Chapter II

Object-Oriented Features in Oracle™

In this chapter, we will describe Oracle™ features that can be used to support the implementation of an object-oriented model. As an overview, Section 2.1 will outline some of the original features within a standard relational model. The next sections will illustrate the additional object-oriented features. We will use these new features for our implementation in the subsequent chapters.

Relational-Model Features

In a relational model, the attributes are stored as columns of a table and the records are stored as rows of a table. As in most standard RDBMSs, Oracle™ provides a create-table statement following the SQL standard. After the declaration of the table name, we define the attributes' names and their data types. We can also perform the checking of attribute value. In the table, Oracle™ enables users to determine the uniqueness of the records by defining the primary key.

Oracle™ also enables the usage of a foreign key. The foreign-key attribute in a table refers to another record in another table. In addition to the foreign key,

Copyright © 2006, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

Figure 2.1. Create-table statement

```
General Syntax:

CREATE TABLE <table schema>
   (key attribute      NOT NULL,
    attribute          attribute type,
    attribute          attribute type
    [CHECK (<attribute value> IN (set of values))]
    PRIMARY KEY        (key attribute));

Example:

CREATE TABLE Employee
   (id          VARCHAR2(10)        NOT NULL,
    name        VARCHAR2(20),
    address     VARCHAR2(35),
    emp_type    VARCHAR2(8)
      CHECK(emp_type IN ('Manager', 'Worker')),
    PRIMARY KEY (id));
```

we can specify the referential integrity constraint every time we want to manipulate the target of a foreign-key reference. There are three types of constraint.

- **Restrict:** The manipulation operation is restricted to the case where there are no such matching attributes; it will be rejected, otherwise.
- **Cascade:** The manipulation operation, such as delete and update, cascades to the matching attributes.
- **Nullify or set null:** The manipulation operation is done after the foreign key is set to null.

Oracle™ performs the restrict integrity constraint as default. It prevents the update or deletion of a superclass key if there is a row in the subclass table that is referencing the key. However, Oracle™ provides only an on-delete integrity constraint. Therefore, to perform integrity constraint on other manipulations such as insert and update, we might need to use triggers.

Once we have created the table, we can perform the data manipulation. The manipulation can take form in the insertion, deletion, or update of data. The syntax of each of these is shown.

Copyright © 2006, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

Figure 2.2. Create table with referential integrity checking

```
General Syntax:

CREATE TABLE <table schema> OF <object schema>
   (key attribute      NOT NULL,
    attribute          attribute type,
    PRIMARY KEY        (key attribute),
    FOREIGN KEY        (key attribute)
    REFERENCES <referenced table schema>(key attribute)
    [ON DELETE] [CASCADE|SET NULL]);

Example:

CREATE TABLE Student
   (id          VARCHAR2(10) NOT NULL,
    course      VARCHAR2(10),
    year        VARCHAR2(4),
    PRIMARY KEY (id),
    FOREIGN KEY (id) REFERENCES Person ON DELETE CASCADE);
```

Figure 2.3. Data manipulation in Oracle™

```
General Syntax of Insertion:

INSERT INTO <table schema> [(attribute, ....,attribute)]
VALUES (attribute value, ....,attribute value);

Example:

INSERT INTO Student
VALUES ('1001', 'BEng', '2005');

General Syntax of Deletion:

DELETE FROM <table schema>
WHERE <statements>;

Example:

DELETE FROM Student
WHERE id = '1001';

General Syntax of Update:

UPDATE <table schema>
SET <statements>
WHERE <statements>;

Example:

UPDATE Student
SET year = '2005'
WHERE id = '1001';
```

Copyright © 2006, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

Object-Oriented Features

More recent commercial RDBMSs such as Oracle™ 8 and above (Loney & Koch, 2000, 2002; ORACLE™ 8, 1997) have extended their database systems with object-oriented features. In Oracle™ 8 and above, these features include enhancement of the existing data types with new data types including object types and user-defined types, and so forth.

Object Types and User-Defined Types

In Oracle™, a statement "create type" is used to create a new data type (object type) that can then be used as a generic type to create a table using the statement "create table," or to create another data type. The general syntax for these two create statements is shown in Figure 2.4. "As object" is used after creating an object type. Note that "or replace" is optional. By having this additional phrase, an object with the same name will automatically be replaced with the newest version of the object type. Figure 2.4 also shows an example of using object type Person_T as an attribute type in a new table, Course.

Figure 2.4. Oracle™ object type

```
General Syntax:

CREATE [OR REPLACE] TYPE <object schema> AS OBJECT
   (attribute  attribute type, ....,
    attribute  attribute type)
/

Example:

CREATE OR REPLACE TYPE Person_T AS OBJECT
   (person_id        VARCHAR2(10),
    person_name      VARCHAR2(30))
/

CREATE TABLE Course
   (course_id        VARCHAR2(10),
    course_name      VARCHAR2(20),
    lecturer         Person_T);
```

Copyright © 2006, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

Figure 2.5. Oracle™ varying array type

```
General Syntax:

CREATE [OR REPLACE] TYPE <object schema> AS VARRAY(n) OF (object/data
type)
/

Example:

CREATE OR REPLACE TYPE Persons AS VARRAY(3) OF Person_T
/

CREATE TABLE Course
   (course_id        VARCHAR2(10),
    course_name      VARCHAR2(20),
    lecturer         Persons);
```

Collection Types

Oracle™ allows the creation of an array type (*varray* or varying array). The syntax is basically using the same statement "create type" with the additional statement "as varray(*n*) of" followed by the object or the data type. Following Figure 2.4, it is possible to have more than one lecturer for a particular course, and therefore a new array of Persons can be defined.

Another extension is the support of nested tables, as shown in Figure 2.6. To create a table object, we use the same "create type" statement with the additional "as table of" statement following the name of the object table. This object table can then be used as a column in a table. When a table type appears as the type of a column in a table or as an attribute of the underlying object type, Oracle™ stores all of the nested table data in a single table, which is associated with the enclosing table or object type. Every time we create a table with columns or column attributes whose type is a nested table, we have to include the nested-table storage clause, "nested table (object table column schema) store as" followed by the separate storage-table name. Using the previous example from Figure 2.4, another data type called Person_Table_T can be created based on the Person_T data type to store the instances of a person. Note that Oracle™ 9 and above have also enabled users to create multilevel nested tables.

Copyright © 2006, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

Figure 2.6. Oracle™ nested table

```
General Syntax:

CREATE [OR REPLACE] TYPE <object table schema> AS TABLE OF (object
schema)
/
CREATE TABLE <table schema>
   (attribute          attribute type, ....,
    attribute          attribute type,
    nested item        object table schema);
   NESTED TABLE nested item STORE AS storage table schema;

CREATE TABLE <table schema>
   (attribute                attribute type, ....,
    outer nested item        object table schema);
   NESTED TABLE <outer nested item>
      STORE AS <outer storage table schema>
      (NESTED TABLE <inner nested item>
         STORE AS <inner storage table schema>);

Example:

CREATE OR REPLACE TYPE Person_T AS OBJECT
   (person_id          VARCHAR2(10),
    person_name        VARCHAR2(30))
   /

CREATE OR REPLACE TYPE Person_Table_T AS TABLE OF Person_T
/

CREATE TABLE Course
   (course_id          VARCHAR2(10),
    course_name        VARCHAR2(20),
    lecturer           Person_Table)
    NESTED TABLE lecturer STORE AS Person_tab;
```

Object Identifiers

In an object-oriented system, the OID is system generated and is used as a reference to locate the particular object. In Oracle™, the notion of an OID as a logical pointer is not supported; however, the concepts of an OID to uniquely identify a record (i.e., as a primary key) can be used. This is particularly useful in a deep inheritance hierarchy, where all subclasses have to carry the OID of the superclass in order to establish the connection between the superclass and its subclasses.

Copyright © 2006, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

Figure 2.7. Oracle™ object-identifiers implementation

```
General Syntax:

CREATE TABLE <table schema> OF <object schema>
   (key attribute      NOT NULL,
    attribute          attribute type,
    PRIMARY KEY        (key attribute),
   FOREIGN KEY         (key attribute)
   REFERENCES <referenced table schema>(key attribute);

Example:

CREATE OR REPLACE TYPE Person_T AS OBJECT
   (person_id          VARCHAR2(10),
    person_name        VARCHAR2(30))
   /

CREATE OR REPLACE TYPE Employee_T AS OBJECT
   (person_id          VARCHAR2(10),
    title              VARCHAR2(10),
    salary             NUMBER)
   /

CREATE TABLE Person OF Person_T
   (person_id  NOT NULL,
    PRIMARY KEY (person_id));

CREATE TABLE Employee OF Employee_T
   (person_id  NOT NULL,
    PRIMARY KEY (person_id),
FOREIGN KEY (person_id) REFERENCES Person(person_id));
```

Figure 2.7 illustrates the implementation of using OID to keep the inheritance between the superclass and its subclasses. Note that we can create a table from an object and determine the primary keys and foreign keys in this table. Every time we determine the foreign key, we have to use a "references" statement followed by the table and the column that is being referred. The general syntax for this primary key and foreign key implementation is shown in Figure 2.7. The table created is derived from an object type. Thus, we do not have to specify the attribute type anymore. They have to be identified while we create the object type. Note, however, that we can add a constraint "not null" statement to avoid a "null" value of an attribute. It is needed for particular attributes.

Copyright © 2006, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

Figure 2.8. Oracle™ relationship using object references

```
General Syntax:

CREATE TABLE <table schema>
   (object REF (object schema) [SCOPE IS (table schema)]);

Example:

CREATE OR REPLACE TYPE Person_T AS OBJECT
   (person_id        VARCHAR2(10),
    person_name      VARCHAR2(30))
   /

CREATE TABLE Academic_Staff OF Person_T;

CREATE TABLE Course
   (course_id        VARCHAR2(10),
    course_name      VARCHAR2(20),
    lecturer    REF Person_T SCOPE IS Academic_Staff);
```

Relationships using *Ref*

Oracle™ provides a way of referencing from one object to another by using the keyword *ref*. This object-referencing technique can be used to replace the standard "join" operations to traverse from one object to another.

We can then run a query:

```
SELECT C.course_name
FROM Course C
WHERE C.lecturer.person_name = 'Rahayu';
```

In the example above, the "scope is" statement is used to specify the exact table being referenced by the object. Whenever the scope parameter is used, the database engine will perform a join operation, which can be optimized using indexes. On the contrary, if the scope parameter is omitted and more than one table has been created using the given object type, the database engine will navigate through a set of object reference values in order to identify the location of the requested records (Dorsey & Hudicka, 1999).

Copyright © 2006, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

In the following chapters, we will not use the "scope is" parameter in our table-creation statement. In most situations, we will not build more than one table for each object type we declared, thereby avoiding the situation where the database engine has to navigate through a number of object references. When only one table is created for the object type, the ref operator will directly point to the associated reference.

Cluster

Oracle™ provides a clustering technique that can be very useful for an aggregation relationship. A cluster is created and will be defined in terms of all components that take part in the aggregation relationship, as is shown in Figure 2.9.

Figure 2.9. Oracle™ cluster

```
General Syntax:

CREATE CLUSTER <cluster schema>
   (cluster attribute            attribute type);

CREATE TABLE <table schema>
   (cluster attribute      attribute type,
    attribute              attribute type, ....,
    attribute              attribute type)
   CLUSTER <cluster schema> (cluster attribute);

CREATE INDEX <index schema> ON CLUSTER <cluster schema>;

Example:

CREATE CLUSTER HD_Cluster
   (hd_id            VARCHAR2(10));

CREATE TABLE Hard_Disk
   (hd_id            VARCHAR2(10) NOT NULL,
    capacity         VARCHAR2(20),
    PRIMARY KEY (hd_id))
   CLUSTER HD_Cluster(hd_id);

CREATE INDEX HD_Cluster_Index
   ON CLUSTER HD_Cluster;
```

Copyright © 2006, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

Inheritance Relationships using *Under*

Oracle™ 9 and above have a new feature that accommodates inheritance-relationship implementation. We do not have to use a primary-foreign-key relationship in order to simulate the relationship between a superclass and its subclasses.

To implement subtypes, we need to define the object as "not final" at the end of its type declaration. By default, without the keyword, the object type will be treated as final and no subtypes can be derived from the type. Oracle™ provides the keyword *under* to be used with the statement "create type" to create a subtype of a supertype such as shown in Figure 2.10.

Figure 2.10. Oracle™ "under" features

```
General Syntax:

CREATE [OR REPLACE] TYPE <super-type object schema> AS OBJECT
   (key attribute      attribute type,
    attribute          attribute type,...,
    attribute          attribute type) [FINAL|NOT FINAL]
/

CREATE [OR REPLACE] TYPE <sub-type object schema> UNDER <super-type
object schema>
   (additional attribute   attribute type, ....,
    additional attribute   attribute type)
    [FINAL|NOT FINAL]
/

CREATE TABLE <super-type table schema> OF
<super-type object schema>
   (key attribute      NOT NULL,
    PRIMARY KEY (key attribute));

Example:

CREATE OR REPLACE TYPE Person_T AS OBJECT
   (id          VARCHAR2(10),
    name        VARCHAR2(20),
    address     VARCHAR2(35)) NOT FINAL
/

CREATE OR REPLACE TYPE Student_T UNDER Person_T
   (course      VARCHAR2(10),
    year        VARCHAR2(4))
/
CREATE TABLE Person OF Person_T
   (id   NOT NULL,
    PRIMARY KEY (id);
```

Copyright © 2006, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

Encapsulation

Oracle™ provides two different types of encapsulation for an object-relational model. The first is through stored procedures or functions. The second is through member procedures or functions.

Stored Procedure or Function

The declaration of a stored procedure or function is basically very similar to the standard procedure declaration in many procedural languages. The encapsulation is provided by giving a grant to a specific role or user to access the particular stored procedure or function.

We need to use a "create procedure" statement. As in other create statements, the "or replace" statement is optional.

A stored procedure can have parameters attached to it, each of which must be followed by its type. We can also add the mode of the parameters between the parameter and the parameter type that is optional. There are three parameter modes (Oracle™, 1998).

- *In*. The value of the actual parameter is passed into the procedure when the procedure is invoked. Inside the procedure, the formal parameter is considered read only: It cannot be changed. Then the procedure finishes and control returns to the calling environment; the actual parameter is not changed.
- *Out*. Any value the actual parameter has when the procedure is called is ignored. Inside the procedure, the formal parameter is considered write only; it can only be assigned to and cannot be read from. When the procedure finishes and control returns to the calling environment, the contents of the formal parameter are assigned to the actual parameter.
- *In Out*. This mode is a combination of the two previous modes. The value of the actual parameter is passed into the procedure when the procedure is invoked. Inside the procedure, the formal parameter can be read from and written to. When the procedure finishes and control returns to the calling environment, the contents of the formal parameter are assigned to the actual parameter.

Copyright © 2006, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

Figure 2.11. Stored-procedures general syntax

```
General Syntax:

CREATE [OR REPLACE] PROCEDURE <procedure name>
    [parameter [{IN | OUT | IN OUT}] parameter type,
     ....,
     parameter [{IN | OUT | IN OUT}] parameter type)] AS

    [local variables]

BEGIN
   <procedure body>;
END <procedure name>;

GRANT EXECUTE ON <procedure_name> TO <user>;

Example:

CREATE OR REPLACE PROCEDURE Delete_Student(
   delete_id          Student.id%TYPE) AS

BEGIN
   DELETE FROM Student
   WHERE id = delete_id;
END Delete_Student;
/
GRANT EXECUTE ON Delete_Student TO Principal;
```

The stored procedure can have local variables in it. These are variables that are used only in the procedure body. Within the procedure body, we can use SQL statements such as select, insert, update, and delete. Thus, methods that are used to manipulate the database tables can be encapsulated within stored procedures. To run the procedure, we use the general syntax below.

```
General Syntax to Run the Stored Procedure:
EXECUTE procedure name [parameter,...,parameter];
EXECUTE Delete_Student['1001'];
```

Apart from stored procedures, stored functions are also available. Similar to stored procedures, stored functions can be likewise declared as in Figure 2.12. Note that for a function, we have to declare the type of the return value after

Copyright © 2006, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

Figure 2.12. Stored-functions general syntax

```
General Syntax:

CREATE [OR REPLACE] FUNCTION <function name>
   [parameter [{IN}] parameter type,
    ....,
    parameter [{IN}] parameter type)]
   RETURN datatype IS

   [local variables]

BEGIN
   <function body>;
   RETURN value;
END <function name>;

Example:

CREATE OR REPLACE FUNCTION Student_Course(
   s_id          Student.id%TYPE)
   RETURN VARCHAR2 IS

   v_course     VARCHAR(10);

BEGIN

   SELECT course INTO v_course
   FROM Student
   WHERE id = s_id;

   RETURN v_course;

END Student_Course;
/
```

we declare the function name. In addition, a stored function can take "in" parameters only.

Member Procedure or Function

Member procedures and member functions are physically implemented as PL or SQL procedures or functions, and they are defined together within the specification of the object type. Figure 2.13 demonstrates the general syntax.

Copyright © 2006, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

Unlike stored procedures, by using member methods we can identify the visibility scope of the methods. There are three types: public, private, and protected. By default, the attributes will be declared public.

Public attributes are visible from the class' interface (Fortier, 1999) and can be accessed by other types, tables, or routines. Private attributes are only visible from internal methods and will not be visible from outside the class specification. Finally, protected attributes are accessible from its own class or from any table or methods that use the class as a subtype.

The biggest advantage of methods over stored routines is the visibility gained by being part of the class. Methods will have access to attributes, procedures, and functions that may not be visible at the class interfaces (private or protected). On the other hand, a stored routine does not have access to these types of attributes, procedures, and functions.

Furthermore, the visibility of the methods inside a class can also be specified as private and protected. Same as attributes, the private methods can only be accessed by internal methods inside the particular class, and protected methods can be accessed only by its own user-defined types or any supertype interface of the particular class. We cannot apply this for stored routines.

Figure 2.14 shows an example of different visibility scopes of attributes and methods. All the attributes in Person are declared public and thus can be visible outside of the type interface. Some attributes in Staff, however, are declared private and protected. These attributes require additional internal methods for access, such as the function RetrieveTotalPayment to access the attributes StaffPayRate and StaffCommRate, and return the total payment. The function RetrieveStaffPhone in Person can be used to access the protected attribute in its subtype, StaffPhone. The procedure RetrievePersonDetail can be used to retrieve the attributes inside Person, including the private function RetrieveStaffPhone.

Finally, member methods have substitutability featured in the inheritance structure. Very often, when we insert data into a table, we wish to store different subtypes derived from a single or multiple supertypes. Using stored routines, we will require a different routine for a different parameter. With the substitutability feature, an instance of a subtype can be used in every context where an instance of a supertype can be used (Fortier, 1999). The context includes the use of different subtypes as parameters of the same function.

Copyright © 2006, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

Figure 2.13. Method implementation of member procedure

```
General Syntax:

CREATE [OR REPLACE] TYPE <object schema> AS OBJECT
   (attribute  attribute types,
    ....,
    attribute  attribute types,

    MEMBER PROCEDURE <procedure name>
       [(parameter [{IN | OUT | IN OUT}] parameter type,
       ....,
       parameter [{IN | OUT | IN OUT}] parameter type)],

    MEMBER FUNCTION <function name>
       [(parameter [{IN}] parameter type,
       ....,
       parameter [{IN}] parameter type)]
       RETURN datatype);
/

CREATE [OR REPLACE] TYPE BODY (object schema) AS

   MEMBER PROCEDURE <member procedure name>
      [parameter [{IN | OUT | IN OUT}] parameter type,
       ....,
       parameter [{IN | OUT | IN OUT}] parameter type)] IS

       [local variables]

   BEGIN
      <procedure body>;
   END <member procedure name>;

    MEMBER FUNCTION <function name>
       [parameter [{IN}] parameter type,
       ....,
       parameter [{IN}] parameter type)]
       RETURN datatype IS

       [local variables]

    BEGIN
       <procedure body>;
    END <member function name>;

END;
/
```

Copyright © 2006, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

Figure 2.13. (continued)

```
Example:

CREATE OR REPLACE TYPE Student_T AS OBJECT
  (id                    VARCHAR2(10),
   course                VARCHAR2(20),
   year                  VARCHAR2(4),

   MEMBER PROCEDURE
     Delete_Student )
/

CREATE OR REPLACE TYPE BODY Student_T AS

   MEMBER PROCEDURE
   Delete_Student IS

   BEGIN
        DELETE FROM Student
        WHERE Student.id = self.id;
   END Delete_Student;

END;
/
```

Figure 2.14. Visibility scope inside a class

```
CREATE TYPE Person
  (PersonID          VARCHAR(10),
   FirstName      VARCHAR(20),
   LastName           VARCHAR(20),
   Domicile           ADDRESS,
   BirthDate      DATE,

   PRIVATE FUNCTION RetrieveStaffPhone,
   PUBLIC PROCEDURE RetrievePersonDetail);

CREATE TYPE Staff UNDER PERSON
  (PUBLIC        StaffStartDate     DATE,
   PROTECTED     StaffPhone         CHAR(10),
   PRIVATE       StaffPayRate            DECIMAL(5,2),
   PRIVATE       StaffCommRate           DECIMAL(5,2),

   PUBLIC FUNCTION RetrieveTotalPayment)
```

Copyright © 2006, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

Figure 2.15. Member-methods substitutability

```
CREATE TYPE <object1 schema>                                  Original
(attr1 data type,...,                                          Method
 attrj data type,

 PROCEDURE <procedure1 name>(
 param1 parameter type data type,...,
 paramn parameter type data type);
) NOT FINAL/

CREATE TYPE <object2 schema> UNDER <object1 schema>
(attr1 data type,...,
 attrj data type,

 OVERRIDING PROCEDURE <procedure1 name>(
 param1 parameter type data type,...,                         Overriding
 paramn parameter type data type);                             Method
)/
```

Summary

Similar to many other DBMSs, Oracle™ was first targeted for RDBs. It has supported standard relational features in SQL including the data-definition language and the data-manipulation language. Due to the increased demand of a more powerful database, Oracle™ has added some object-oriented features into its DBMS. This chapter introduces some of them including the object type, collection type, inheritance, nested tables, and so forth. A list of references below provides more information on the syntax and definition of the features described in this chapter.

References

Dorsey, P., & Hudicka, J. (1999). *Oracle™ 8 design using UML object modelling* (chap. 1). Oracle Press, Osborne McGraw Hill.

Fortier, P. (1999). *SQL3 implementing the SQL foundation standard.* McGraw Hill.

Loney, K., & Koch, G. (2000). *Oracle™ 8i: The complete reference.* Osborne McGraw-Hill.

Copyright © 2006, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

Loney, K., & Koch, G. (2002). *Oracle™ 9i: The complete reference.* Oracle Press.

ORACLE™ 8. (1997). *Oracle™ 8 product documentation library.* Redwood City, CA: Oracle Corporation.

Urman, S. (2000). *Oracle™ 8i advanced PL/SQL programming.* Oracle Press, Osborne.

Chapter Problems

1. Using Oracle™, create a table to store book records. Each record has the title, the author, the publisher, and the ISBN (International Standard Book Number) that uniquely differentiate the book.
2. Continuing from Question 1, now we want to refer the attribute publisher into a table Publisher that has "name" as the primary key. If a deletion is performed in the publisher table, the associated referring key will be nullified. Alter your create-table statement from Question 1.
3. Write a statement in Oracle™ to implement an ordered collection type of the 20 most expensive book prices in a bookstore.
4. As in Question 1, you want to create a table Book. However, you want to instantiate the table from a specified Book_Type. Write the create-type and create-table statement.
5. *Movie Guide* magazine wants to keep a database of directors and the films that they directed. The director table has the attributes of name, age, and residence. The film is saved as an object with the attributes of title, genre, year, and rating. As a director may direct more than one film, the film object is implemented into the director table using a nesting technique. Show the implementation of the relationships described.
6. Discuss briefly the two mechanisms of encapsulation to implement methods or operations in an object-relational DBMS.

Copyright © 2006, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

Chapter Solutions

1. We use a create-table statement with the attributes of ISBN, title, author, and publisher. The primary key is the ISBN attribute.

```
CREATE TABLE Book
  (isbn        VARCHAR2(10)      NOT NULL,
   title       VARCHAR2(100),
   author      VARCHAR2(100),
   publisher   VARCHAR2(50),
   PRIMARY KEY (isbn));
```

2. We assume the table Publisher already exists.

```
CREATE TABLE Book
  (isbn        VARCHAR2(10)      NOT NULL,
   title       VARCHAR2(100),
   author      VARCHAR2(100),
   publisher   VARCHAR2(50),
   PRIMARY KEY (isbn),
   FOREIGN KEY (publisher) REFERENCES Publisher
              (Name) ON DELETE NULLIFY);
```

3. For an ordered collection with only one data element (in this case the price), we can use varray.

```
CREATE OR REPLACE TYPE prices AS VARRAY(20) OF
NUMBER(12,2)
/
```

4. First, create the type and then follow this by creating the table.

```
CREATE OR REPLACE TYPE Book_Type AS OBJECT
  (isbn        VARCHAR2(10),
   title       VARCHAR2(100),
   author      VARCHAR2(100),
   publisher   VARCHAR2(50))
   /
```

Copyright © 2006, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

```
CREATE TABLE Book OF Book_Type
  (isbn NOT NULL,
   PRIMARY KEY (isbn),
   FOREIGN KEY (publisher) REFERENCES Publisher(name)
   ON DELETE NULLIFY);
```

5. For a nested table, we have to create the object type followed by an object table before we can nest it to the table as an attribute.

```
CREATE OR REPLACE TYPE Film_T AS OBJECT
  (title      VARCHAR2(50),
   genre      VARCHAR2(10),
   year       NUMBER,
   rating     VARCHAR2(10))
/

CREATE OR REPLACE TYPE Film_Table_T AS TABLE OF
Film_T
/

CREATE OR REPLACE TYPE Director_T AS OBJECT
  (name        VARCHAR2(20),
   age         NUMBER,
   residence   VARCHAR2(20),
   filmography Film_Table_T)
/

CREATE TABLE Director OF Director_T
  (name NOT NULL,
   PRIMARY KEY (name))
  NESTED TABLE filmography STORE AS Film_tab;
```

6. The two mechanisms are encapsulation using stored procedures or functions with grants, and encapsulation using member procedures or functions.

   The first mechanism is based on pure RDBMS practice. It is a specific method for accessing the data that can be privileged to certain users by a grant mechanism. The second mechanism is based on an object-oriented model where the methods are encapsulated inside the class with the attributes. They are usually called member methods such as member procedures and member functions.

Copyright © 2006, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

Chapter III

Using Object-Oriented Features

In Chapter II, we discussed the different features available in Oracle™ that can be used to implement an object-oriented model. We will use those features in this chapter. The discussion in this chapter will be categorized based on the relationship types.

There are three distinct relationship types that we have to consider in object-oriented modeling for implementation in object-relational databases: inheritance, association, and aggregation. Some manipulations will be needed in order to accommodate the features of these relationships.

Using Inheritance Relationships

The concept of inheritance, where an object or a relation inherits the attribute (and methods) of another object, is not supported in the older versions of Oracle™ (prior to Oracle™ 9). The implementation of an inheritance relationship is established using primary-key and foreign-key relationships (shared ID) in order to simulate the relationship between a superclass and its subclasses.

Copyright © 2006, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

Generalización
Solapada y
Total

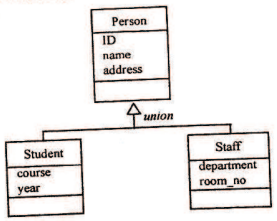
Union Inheritance Implementation

Figure 3.1 shows an inheritance relationship of union type. It declares that the union of a group of subclasses constitutes the entire membership of the superclass. In a union inheritance, we know that every object in the superclass is an object of at least one of the subclasses. In the example (see Figure 3.1), the union type does not preclude a member of a subclass from being a member of another subclass. For example, a person who is a staff member may also be a student at that university.

In order to simulate the union inheritance, Student and Staff will carry the primary key of the superclass, Person, in their relational tables. The primary key of the superclass becomes a foreign key in the subclasses. The foreign keys in the subclasses are also their primary keys. It becomes the main difference between the primary-key and foreign-key relationships in association and in inheritance. Thus, in Figure 3.1 it is noted that the primary key of Person is also the primary key of both Student and Staff. At the same time, the constraint of the primary-key and foreign-key relationship between the ID attributes in Student and Staff and the ID in Person is maintained in order to make sure that each student and staff is also a person. Thus, we have to specify the referential integrity constraint every time we want to manipulate the target of a foreign-key reference.

If we use the newer Oracle™ version, which supports inheritance using the "under" keyword, we can create Student and Staff subclasses under the superclass Person. The implementation is shown in Figure 3.3. Note that for union inheritance, we need to create one table each for the superclass and all the subclasses. As can be seen in the later sections, this union inheritance has a different way of implementation compared with other inheritance types. Using

Figure 3.1. Union inheritance



Copyright © 2006, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

Figure 3.2. Implementation of union inheritance

```
CREATE TABLE Person
   (id          VARCHAR2(10) NOT NULL,
    name        VARCHAR2(20),
    address     VARCHAR2(35),
    PRIMARY KEY (id));

CREATE TABLE Student
   (id          VARCHAR2(10) NOT NULL,
    course      VARCHAR2(10),
    year        VARCHAR2(4),
    PRIMARY KEY (id),
    FOREIGN KEY (id) REFERENCES Person ON DELETE CASCADE);

CREATE TABLE Staff
   (id          VARCHAR2(10) NOT NULL,
    department  VARCHAR2(10),
    room_no     VARCHAR2(4),
    PRIMARY KEY (id),
    FOREIGN KEY (id) REFERENCES Person ON DELETE CASCADE);
```

Figure 3.3. Implementation of union inheritance using "under"

```
CREATE OR REPLACE TYPE Person_T AS OBJECT
   (id          VARCHAR2(10),
    name        VARCHAR2(20),
    address     VARCHAR2(35)) NOT FINAL
/

CREATE TABLE Person OF Person_T
   (id          NOT NULL,
   PRIMARY KEY (id));

CREATE OR REPLACE TYPE Student_T UNDER Person_T
   (course      VARCHAR2(10),
    year        VARCHAR2(4))
/

CREATE TABLE Student OF Student_T
   (id          NOT NULL,
   PRIMARY KEY (id));

CREATE OR REPLACE TYPE Staff_T UNDER Person_T
   (department  VARCHAR2(10),
    room_no     VARCHAR2(4))
   /

CREATE TABLE Staff OF Staff_T
   (id          NOT NULL,
    PRIMARY KEY (id));
```

Copyright © 2006, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

the "under" keyword, normally we do not need to create separate tables for the subclasses because the table created for the superclass can also be used to store the instances of the subclasses. However, in this union type of inheritance, we need to allow a particular person to be both a student as well as a staff. If we are to store all instances into one superclass table, we will not be able to store the two records together as they will violate the primary-key constraints (i.e., two records with the same ID). Therefore, we need to create a separate table for each of the subclasses to allow the same person's record to appear in both the Student as well as Staff tables. We also need to create a table for Person to store persons who are neither staff members nor students.

pero si era total no hacia tal Cap 50

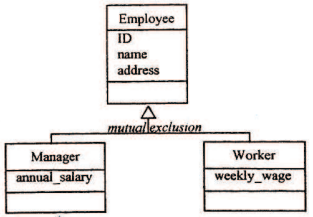
Generalización Disjunta

Mutual-Exclusion Inheritance Implementation

Mutual-exclusion inheritance declares that a group of subclasses in an inheritance relationship is pairwise disjointed. An example of this type is shown in Figure 3.4. This example is called mutual exclusion because there is no manager who is also a worker, and vice versa. However, in this case there may be an employee who is neither a manager nor a worker.

The best way to handle mutual-exclusion inheritance without losing the semantics of the relationship is by adding to the superclass table an attribute that reflects the type of the subclasses or has the value null. For example (see Figure 3.4), in the table Employee, an attribute called emp_type is added. Thus, emp_type can take the values manager, worker, or null. There are no employees that can have two values for this attribute, such as a manager who is also a worker simultaneously (mutual exclusion). Figure 3.5 shows the implementation details. Note that we use the "check" keyword for the purpose of checking the value of an attribute in a set of values.

Figure 3.4. Mutual-exclusion inheritance



Copyright © 2006, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

Figure 3.5. Implementation of mutual-exclusion inheritance

```
CREATE TABLE Employee
   (id          VARCHAR2(10) NOT NULL,
    name        VARCHAR2(20),
    address     VARCHAR2(35),
    emp_type    VARCHAR2(8)
      CHECK(emp_type IN ('Manager', 'Worker', NULL)),
    PRIMARY KEY (id));

CREATE TABLE Manager
   (id                VARCHAR2(10) NOT NULL,
    annual_salary     NUMBER,
    PRIMARY KEY (id),
    FOREIGN KEY (id) REFERENCES Employee (id)
    ON DELETE CASCADE);

CREATE TABLE Worker
   (id                VARCHAR2(10) NOT NULL,
    weekly_wage       NUMBER,
    PRIMARY KEY (id),
    FOREIGN KEY (id) REFERENCES Employee (id)
    ON DELETE CASCADE);
```

Figure 3.6. Implementation of mutual-exclusion inheritance using "under"

```
CREATE OR REPLACE TYPE Employee_T AS OBJECT
   (id          VARCHAR2(10),
    name        VARCHAR2(20),
    address     VARCHAR2(35),
    emp_type    VARCHAR2(8)) NOT FINAL
   /

CREATE TABLE Employee OF Employee_T
   (id   NOT NULL,
    emp_type CHECK (emp_type in ('Manager', 'Worker', 'NULL')),
    PRIMARY KEY (id));

CREATE OR REPLACE TYPE Manager_T UNDER Employee_T
   (annual_salary    NUMBER)
   /

CREATE OR REPLACE TYPE Worker_T UNDER Employee_T
   (weekly_wage      NUMBER)
   /
```

Using the newer Oracle™ version for the same example, we can create Manager and Worker subclasses under the superclass Employee (see Figure 3.6). Notice that in this type of inheritance, we create only one table for the superclass. We do not need subclass tables because an object can be a member

Copyright © 2006, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

of only one subclass. These subclasses are instantiations of the superclass. Also notice that although the table is created from the superclass table, Oracle™ maintains the integrity constraint between the subclass and the superclass table. We cannot delete the subclass while the superclass table still exists.

In this case, an employee can only be a manager, a worker, or neither. If an employee is neither a manager nor a worker, he or she is only an object of the superclass, Employee. If an employee is a manager, for example, he or she will be an object of the subclass Manager. Thus, the employee will have all of the attributes of the Manager type and all other attributes that are inherited from the Employee type. However, all of the subclass tables can be kept in the superclass table. ¿y cómo mete los datos en los subtipos, si no ha creado la clase?

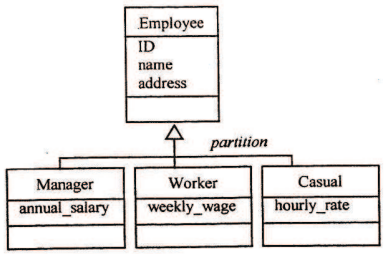
Partition Inheritance Implementation

Total y disjunta

Partition inheritance declares that a group of subclasses partitions a superclass. A partition requires that the partitioning sets be pairwise disjointed and that their union constitute the partitioned set. Therefore, a partition type can be said to be a combination of both union and mutual-exclusion types. Figure 3.7 shows an example of a partition type of inheritance. We use the example of an employee again, but here a new class, Casual, is added, and it is assumed that each member of the Employee class must belong to one and only one of the classes Manager, Worker, and Casual. For example, an employee cannot be both a manager and a casual.

Similar to the other types of inheritance, the best way to map the partition type of inheritance into tables is to have one table for each superclass and one for

Figure 3.7. Partition inheritance



Copyright © 2006, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

each subclass. Like the mutual-exclusion type, a new attribute emp_type is added to the superclass table. The difference is that this new attribute has a constraint, which is "not null." This will ensure that each superclass object belongs to a particular subclass type. It also ensures that no superclass object belongs to more than one subclass. Figure 3.8 shows an example of the implementation of partition inheritance. Notice that the attribute emp_type is also needed in the Employee table with the "not null" constraint.

The newer Oracle™ version can also accommodate this inheritance type. It is very similar to the implementation in the mutual-exclusion type. The only difference is the constraint of emp_type in the Employee table as is shown in Figure 3.9.

Multiple Inheritance Implementation

The last type of inheritance relationship is called multiple inheritance. Figure 3.10 gives an example of multiple inheritance. A Tutor class can be said to be inheriting from overlapping classes because basically a tutor can be a student who is also a staff member.

The best way to handle this inheritance from overlapping classes is to use one table for each superclass and one table for the subclass. Figure 3.11 gives an example of a multiple inheritance implementation. A Tutor class can be said to be inheriting from overlapping classes Student and Staff.

Figure 3.8. Implementation of partition inheritance

```
CREATE TABLE Employee
   (id          VARCHAR2(10) NOT NULL,
    name        VARCHAR2(20),
    address     VARCHAR2(35),
    emp_type    VARCHAR2(8) NOT NULL
      CHECK(emp_type IN ('Manager', 'Worker' ,'Casual')),
    PRIMARY KEY (id));

CREATE TABLE Manager same as in mutual exclusive inheritance
CREATE TABLE Worker  same as in mutual exclusive inheritance

CREATE TABLE Casual
   (id                VARCHAR2(10) NOT NULL,
    hourly_rate       NUMBER,
    PRIMARY KEY (id),
    FOREIGN KEY (id) REFERENCES Employee (id) ON DELETE CASCADE);
```

Copyright © 2006, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

Figure 3.9. Implementation of partition inheritance relationship using "under"

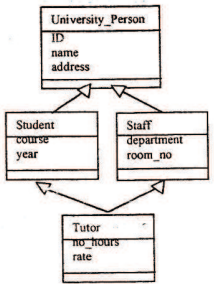
```
CREATE OR REPLACE TYPE Employee_T AS OBJECT
   (id          VARCHAR2(10),
    name        VARCHAR2(20),
    address     VARCHAR2(35),
    emp_type    VARCHAR2(8)) NOT FINAL
   /

CREATE TABLE Employee OF Employee_T
   (id NOT NULL,
    emp_type NOT NULL
    CHECK (emp_type in ('Manager', 'Worker', 'Casual')),
    PRIMARY KEY (id));

CREATE TYPE Manager_T       same as in mutual exclusive inheritance
CREATE TYPE Worker_T        same as in mutual exclusive inheritance

CREATE OR REPLACE TYPE Casual_T UNDER Employee_T
   (hourly_rate       NUMBER)
/
```

Figure 3.10. Multiple inheritance



At the time of this writing, the newer Oracle™ does not support multiple inheritance using the "under" keyword. This keyword is applicable only to the single inheritance type. However, this multiple inheritance concept is often simulated using other existing techniques. For example, we can use the "under" keyword to implement one inherited parent, and use an association type to link to the other parent. The drawback of using this technique is that only the parent type implemented using "under" can be inherited, and therefore we have to be careful when choosing which parent to inherit and which one to associate.

Copyright © 2006, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

Figure 3.11. Implementation of multiple inheritance relationship

```
CREATE TABLE Person
   (id          VARCHAR2(10) NOT NULL,
    name        VARCHAR2(20),
    address     VARCHAR2(35),
    PRIMARY KEY (id));

CREATE TABLE Student
   (id          VARCHAR2(10) NOT NULL,
    course      VARCHAR2(10),
    year        VARCHAR2(4),
    PRIMARY KEY (id),
    FOREIGN KEY (id) REFERENCES Person (id) ON DELETE CASCADE);

CREATE TABLE Staff
   (id          VARCHAR2(10) NOT NULL,
    department  VARCHAR2(10),
    room_no     VARCHAR2(4),
    PRIMARY KEY (id),
    FOREIGN KEY (id) REFERENCES Person (id) ON DELETE CASCADE);

CREATE TABLE Tutor
   (id          VARCHAR2(10) NOT NULL,
    no_hours    NUMBER,
    rate        NUMBER,
    PRIMARY KEY (id)
    FOREIGN KEY (id) REFERENCES Person (id) ON DELETE CASCADE);
```

Using Association Relationships

Relational data structures can be related to the concepts of sets through the fact that tuples are not in any particular order and duplicate tuples are not allowed. Therefore, the implementation of association relationships with a set semantic into object-relational tables is identical to the well-known transformation of many-to-many or one-to-many relationships from relational modeling to relational tables.

In relational modeling, many-to-many relationships are converted into tables in which the primary key is a composite key obtained from the participating entities. Should there be any attributes of the relationships, these will automatically be added to the tables that represent the many-to-many relationships. Likewise, in object modeling, if a class has a set relationship with another class and the inverse relationship is also a set, the transformation of such an

Copyright © 2006, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

association is identical to the many-to-many relationships' transformation from relational modeling to relational tables where a table is created to represent the set relationship. This transformation strategy also enforces that each element within a set cannot be duplicated, which is realized by the implementation of the composite primary key of the relationship tables.

In one-to-many relationships, as in relational modeling, the primary key of the one side is copied to the many side to become a foreign key. In other words, there is no special treatment necessary for the transformation of association relationships having a set semantic.

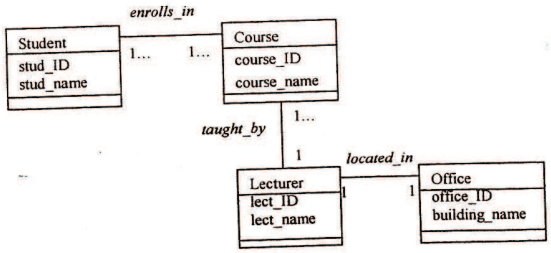
In Oracle™, there are two ways of implementing an association relationship: by primary-key and foreign-key relationships and by object references. Each of these methods will be described as follows.

Creating an Association Relationship by a Primary-Key and Foreign-Key Relationship

This first method is the traditional relational implementation of connecting two or more tables together. The placement of the foreign keys is based on the cardinality of the association relationship, whether it is one to one, one to many, or many to many. We will use the following object-oriented diagram to show the implementation of association relationships.

The first association between Student and Course is a many-to-many relationship. A third table needs to be created to keep the relationship between the two

Figure 3.12. Object-oriented diagram for association relationships



Copyright © 2006, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

Figure 3.13. Implementation of many to many using a primary-key and foreign-key relationship

```
CREATE TABLE Course
   (course_id        VARCHAR2(10) NOT NULL,
    course_name      VARCHAR2(20),
    PRIMARY KEY (course_id));

CREATE TABLE Student
   (stud_id     VARCHAR2(10) NOT NULL,
    stud_name   VARCHAR2(20),
    PRIMARY KEY (stud_id));

CREATE TABLE Enrolls_in
   (course_id  VARCHAR2(10) NOT NULL,
    stud_id    VARCHAR2(10) NOT NULL,
    PRIMARY KEY (course_id, stud_id),
    FOREIGN KEY (course_id) REFERENCES Course (course_id)
    ON DELETE CASCADE,
    FOREIGN KEY (stud_id) REFERENCES Student (stud_id)
    ON DELETE CASCADE);
```

Figure 3.14. Implementation of one to many using a primary-key and foreign-key relationship

```
CREATE TABLE Lecturer
   (lect_id     VARCHAR2(10) NOT NULL,
    lect_name   VARCHAR2(20),
    PRIMARY KEY (lect_id));

CREATE TABLE Course
   (course_id        VARCHAR2(10) NOT NULL,
    course_name      VARCHAR2(20),
    lect_id          VARCHAR(10),
    PRIMARY KEY (course_id),
    FOREIGN KEY (lect_id) REFERENCES Lecturer (lect_id)
    ON DELETE CASCADE);
```

connected tables. This table will have the primary keys of the connected tables as its primary (composite) key. Each of the primary keys, which form the composite, is connected to the originated table through a primary-key and foreign-key relationship.

The second association is a one-to-many relationship between Lecturer and Course. In order to establish the association relationship in the implementation, the primary key of the one side, Lecture, becomes a foreign key of the table that holds the many side, Course.

Copyright © 2006, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

Figure 3.15. Implementation of one-to-one using a primary-key and foreign-key relationship

```
CREATE TABLE Office
   (office_id        VARCHAR2(10) NOT NULL,
    building_name    VARCHAR2(20),
    PRIMARY KEY (office_id));

CREATE TABLE Lecturer
   (lect_id     VARCHAR2(10) NOT NULL,
    lect_name   VARCHAR2(20),
    office_id   VARCHAR2(10),
    PRIMARY KEY (lect_id),
    FOREIGN KEY (office_id) REFERENCES Office (office_id)
    ON DELETE CASCADE);
```

The third association is a one-to-one relationship between Lecturer and Office. In this type of relationship, one has to decide the *participation constraint* between the two connected tables (Elmasri & Navathe, 2000). There are two types of participation constraints, namely, *total* and *partial*. In the above example, every lecturer must be located in one particular office; thus, the participation of the lecturer in the relationship is total. On the other hand, one particular office may be vacant; no particular lecturer has been assigned the room. In this case, the participation of the office in the relationship is partial. In order to establish the association relationship in the implementation, the primary key of the table with partial participation, Office, becomes a foreign key of the table that holds the total participation, Lecturer.

Creating an Association Relationship by Object References

Another implementation method of association relationships in Oracle™ is using object references. Instead of connecting two tables through the values of the associated primary key and foreign key, this method allows one to directly connect two tables through the referencing attribute. Thus, the associated attribute that connects the two tables is not holding a value of the primary key of the other connected table, but a reference of where the connected table is actually stored.

Copyright © 2006, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

Figure 3.16. Implementation of many to many using object references

```
CREATE OR REPLACE TYPE Person_T AS OBJECT
   (person_id        VARCHAR2(10),
    person_name      VARCHAR2(30))
   /

CREATE OR REPLACE TYPE Course_T AS OBJECT
   (course_id        VARCHAR2(10),
    course_name      VARCHAR2(30))
   /

CREATE TABLE Student OF Person_T
   (person_id NOT NULL,
    PRIMARY KEY (person_id));

CREATE TABLE Course OF Course_T
   (course_id NOT NULL,
    PRIMARY KEY (course_id));

CREATE TABLE Enrolls_in
   (student     REF Person_T,
   course       REF Course_T);
```

Figure 3.17. Implementation of one to many using object references

```
CREATE OR REPLACE TYPE Person_T AS OBJECT
   (person_id        VARCHAR2(10),
    person_name      VARCHAR2(30))
   /

CREATE OR REPLACE TYPE Course_T AS OBJECT
   (course_id        VARCHAR2(10),
    course_name      VARCHAR2(30),
    course_lecturer  REF Person_T)
   /

CREATE TABLE Lecturer OF Person_T
   (person_id NOT NULL,
    PRIMARY KEY (person_id));

CREATE TABLE Course OF Course_T
   (course_id NOT NULL,
    PRIMARY KEY (course_id));
```

Copyright © 2006, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

Figure 3.18. Implementation of one to one using object references

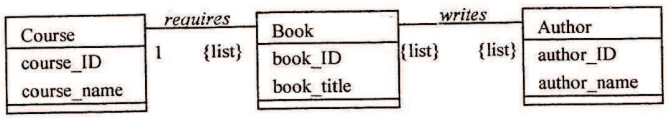
```
CREATE OR REPLACE TYPE Office_T AS OBJECT
   (office_id          VARCHAR2(10),
    building_name      VARCHAR2(20))
   /

CREATE OR REPLACE TYPE Person_T AS OBJECT
   (person_id          VARCHAR2(10),
    person_name        VARCHAR2(30),
    person_office      REF Office_T)
   /

CREATE TABLE Office OF Office_T
   (office_id NOT NULL,
    PRIMARY KEY (office_id));

CREATE TABLE Lecturer OF Person_T
   (person_id NOT NULL,
   PRIMARY KEY (person_id));
```

Figure 3.19. Association example using collection types



The following figures show the implementation of many-to-many, one-to-many, and one-to-one relationships of the example in Figure 3.17 using object references.

In some cases, we want to have additional semantics at the many side, for example, by incorporating an ordering semantic. To show more implementation examples of association relationships involving collection types, we will extend the example in Figure 3.12. The additional classes are shown in Figure 3.19. Every course will require a list of books as references. The class Book is also associated with a list of authors. Note that in this example we use the term *list* to represent an ordered collection as opposed to the earlier example of *set* for an unordered collection.

Copyright © 2006, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

Figure 3.20. Implementation of one to list using object references

```
CREATE OR REPLACE TYPE Course_T AS OBJECT
   (course_id          VARCHAR2(10),
    course_name        VARCHAR2(30))
/

CREATE OR REPLACE TYPE Book_T AS OBJECT
   (book_id            VARCHAR2(10),
    book_title         VARCHAR2(30),
    course_book        REF Course_T)
/

CREATE TABLE Course OF Course_T
   (course_id NOT NULL,
    PRIMARY KEY (course_id));

CREATE TABLE Book OF Book_T
   (book_id NOT NULL,
    PRIMARY KEY (book_id));

CREATE TABLE Require
   (Book               REF    Book_T,
    Index_Book         NUMBER NOT NULL,
    Course             REF    Course_T);
```

The following figures show the implementation of one-to-list and list-to-list relationships of this example using object references. Note that we have the attribute Index_Book in table Require because we need the ordering semantic of the book associated with a specific course.

The main difference between a list implementation and the earlier many-to-many association is the need to add one index attribute (e.g., Index_Author in Figure 3.21). This index will maintain the ordering semantic within the list.

Primary Keys: Foreign Keys vs. Object References in an Association Relationship

An association relationship uses keys to provide a solid referential integrity constraint. As mentioned earlier, we can add constraints (cascade, restrict, and nullify) by either using the Oracle™ system-defined constraints or by triggers. With the referential integrity constraints, there will be an automatic check on the table that is being referenced before data manipulation is performed. On the

Copyright © 2006, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

re 3.21. Implementation of list to list using object references

```
CREATE OR REPLACE TYPE Book_T AS OBJECT
   (book_id          VARCHAR2(10),
    book_title       VARCHAR2(30))
/

CREATE OR REPLACE TYPE Author_T AS OBJECT
   (author_id        VARCHAR2(10),
    author_name      VARCHAR2(30))
/

CREATE TABLE Book OF Book_T
   (book_id NOT NULL,
    PRIMARY KEY (book_id));

CREATE TABLE Author OF Author_T
   (author_id NOT NULL,
    PRIMARY KEY (author_id));

CREATE TABLE Write
   (Book             REF Book_T,
    Index_Book       NUMBER NOT NULL,
    Author           REF Author_T);

CREATE TABLE Written_By
   (Author           REF Author_T,
    Index_Author     NUMBER NOT NULL,
    Book             REF Book_T);
```

- hand, using the object reference ref, there is no referential integrity traint performed. There is the possibility for an object reference to be *ling* if the object it refers to has been accidentally deleted.

suggestion to avoid this is by applying a foreign key to the object-reference ept. For example, recalling Figure 3.18, we can create a new version to eferential integrity into the object reference (see Figure 3.22).

re 3.22. Implementation of one to one using ref and references

```
CREATE TYPE Office_T -- same as in Figure 3.18
CREATE TYPE Person_T -- same as in Figure 3.18
CREATE TABLE Office -- same as in Figure 3.18

CREATE TABLE Lecturer OF Person_T
   (person_id  NOT NULL,
    PRIMARY KEY (person_id),
    FOREIGN KEY (person_office) REFERENCES Office
    ON DELETE CASCADE);
```

ght © 2006, Idea Group Inc. Copying or distributing in print or electronic forms without written sion of Idea Group Inc. is prohibited.

Using Aggregation Relationships

There are two techniques that can be used in Oracle™ in order to simulate the implementation of aggregations: the clustering technique and the nesting technique.

Implementing Existence-Dependent Aggregation using the Clustering Technique

In this section we use an example of a homogeneous aggregation relationship between Hard Disk (HD) and HD Controller (HD_Contr; see Figure 1.13). The Oracle™ implementation of this type of aggregation using the clustering technique is shown in Figure 3.23.

It is clear from the implementation that the clustering technique supports only an existence- dependent aggregation. It is not possible to have an HD controller (part object) that does not belong to an HD (whole object). This is enforced by the existence of the cluster key in all the part tables. Moreover, the example in Figure 3.23 also shows a nonexclusive aggregation type, where each part

Figure 3.23. Implementation of existence-dependent aggregation using the clustering technique

```
CREATE CLUSTER HD_Cluster
   (hd_id            VARCHAR2(10));

CREATE TABLE Hard_Disk
   (hd_id            VARCHAR2(10) NOT NULL,
    capacity         VARCHAR2(20),
    PRIMARY KEY (hd_id))
   CLUSTER HD_Cluster(hd_id);

CREATE TABLE HD_Contr
   (hd_id            VARCHAR2(10) NOT NULL,
    hd_contr_id      VARCHAR2(10) NOT NULL,
    description      VARCHAR2(25),
    PRIMARY KEY (hd_id, hd_contr_id),
    FOREIGN KEY (hd_id) REFERENCES Hard_Disk (hd_id))
   CLUSTER HD_Cluster(hd_id);

CREATE INDEX HD_Cluster_Index
ON CLUSTER HD_Cluster;
```

Copyright © 2006, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

Figure 3.24. Implementation of exclusive aggregation using the clustering technique

```
CREATE TABLE Hard_Disk
   (hd_id              VARCHAR2(10) NOT NULL,
    capacity           VARCHAR2(20),
    PRIMARY KEY (hd_id))
   CLUSTER HD_Cluster(hd_id);

CREATE TABLE HD_Contr
   (hd_id              VARCHAR2(10) NOT NULL,
    hd_contr_id        VARCHAR2(10) NOT NULL,
    description        VARCHAR2(25),
    PRIMARY KEY (hd_contr_id),
    FOREIGN KEY (hd_id) REFERENCES Hard_Disk (hd_id))
   CLUSTER HD_Cluster(hd_id);
```

object can be owned by more than one whole object. For example, HD controller HDC1 may belong to HD1 as well as HD2.

Depending on the situation, the above nonexclusive type may not be desirable. We can enforce the aggregation-exclusive type by creating a single primary key for the part object and treating the cluster key as a foreign key rather than as part of the primary key. Figure 3.24 shows the implementation of the previous example as an exclusive type (the implementation of the cluster and the cluster index remain the same).

Each time a new record is inserted into the part table, HD_Contr, the value of the cluster key, hd_id, is searched for. If it is found, the new record will be added to the cluster. The rows of the whole table, Hard_Disk, and the rows of the part table, HD_Contr, are actually stored together physically (see Figure 3.25). The index is created in order to enhance the performance of the cluster storage.

Figure 3.25. Physical storage of the aggregation relationship using cluster

hd_id	capacity	hd_contr_id	description
HD11	2GB	Contr111	.....
		Contr112	.....
HD12	6GB	Contr121	.....
		Contr122	.....
		Contr123	.....

Copyright © 2006, Idea Group Inc. Copying or distributing in print or electronic forms without written

Figure 3.26. Implementation of an aggregation relationship with multiple part objects

```
CREATE CLUSTER PC_Cluster
   (pc_id      VARCHAR2(10));

CREATE TABLE PC
   (pc_id      VARCHAR2(10) NOT NULL,
    type       VARCHAR2(20),
    PRIMARY KEY (pc_id))
   CLUSTER PC_Cluster(pc_id);

CREATE TABLE Hard_Disk
   (pc_id      VARCHAR2(10) NOT NULL,
    hd_id      VARCHAR2(10) NOT NULL,
    capacity   VARCHAR2(20),
    PRIMARY KEY (pc_id, hd_id),
    FOREIGN KEY (pc_id) REFERENCES PC (pc_id))
   CLUSTER PC_Cluster(pc_id);

CREATE TABLE Monitor
   (pc_id      VARCHAR2(10) NOT NULL,
    monitor_id VARCHAR2(10) NOT NULL,
    resolution VARCHAR2(25),
    PRIMARY KEY (pc_id, monitor_id),
    FOREIGN KEY (pc_id) REFERENCES PC (pc_id))
   CLUSTER PC_Cluster(pc_id);

CREATE TABLE Keyboard
   (PC_id              VARCHAR2(10) NOT NULL,
    keyboard_id        VARCHAR2(10) NOT NULL,
    type               VARCHAR2(25),
    PRIMARY KEY (pc_id, keyboard_id),
    FOREIGN KEY (pc_id) REFERENCES PC (pc_id))
   CLUSTER PC_Cluster(pc_id);

CREATE TABLE CPU
   (pc_id      VARCHAR2(10) NOT NULL,
    cpu_id     VARCHAR2(10) NOT NULL,
    speed      VARCHAR2(10),
    PRIMARY KEY (pc_id, cpu_id),
    FOREIGN KEY (pc_id) REFERENCES PC (pc_id))
   CLUSTER PC_Cluster(pc_id);

CREATE INDEX PC_Cluster_Index
   ON CLUSTER PC_Cluster;
```

It is also possible to use the cluster method to implement an aggregation relationship between a whole object with a number of part objects. Figure 3.26 demonstrates the implementation of an aggregation between a PC with Hard_Disk, Monitor, Keyboard, and CPU (see Figure 1.8 in Chapter I).

Copyright © 2006, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

Figure 3.27. Physical storage of multiple-aggregation relationships using cluster

whole id	whole attribute	part ID	part attribute
PC001	.....	HardDisk1	.....
		HardDisk1	.....
		Monitor11	.....
		Keyboard	.....
		CPU11	.....
PC002	.....	HardDisk2	.....
		Monitor21	
		Keyboard	
		CPU21	.....

Figure 3.27 shows the physical storage of the multiple aggregation relationship between a PC with Hard_Disk, Monitor, Keyboard, and CPU.

Implementing Existence-Dependent Aggregation using the Nesting Technique

Another Oracle™ implementation technique for aggregation involves using nested tables. In this technique, similar to the clustering one, the part information is tightly coupled with the information of the whole object and it is implemented as a nested table. This actually enforces the aggregation existence-dependent type. If the data of the whole object is removed, all associated part objects will need to be removed as well. Moreover, the data in the part nested table is normally accessed through the whole object only. Because of this, this nested-table technique is suitable only for the implementation of the aggregation existence-dependent type.

Figure 3.28 describes the link between the whole and the part table in a nesting structure, whereas Figure 3.29 shows the implementation of the homogenous aggregation depicted in Figure 1.13 using the nested-table technique.

Note that there is neither the concept of a primary key nor the integrity constraint in the part nested table as shown in Figure 3.28. For example, if a particular HD controller is used by another HD from the whole table, then all the details of the HD controller will be written again as a separate record within the nested table.

Copyright © 2006, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

Figure 3.28. Aggregation relationships using a nested table

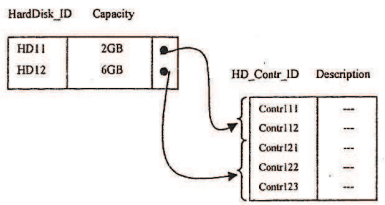


Figure 3.29. Implementation of aggregation relationships using nested tables

```
CREATE OR REPLACE TYPE HD_Contr AS OBJECT
   (hd_contr_id       VARCHAR2(10),
    description       VARCHAR2(30));
/

CREATE OR REPLACE TYPE HD_Contr_Table AS TABLE OF HD_Contr
/

CREATE TABLE Hard_Disk
   (hd_id       VARCHAR2(10) NOT NULL,
    capacity    VARCHAR2(20),
    controller HD_Contr_Table,
    PRIMARY KEY (hd_id))
  NESTED TABLE controller STORE AS HD_Contr_tab;
```

Oracle™ also facilitates multilevel nested tables and thus can be used for implementing a multilevel aggregation relationship. It is implemented by using the inner and outer table principle (see Figure 3.30). A PC is an aggregation of several HDs, and a HD is an aggregation of several HD controllers. In this case, the inner table is a nested table of HD controller, and the outer table is a nested table of HD. The implementation of this aggregation is shown in Figure 3.31.

Note in the implementation (see Figure 3.29 and Figure 3.31) that we do not create standard tables for the HD controller. We only need to define a HD controller type, and define it as a nested table later when we create the Hard Disk table (for Figure 3.29) and the PC table (for Figure 3.31). It is also shown that the information of the nested table is stored externally in a table called HD_Contr_tab. This is not a standard table; no additional constraints can be

Copyright © 2006, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

Figure 3.30. Multilevel aggregation relationships using nested tables

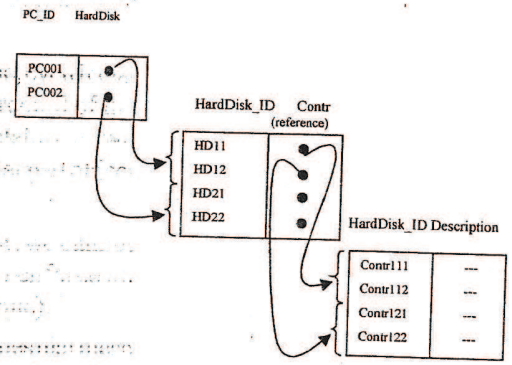


Figure 3.31. Implementation of multilevel aggregation relationships using nested tables

```
Example:

CREATE OR REPLACE TYPE HD_Contr AS OBJECT
   (hd_contr_id        VARCHAR2(10),
    description        VARCHAR2(30))
   /

CREATE OR REPLACE TYPE HD_Contr_Table AS TABLE OF HD_Contr
/

CREATE OR REPLACE TYPE Hard_Disk AS OBJECT
   (hd_id        VARCHAR2(10),
    capacity     VARCHAR2(20),
    controller HD_Contr_Table)
   /

CREATE OR REPLACE TYPE Hard_Disk_Table AS TABLE OF Hard_Disk
/

CREATE TABLE PC
   (pc_id        VARCHAR2(10) NOT NULL,
    hd           Hard_Disk_Table,
    PRIMARY KEY (pc_id))
   NESTED TABLE hd STORE AS HD_tab
      (NESTED TABLE controller STORE AS HD_Contr_tab);
```

Copyright © 2006, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

attached to this table and no direct access can be performed to this table without going through the Hard Disk table.

Every whole object can own any part object in the nesting technique, even if that particular part has been owned by another whole object. The record of the HD_Contr object will simply be repeated every time a hard disk claims to own it. This shows a nonexclusive type of aggregation, where a particular part object can be shared by more than one whole object.

Because there is no standard table created for the HD controller, we cannot have a primary key for the table, which we usually employ to enforce an exclusive type of aggregation (see the previous clustering technique).

It is clear from the above sections on clustering and nesting techniques that these techniques are suitable only for the implementation of the existence-dependent type of aggregation. The clustering technique supports both nonexclusive and exclusive aggregation. However, the nesting technique supports only the nonexclusive type.

In the following section we will see how we can implement an existence-independent type of aggregation.

Implementing Existence-Independent Aggregation

To implement the existence-independent aggregation type in relational tables, an Aggregate table is created. This table maintains the part-of relationship between the whole table and the part tables. By having one Aggregate table, we avoid having a link from the whole to the part that is hard coded within one of the tables. In both the clustering and nesting techniques, the connection between whole and part is either hard coded within the whole table (in the nesting technique) or within the part tables (in the clustering technique). These techniques actually prevent us from creating independent part objects that exist but are not necessarily connected to a particular whole at any given time.

In the Aggregate table, only the relationships between the identifiers of the whole table and the part tables are stored. To maintain consistency in the Aggregate table, the identifiers across different part tables should be kept unique. If the number of the part tables is more than one, a new attribute type is used to distinguish the different types of the part tables.

Figure 3.32 shows an existence-independent aggregation, where lab is an aggregate of Computer, Printer, and Scanner. There are times when we have

Copyright © 2006, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

Figure 3.32. Existence-independent type of aggregation

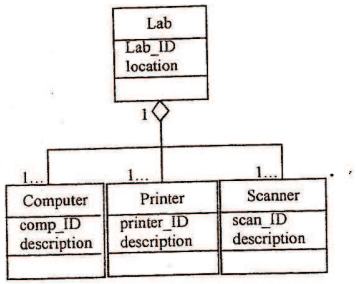
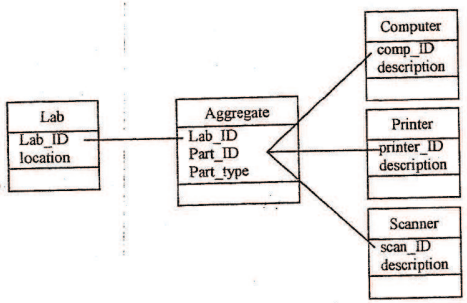


Figure 3.33. Existence-independent type of aggregation using the Aggregate table



new computers or printers that have not been allocated to any particular lab. We want to still be able to keep the record of the new parts even when no associated whole is established.

This situation cannot be implemented using either the clustering or the nested technique. In the nesting technique, we can only insert a new part record within the nested table if we have an existing whole record for it. In the clustering technique, the primary key of the whole serves as the cluster key; thus, it is not supposed to be null.

Figure 3.33 shows how an Aggregate table is created to store the relationship between Lab and Computer, Printer, and Scanner. The Aggregate table contains the primary key of the whole, which is lab_ID, and an attribute called part_ID, which is the primary key of either one of the part tables (comp_ID, printer_ID, or scan_ID). The last attribute is called part_type, which is the type

Copyright © 2006, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

Figure 3.34. Implementation of the existence-independent type of aggregation

```
CREATE TABLE Lab
   (lab_id            VARCHAR2(10) NOT NULL,
    location          VARCHAR2(20),
    PRIMARY KEY (lab_id));

CREATE TABLE Computer
   (comp_id           VARCHAR2(10) NOT NULL,
    description       VARCHAR2(10),
    PRIMARY KEY (comp_id));

CREATE TABLE Printer
   (printer_id        VARCHAR2(10) NOT NULL,
    description       VARCHAR2(10),
    PRIMARY KEY (printer_id));

CREATE TABLE Scanner
   (scan_id           VARCHAR2(10) NOT NULL,
    description       VARCHAR2(10),
    PRIMARY KEY (scan_id));

CREATE TABLE Aggregate
   (lab_id            VARCHAR2(10) NOT NULL,
    part_id           VARCHAR2(10),
    part_type         VARCHAR2(20)
      CHECK (part_type in ('Computer', 'Printer', 'Scanner')),
    PRIMARY KEY (lab_id, part_id),
    FOREIGN KEY (lab_id) REFERENCES Lab(lab_id));
```

of the part_ID (computer, printer, or scanner). Figure 3.34 demonstrates the implementation of the above aggregation structure.

Figure 3.3 shows an implementation of the existence–independent, nonexclusive aggregation type. It is an existence-independent type because the new records of part tables, Computer, Printer, and Scanner, can be inserted without any associated record within the whole table, Lab. If a Lab record is removed from the Lab table, it will only be cascaded to the Aggregate table where the specific Lab record appears; however, it does not have to affect the records within the associated part tables. The above example is also a nonexclusive type because due particular part, such as a printer, can appear in the Aggregate table more than once and is associated with a different lab_ID. This is possible because both the lab_ID and part_ID are primary keys of the Aggregate table. If this situation is not desirable, then we can make the lab_ID a foreign key in

Copyright © 2006, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

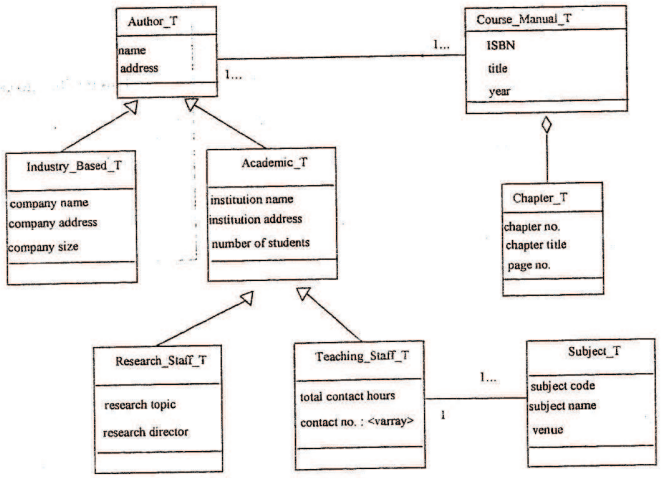
the Aggregate table, and only the part_ID will be the primary key. This will enforce each part_ID to appear only once within the Aggregate table and be associated with one particular lab_ID only.

Case Study

The following course-manual authorship case study shows how we can implement an object-oriented model in Oracle™. The diagram shows two inheritance relationships. First is the union inheritance between an author and an industry-based author and an academic author. Second is the mutual-exclusion inheritance between an academic and a research staff and a teaching staff. There are association relationships between the author and course manual, as well as between the teaching staff and subject. There is also one aggregation relationship between the course manual and its chapters.

To implement the course-manual authorship object-oriented model into Oracle™, we will apply the following systematic steps: type and table. We have

Figure 3.35. Course-manual authorship case study



Copyright © 2006, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

Figure 3.36. Implementation of the case study in Oracle™

```
CREATE OR REPLACE TYPE Author_T AS OBJECT
  (ao_id          VARCHAR2(3),
   name           VARCHAR2(10),
   address        VARCHAR2(20)) NOT FINAL
  /

CREATE OR REPLACE TYPE Industry_Based_T UNDER Author_T
  (c_name         VARCHAR2(10),
   c_address      VARCHAR2(20),
   c_size         VARCHAR2(10))
  /

CREATE OR REPLACE TYPE Academic_T UNDER Author_T
  (i_name         VARCHAR2(10),
   i_address      VARCHAR2(20),
   no_student     NUMBER,
   academic_type  VARCHAR2(20)) NOT FINAL
  /

CREATE OR REPLACE TYPE Research_Staff_T UNDER Academic_T
  (topic          VARCHAR2(20),
   director       VARCHAR2(10))
  /

CREATE OR REPLACE TYPE Contacts AS VARRAY(3) OF NUMBER
/

CREATE OR REPLACE TYPE Teaching_Staff_T UNDER Academic_T
  (total_hour     NUMBER,
   contact_no     Contacts)
  /

CREATE TABLE Author OF Author_T
   (ao_id NOT NULL,
    PRIMARY KEY (ao_id));

-- implementation of inheritance using an earlier version of Oracle
-- or traditional relational databases

CREATE OR REPLACE TYPE Author_T AS OBJECT
  (ao_id          VARCHAR2(3),
   name           VARCHAR2(10),
   address        VARCHAR2(20))
  /
```

Copyright © 2006, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

Figure 3.36. (continued)

```
CREATE OR REPLACE TYPE Industry_Based_T AS OBJECT
  (ao_id          VARCHAR2(3),
   c_name         VARCHAR2(10),
   c_address            VARCHAR2(20),
   c_size         VARCHAR2(10))
  /
CREATE OR REPLACE TYPE Academic_T AS OBJECT
  (ao_id          VARCHAR2(3),
   i_name         VARCHAR2(10),
   i_address            VARCHAR2(20),
   no_student     NUMBER,
   academic_type  VARCHAR2(20))
  /
CREATE OR REPLACE TYPE Research_Staff_T AS OBJECT
  (ao_id          VARCHAR2(3),
   topic          VARCHAR2(20),
   director       VARCHAR2(10))
  /
CREATE OR REPLACE TYPE Contacts AS VARRAY(3) OF NUMBER
  /
CREATE OR REPLACE TYPE Teaching_Staff_T AS OBJECT
  (ao_id          VARCHAR2(3),
   total_hour     NUMBER,
   contact_no     Contacts)
  /
CREATE TABLE Author OF Author_T
   (ao_id NOT NULL,
    PRIMARY KEY (ao_id));

CREATE TABLE Industry_Based OF Industry_Based_T
   (ao_id NOT NULL,
    PRIMARY KEY (ao_id),
    FOREIGN KEY (ao_id) REFERENCES author (ao_id)
    ON DELETE CASCADE);

CREATE TABLE Academic OF Academic_T
   (ao_id NOT NULL,
    academic_type
    CHECK (academic_type IN ('Research', 'Teaching', NULL)),
    PRIMARY KEY (ao_id),
    FOREIGN KEY (ao_id) REFERENCES author (ao_id)
    ON DELETE CASCADE);

CREATE TABLE Research_Staff OF Research_Staff_T
   (ao_id NOT NULL,
    PRIMARY KEY (ao_id),
    FOREIGN KEY (ao_id) REFERENCES author (ao_id)
    ON DELETE CASCADE);
```

Copyright © 2006, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

Figure 3.36. (continued)

```
CREATE TABLE Teaching_Staff OF Teaching_Staff_T
   (ao_id NOT NULL,
    PRIMARY KEY (ao_id),
    FOREIGN KEY (ao_id) REFERENCES author (ao_id)
    ON DELETE CASCADE);

-- implementation of one-to-many association using ref

CREATE OR REPLACE TYPE Subject_T AS OBJECT
   (code          VARCHAR2(10),
    sub_name      VARCHAR2(20),
    venue         VARCHAR2(10),
    lecturer REF Teaching_Staff_T)
   /
-- implementation of aggregation using a nesting technique

CREATE OR REPLACE TYPE Chapter_T AS OBJECT
   (c_no          NUMBER,
    c_title       VARCHAR2(20),
    page_no       NUMBER)
/
CREATE OR REPLACE TYPE Chapter_Table_T AS TABLE OF Chapter_T
/

CREATE OR REPLACE TYPE Course_Manual_T AS OBJECT
   (isbn          VARCHAR2(10),
    title         VARCHAR2(20),
    year          NUMBER,
    chapter             Chapter_Table_T)
/

CREATE TABLE Course_Manual OF Course_Manual_T
   (isbn    NOT NULL,
    PRIMARY KEY (isbn));

-- implementation of the Publish table

CREATE TABLE Publish
   (author REF Author_T,
    course_manual REF Course_Manual_T);

CREATE TABLE Subject OF Subject_T
   (code NOT NULL,
    PRIMARY KEY (code));
```

Copyright © 2006, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

to determine types and tables that we will need to implement the model. For this case, we need the types Author_T, Industry_T, Academic_T, Research_Staff_T, Teaching_Staff_T, and Subject_T.

For each of them, we will create the table respectively. We also need a type Contacts for the multiple-collection varray of the contact_no attribute in Teaching_Staff_T. Finally, we will need type Course_Manual_T and its table, and also Chapter_T type and Chapter_Table_T type if we decide to use the nested-table implementation in an aggregation relationship.

- **Inheritance relationship.** There are two inheritance relationships in the model. First is the inheritance between Author_T and the subclasses Industry_T and Academic_T. Second is the inheritance between Academic_T and its subclasses Research_Staff_T and Teaching_Staff_T. We will show two methods of implementing inheritance in our sample solution.
- **Association relationship.** There are two association relationships from this model. The first one is between Author_T and Course_Manual_T. If we use a nested table in implementing the aggregation relationship between Course_Manual_T and Chapter_T, we will be able to create a new table using the ref of Author_T and Course_Manual_T in it. The second association is the relationship between Teaching_Staff_T and Subject_T. As it is a one-to-many association, we will need to use the ref of the one side, in this case Teaching_Staff_T, in the many side, Subject_T.
- **Aggregation relationship.** There is one homogeneous aggregation relationship in this model. If we use a nested table, we have to create the type and type table for the part class, and the type and table for the whole class. If we use the clustering technique, we do not need the type, but we do need to create the cluster beforehand using the primary key of the whole class, Course_Manual, and then create an index after that.
- **Complete solution.** The complete solution is shown in Figure 3.36.

Copyright © 2006, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.

Summary

Object-oriented features such as object types, object identity, object references, and relationships are the new object-based features that have been introduced in Oracle™ to enrich traditional RDBMSs with object-oriented characteristics. Using these new features, complex relationships can be implemented, including different semantics of inheritance, associations among different collection types, and aggregation relationships. Although the latest object-oriented Oracle™ has incorporated various object-model features, it still maintains some basic concepts of the relational model such as data integrity and the simplicity of the implementation.

References

Elmasri, R., & Navathe, S. B. (2000). *Fundamentals of database systems* (3rd ed.). Addison Wesley.

Loney, K., & Koch, G. (2000). *Oracle™ 8i: The complete reference.* Osborne McGraw-Hill.

Loney, K., & Koch, G. (2002). *Oracle™ 9i: The complete reference.* Oracle Press.

ORACLE™ 8. (1997). *Oracle™ 8 product documentation library.* Redwood City, CA: Oracle Corporation.

Urman, S. (2000). *Oracle™ 8i advanced PL/SQL programming.* Oracle Press, Osborne.

Chapter Problems

1. A university has a number of books listed as textbooks, each of which may be used by more than one university. A book is published by only one publisher, but one publisher can publish more than one book. Show the implementation of the association relationships above using object refer-

Copyright © 2006, Idea Group Inc. Copying or distributing in print or electronic forms without written permission of Idea Group Inc. is prohibited.