionary.

12.15. Describe the built-in structured literals of the ODMG object model and the operations of each.

12.16. What are the differences and similarities of attribute and relationship properties of a user-defined (atomic) class?

12.17. What are the differences and similarities of class inhertance via **extends** and interface inheritance via ":" in the ODMG object model?

12.18. Discuss how persistence is specified in the ODMG object model in the C++ binding.

12.19. Why are the concepts of extents and keys important in database applications?

12.20. Describe the following OQL concepts: *database entry points, path expressions, iterator variables, named queries (views)*, *aggregate functions*, *grouping*, and *quantifiers*.

12.21. What is meant by the type orthogonality of OQL?

12.22. Discuss the general principles behind the C++ binding of the ODMG standard.

12.23. What are the main differences between designing a relational database and an object database?

12.24. Describe the steps of the algorithm for object database design by EER-to-OO mapping.

Exercises

12.25. Convert the example of GEOMETRY_OBJECTs given in Section 12.1.5 from the functional notation to the notation given in Figure 12.2 that distinguishes between attributes and operations. Use the keyword INHERIT to show that one class inherits from another class.

12.26. Compare inheritance in the EER model (see Chapter 4) to inheritance in the OO model described in Section 12.1.5.

12.27. Consider the UNIVERSITY EER schema in Figure 4.10. Think of what operations are needed for the entity types/classes in the schema. Do not consider constructor and destructor operations.

12.28. Consider the COMPANY ER schema in Figure 3.2. Think of what operations are needed for the entity types/classes in the schema. Do not consider constructor and destructor operations.

12.29. Design an OO schema for a database application that you are interested in. Construct an EER schema for the application, and then create the corresponding classes in ODL. Specify a number of methods for each class, and then specify queries in OQL for your database application.

12.30. Consider the AIRPORT database described in Exercise 4.21. Specify a number of operations/methods that you think should be applicable to that application. Specify the ODL classes and methods for the database.

12.31. Map the COMPANY ER schema in Figure 3.2 into ODL classes. Include appropriate methods for each class.

12.32. Specify in OQL the queries in the exercises of Chapters 6 and 7 that apply to the COMPANY database.

Selected Bibliography

Object-oriented database concepts are an amalgam of concepts from OO programming languages and from database systems and conceptual data models. A number of textbooks describe OO programming languages—for example, Stroustrup (1997) for C++, and Goldberg and Robson (1989) for Smalltalk. Books by Cattell (1994) and Lausen and Vossen (1997) describe OO database concepts. Other books on OO models include a detailed description of the experimental OODBMS developed at Microelectronic Computer Corporation called ORION and related OO topics by Kim and Lochovsky (1989). Bancilhon et al. (1992) describes the story of building the O2 OODBMS with a detailed discussion of design decisions and language implementation. Dogac et al. (1994) provides a thorough discussion on OO database topics by experts at a NATO workshop.

There is a vast bibliography on OO databases, so we can only provide a representative sample here. The October 1991 issue of *CACM* and the December 1990 issue of *ieee Computer* describe OO database concepts and systems. Dittrich (1986) and Zaniolo et al. (1986) survey the basic concepts of OO data models. An early paper on OO database system implementation is Baroody and DeWitt (1981). Su et al. (1988) presents an OO data model that was used in CAD/CAM applications. Gupta and Horowitz (1992) discusses OO applications to CAD, Network Management, and other areas. Mitschang (1989) extends the relational algebra to cover complex objects. Query languages and graphical user interfaces for OO are described in Gyssens et al. (1990), Kim (1989), Alashqur et al. (1989), Bertino et al. (1992), Agrawal et al. (1990), and Cruz (1992).

The Object-Oriented Manifesto by Atkinson et al. (1990) is an interesting article that reports on the position by a panel of experts regarding the mandatory and optional features of OO database management. Polymorphism in databases and OO programming languages is discussed in Osborn (1989), Atkinson and Buneman (1987), and Danforth and Tomlinson (1988). Object identity is discussed in Abiteboul and Kanellakis (1989). OO programming languages for databases are discussed in Kent (1991). Object constraints are discussed in Delcambre et al. (1991) and Elmasri, James, and Kouramajian (1993). Authorization and security in OO databases are examined in Rabitti et al. (1991) and Bertino (1992).

Cattell et al. (2000) describe the ODMG 3.0 standard, which is described in this chapter, and Cattell et al. (1993) and Cattell et al. (1997) describe the earlier versions of the standard. Bancilhon and Ferrari (1995) give a tutorial presentation of the important aspects of the ODMG standard. Several books describe the CORBA architecture—for example, Baker (1996).

The O2 system is described in Deux et al. (1991), and Bancilhon et al. (1992) includes a list of references to other publications describing various aspects of O2. The O2 model was formalized in Velez et al. (1989). The ObjectStore system

is described in Lamb et al. (1991). Fishman et al. (1987) and Wilkinson et al. (1990) discuss IRIS, an object-oriented DBMS developed at Hewlett-Packard Laboratories. Maier et al. (1986) and Butterworth et al. (1991) describe the design of GEMSTONE. The ODE system developed at AT&T Bell Labs is described in Agrawal and Gehani (1989). The ORION system developed at MCC is described in Kim et al. (1990). Morsi et al. (1992) describes an OO testbed.

Cattell (1991) surveys concepts from both relational and object databases and discusses several prototypes of object-based and extended relational database systems. Alagic (1997) points out discrepancies between the ODMG data model and its language bindings and proposes some solutions. Bertino and Guerrini (1998) propose an extension of the ODMG model for supporting composite objects. Alagic (1999) presents several data models belonging to the ODMG family.