Oracle Objects

Please read the following materials on Oracle Objects and run the examples using your own account. This may take more than one lab hour.

Oracle has traditionally been used as a relation database management system (RDBMS). With the release of **Oracle 8** new extended functionality allows the use of object oriented design. The database designer can now incorporate object-oriented concepts and structures such as UDTs (ADTs), nested tables and varying arrays. This gives designers three different options for Oracle:

Relational	The traditional Oracle relational database
Object-Relational	The traditional Oracle relational database, extended to include object-oriented concepts and structures.
Object-Oriented	An object-oriented database based solely on object-oriented design methods.

Oracle supports all three and users of previous versions do not need to make any changes as the object-oriented capabilities are extensions to the relational database and can be added to existing relational applications or incorporated into new applications (or not used at all).

Benefits of objects

A database object is data and methods to manipulate or observe this data logically grouped together. The use of object-oriented design and database objects results in the following designer and user benefits:

Object Reuse	With object-oriented database objects the ease and chances for reuse are increased. Object reuse reduces work and increases standardization.
Standards	Creating standard object-oriented database objects increases the chance they will be reused. When multiple tables or databases use the same object, a level of standardization is achieved. The applications using the same object will have the same internal format.
Control Data Access	Each database object can have well defined procedures and functions that limit and control access to the data. This standardizes data access and supports reusability of the objects.

The only disadvantage is the time it takes to learn and implement the available object-oriented features.
Oracle supports different types of database objects including:

- User-Defined Data Types /Abstract data types (ADT)
- Nested tables
- Varying Arrays
- Large objects

User-Defined Data Types (Abstract data types)

A UDT consists of one or more subtypes that logically describe the attributes of an object. A common database example would be an address consisting of Street, City, Province, and Postal Code. A UDT could be created containing the following column definitions:

Street	varchar2(30)
City	varchar2(30)
Province	varchar2(30)
Postal Code	varchar2(7)

When you create a table to use this ADT a column is created of type ADDRESS. Each entry in this column would in turn contain the columns of the ADT as defined above. The ADT subtypes are not limited to the standard Oracle data types, they can be user-defined ADTs as well. Functions to change or observe the address data could also be included in the ADT.

The use of ADTs results in two of the benefits listed earlier; reuse and standardization. An ADT creates a standard representation for common data across tables and databases. If the address ADT is used in different applications the data must be in the same format. This encourages and reinforces ADT reuse. Once defined, it is faster and easier to reuse an existing ADT than to define the individual data types.

Nested Tables

A nested table is simply a table within a table. More precisely a nested table is a collection of rows represented as a column in the main table and you can have multiple rows in the nested table for each row in the main table. The number of entries per row is unlimited (limited by OS). A nested table can be considered a one to many relationship within one table without the use of foreign keys. For example; a customer may have many orders. In a strictly relational database, in the first normal form, this data would be in two separate tables. A Customer table with customer details and an Order table with CustomerID as a foreign key. With a nested table the Orders column in the Customer table would contain multiple entries for each CustomerID row. Access to this data does not require a join of two tables.

Varying Arrays

A varying array is similar to a nested table in that it is a collection of objects with the same datatype. A varying array is used like a nested table, the difference being that the size of the array is limited when the array is created.

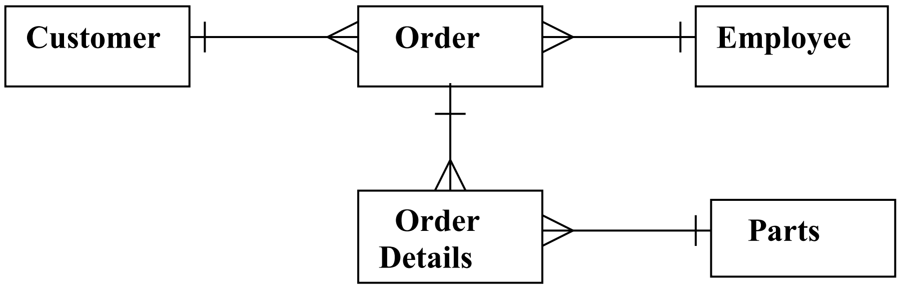
Large Objects

A large object (LOB) is used to store large volumes of data. The types of LOBs available are:

BLOB	Binary LOB of binary data up to 4 GB stored in the database
CLOB	Character LOB of character data up to 4 GB stored in the database
BFILE	Binary file of read-only binary data stored outside the database. Length limited only by the operating system.
NCLOB	A CLOB column that supports a multibyte characterset

Implementation of Oracle Object Data Types

In this section we will be investigating how to use the object type, the varying array and the nested table within Oracle. This is accomplished using a variance of our already familiar SQL statements. We will be implementing various objects types in Oracle using the CREATE, INSERT, SELECT, UPDATE and DELETE statements. We will use a simple database containing the following tables: Employees, Customers, and Orders, OrderDetails, and Parts We've seen several databases of this general format previously in the course and have generally modeled them as follows:



The customer and employee share common attributes such as address and phone numbers. Therefore, this information can be defined within its own type and then used as building blocks within the other types. We will create this database using object types, a varying array, and a nested table.

CREATE TYPE Statement

The **Create Type** statement is used to create all three object types used in this database. It uses the following syntax:

```
create type [schema.]type_name as object (
  attribute_name datatype
  [, attribute_name datatype]…
  | [ member{procedure specification | function_specification}
    [, member {procedure_specification | function_specification}]… ]
  | [ pragma restrict_references (method_name, constraints)
    [, pargma restrict_references (method_name, constraints)}… ]  );
```

In this statement, you give the new object_type a name. Then you must include at least one, but may have more, attributes, whose data types could be any Oracle data type, including another object_type. After the attributes, you may list the prototype specifications of the member functions or procedures, if any. Note that the actual body of the function or procedure is defined separately in another statement, which we will cover later. Lastly, we include any pragmas. Pragmas are compiler directives that impose some restrictions on the function or procedure. For instance, you can specify that the function cannot write to the database (WNDS). We will see all of these later in our example.

Object type (Abstract data type)

The first step in setting up our object-relational database is to create the object types defined in our design. First, we will create the object type for the address, to be used in both the Customer and the Employee tables:

```
create type address_type as object (
  street   varchar2(30),
  city     varchar2(30),
  state    varchar2(20),
  pcode    number(7)
);
```

When this statement is executed, the object type is created, along with its default methods to construct and access the data members. If we wanted to define some user-defined methods, we would include their specifications in this statement. We will see this later.

Varying Array Collection Type (VARRAY)

Next we will create a varying array collection type to store the telephone numbers. The general format of this create type statement is as follows:

CREATE TYPE type_name AS VARRAY (limit) OF datatype;

In this statement, *type_name* is the name given to the new varying array type, *limit* is the maximum number of values allowed in the varying array, and *datatype* can be any Oracle or user-defined data type. So, to create a varying array of telephone number, we would use the following:

create type phones_varray_type as varray(3) of char(12);

Using the two preceding data types, we can now create another type to represent a person:

```
create type person_type as object (
  name     varchar2(30),
  address  address_type,
  phones   phones_varray_type);
```

This type consists of the person's name, address (using our address_type) and telephone numbers (using our phones_varray_type).
Now, using the new person_type object type, we can create our Customer and Employee tables:

```
create table employees (
  eno      number(4) not null primary key,
  person   person_type,
  hdate    date
);

create table customers (
  cno      number(5) not null primary key,
  person   person_type
);
```

User-defined methods

Now that we've got these tables created, we'll concentrate on the Orders table. The orders table consists of orders and the order details. We know that this would be handled in a relational database using two tables, an Order table and an OrderDetail table. In an object-relational database, it will be handled somewhat differently. First, we will write a statement to create an object type to represent the order details:

```
create type details_type as object (
  pno      number(5),
  qty      integer,
  member function cost return number,
  pragma restrict_references(cost,WNDS)
);
```

The *details_type* object type contains two data members (pno and qty) and a method called *cost* which calculates and returns the cost of this detail item in the order. Although this particular function has no input parameters, parameters are allowed in the function or procedure specification, similar to a PL/SQL function or procedure.
The *pragma* statement in the function prototype specification prevents the function from updating anything in the database. The available restrictions are:

. WNDS: Write No database State (no database updates allowed)
. RNDS: Read No Database State (no database queries allowed)
. WNPS: Write No Package State (no values of package variables can be modified)
. RNPS Read No Package State (no package variables can be referenced)

The statement above creates only the prototype specification of the *cost* member function. The body of a member function or procedure is created using the *create type body* statement. For example the body of the above *cost* function is specified using the following PL/SQL code.

First we need to create the Parts table. This table is a straight-forward relational table. It contains no object types, varying arrays, or nested tables. This table will be used by the *cost* function to determine the price of the specified item.

```
create table parts(
  pno      number(5) not null primary key,
  pname    varchar2(30),
  qoh      integer check(qoh >= 0),
  price    number(6,2) check(price >= 0.0),
  olevel   integer);
```

Now we can specify the body of the *cost* function:

```
create type body details_type as
member function cost return number is
  p parts.price%type;
begin
  select price into p from parts where pno = self.pno;
  return p * self.qty;
end;
```

```
end;
```

Nested Tables

The *Nested Table* is the second type of collection object supported by Oracle. It differs from the *Varying Array* collection object in that a nested table can support any number of entries per row. No upper limit is set. As the name indicates, these tables are nested within other tables. To create a nested table, you use the *Create* statement with the following syntax:

```
CREATE TYPE type_name AS TABLE OF datatype;
```

Where *type_name* is the name of the nested table and *datatype* is any valid Oracle8 datatype or object type.

For this example, we will create a table object type consisting of previously defined *details_type* objects:

```
create type details_ntable_type as
  table of details_type;
```

Now we can create another object type that will represent an order in the example database. This *order_type* object type will contain the information about all the items in a particular order. No longer are the order items stored separately from the orders themselves.

```
create type order_type as object (
  ono      number(5),
  details details_ntable_type,
  cno      number(5),
  eno      number(4),
  received date,
  shipped  date,
  member function total_cost return number,
  pragma restrict_references(total_cost,WNDS)
);
```

The *order_type* object type contains an order number (ono), a nested table of *details_ntable_type* type, the customer number of the customer placing the order, the employee number of the employee taking the order, the date the order was received, and the date the order was shipped. It also contains a member function *total_cost* that calculates and returns the total cost of the order. The code to create the body specification for the *total_cost* function follows:

```
create or replace type body order_type as
  member function total_cost return number is
    i          integer;
    item       details_type;
    total      number := 0;
    item_cost  number;
  begin
    for i in 1..self.details.count  loop
      item := self.details(i);
      item_cost := item.cost();
      total := total + item_cost ;
    end loop;
    return total;
  end;
end;
```

In the loop in the function, the cost of each detail item in the order is calculated using the *details_type* member function *cost*. These item totals are then summed to determine the total cost of the entire order. Note that in this function, the keyword "self" is used. This keyword allows the member function to reference attributes within its object. Nested tables can be treated like PL/SQL tables when accessed from within PL/SQL code.

Object Tables

In Oracle8, you can create tables that are made up of objects instead of rows. After you have defined an object type, you can then define a table of that object type. In our example database, we will create a table for the orders consisting of objects of the above *order_type* object type.

```
create table orders of order_type (
  primary key (ono)
)
nested table details store as details_tab;
```

There are several items to notice about this statement. The first item is the use of the keyword "of" in the first line when specifying the object type that makes up the objects of the table. Also, the primary key constraint for the attribute "ono" is specified in this *create* statement. The third item is the clause at the end of the statement. This clause specifies that the nested table attribute "details" is to be stored as an external table using the name "details_tab". This clause is required for any table that includes a nested table.

INSERT Statement

Just as with the regular relational tables, the SQL "INSERT INTO" statement is used to insert data into object-relational tables. However, the syntax of these INSERT statements is modified somewhat to make use of the object type constructors. These constructors have the same name as the object type. The exact syntax of the INSERT statement depends on the structure of the table into which information is being inserted.

Inserting Rows

To insert the information about a new employee into the previously created Employees table:

```
insert into employees values
  (1000,
   person_type('Fred Jones',
             address_type('123 Main St','Kamloops','BC',V2A 1A1),
             phones_varray_type('250-555-1212',null,null)),
  '12-DEC-95');
```

The INSERT statement uses a nested syntax that mirrors the nesting in the Employees table itself. The constructor for each of the included object types is called along with a list of input parameters. This creates the object instances for the column values. You can also include a NULL for an object type if there is no information to enter. This is shown above where some of the telephone number values are set to NULL, and in the following INSERT statement, where the address_type is set to NULL.

```
insert into employees values
  (1002,
   person_type('John Brown',
              null,
              phones_varray_type('780-555-1111',null,null)),
  '01-SEP-94');
```

Inserting Rows into Nested Tables

To insert a row into a table that contains a nested table, you use a statement similar to that used above for the varying array of telephone numbers.

```
insert into orders values
  (1020,
   details_ntable_type(details_type(10506,1),
                    details_type(10507,1),
                    details_type(10508,2),
                    details_type(10509,3)),
   1111,1000,SYSDATE, SYSDATE + 1);
```

Looking back to the CREATE statement for the Orders table, you might conclude that since this is a table of objects, you would have to call the constructor for the *order_type* object type after the keyword "values". But Oracle allows object tables to be views as relational tables. So both types on INSERT statements (with and without the object constructor) are valid.

SELECT Statements (Queries)

To query the tables in an object-relational database, you use a modified version of the SELECT statement. For example, to access nested attributes, you use the "dot" notation. We can see this in the following SELECT statement to retrieve the names and street addresses of all employees hired since January 1st, 1993.

```
select   e.person.name, e.person.address.street
from     employees e
where    e.hiredate >  ……………………………….
```

Using the dot notation, you can get right down to the innermost attributes of the table. The next SELECT statement performs a join on the *pcode* attributes of both the Employees and Customers tables to determine the list of customers and employees who live in the same postal code area.

```
Select   e.person.name EMPLOYEE, c.person.name CUSTOMER
from     employees e, customers c
where    e.person.address.pcode = c.person.address.pcode
```

Querying tables containing the simpler data structures is fairly straight-forward. Querying nested tables and varying arrays gets more involved. You can create SELECT statements to query nested tables, but the syntax to achieve this is different from regular SQL syntax. The reason for this is that the result of any SELECT operation is a table. But if the table contains a nested table, then the result would be a table within a table. You need to specify the “THE” operator. This operator basically flattens the nested table. You supply an inner SELECT and you need to also supply an alias for this flattened table so that it can be referenced by the outer SELECT. The correct format for this type of SELECT statement therefore is:

```
select   nt.pno, nt.qty
from     THE(   select   o.details
                from     orders o
                where o.ono = 1020) nt
where nt.qty > 1;
```

The above query represents the question: what are the *pno* and *qty* values for parts ordered in order with ono = 1020 and qty > 1. It accesses the nested table column *details* in the Orders table. The nested table can appear in the FROM clause of the SELECT statement only if it appears in the select-list of a nested SELECT statement and this nested SELECT statement is enclosed as an argument to the special THE operator. Note that both the outer and the inner SELECT statements have WHERE clauses, indicating that you can restrict the rows from the nested table as well as the main table in the same overall query. You must ensure that your nested SELECT statement will not result in more than one nested table.

Queries can call the user-defined methods of an object. In the following query, the total cost of a particular order is returned.

```
select   sum(nt.cost())
from     THE(   select   details
                from     orders o
                where o.ono = 1020) nt;
```

The above query could also have been written as:

```
select   o.total_cost()
from     orders o
where    o.ono = 1020;
```

Unfortunately, you cannot query a varying array in SQL; you must use PL/SQL and its procedural notation. PL/SQL allows varying arrays in the database to be read into PL/SQL variables. This anonymous block read in and displays the *phones* varying array for employee 1111.

```
declare
   enum                 employees.eno%type;
   a_phones     employees.person.phones%type;
begin
  enum := 1111;

  selecte.person.phones
  into    a_phones
  from   employees e
  where e.eno = enum;
  for I = 1..3 loop
    if (a_phones(i) is not null) then
      dbms_output.put_line('Phone for employee ' || enum || ' is ' || a_phones(i));
    end if;
  end loop;
end;
```

Another situation that arises when dealing with objects in Oracle databases is the access of object data members from a SELECT statement. An object table has no column names. Therefore, it is impossible to select any named columns from the table. To access the row objects, you use the "value" operator. The table alias is supplied to the operator as an input value, and the operator returns the row object. An example of the use of the "value" operator is given below.

```
declare
   enumber       employees.eno%type;
   total         number(10,2);

  function total_empl_slaes( emp_no     in        employees.eno%type)
    return number is
  sales          number := 0;
  cursor c       is
    select       value(o)
    from         orders o
    where        o.eno = emp_no;

  e_order        order_type;

  begin
    open c;
    loop
         fetch c into e_order;
         exit when c%notfound;
         sales := sales + e_order.total_cost();
    end loop;
    return(sales);
  end;

' mainline
begin
  enumber := 1001;
  total := totalemp_sales(enumber);
  dbms_output.put_line('Total sales for employee ' || enumber || ' is ' || total);
end;
```

In this anonymous block, the order_type object's "total_cost" method is used to calculate the total cost of a particular order. These totals are summed into a grand total for all the orders belonging to the specified employee.

UPDATE and *DELETE* Statements

UPDATE and DELETE statements for object-relational databases are very similar to those of purely relational tables. For example, to update the street address for a particular customer, you would use the following statement:

```
update  customers c
        Set     c.person.address.street = ‘111 Victoria Avenue’
        where   c.cno = 1234;
```

To delete a record from the Customer table, we would use the following statement:

```
delete  from customer c
where   c.person.address.street like ‘111%’;
```

Once again, dealing with varying arrays in these situations is a bit more complicated. Since it is impossible to access varying arrays from SQL, you have to user PL/SQL. In PL/SQL, the object’s varying array is copied into PL/SQL array and dealt with from there.

```
declare
  enumber       employees.eno%type;
  a_phones      employees.person.phones%type;

begin
  enumber := 1111;

  select        e.person.phones
  into          a_phones
  from          employees
  where         e.eno = enumber;

  for I in 1..3 loop
        if (a_phones(i) is not null) then
          dbms_output.put_line(‘Phone for employee ‘ || enumber || ‘ is ‘ || a_phones(i));
        end if;
  end loop;
end;
```

Basic Rules for the Use of VARRAYs and Nested Tables

The following rules must be considered when using VARRAYs and nested tables:

- A VARRAY or nested table cannot contain a VARRAY or nested table as an attribute.
- Constraints of any kind cannot be used in type definitions. Rather, they must be specified using the ALTER TABLE command.
- The scalar parts of a type can be accessed directly on a parent table
- A “store table” must be specified to store the records of a nested table. Store tables inherit the physical attributes of the parent table.
- Default values cannot be specified for VARRAYs.
- A table column that is specified as VARRAY cannot be indexed.
- A table using VARRAYs or nested tables cannot be partitioned.
- VARRAYs cannot be directly accessed using SQL.

References:

Koch, George, and Kevin Loney. ***ORACLE8 The complete Reference.*** Berkeley, CA: Oracle Press, Osborne/McGraw-Hill , 1997

Sunderraman, Rajshekhar. ***ORACLE8 programming: A Primer***. Georgia State University: Addison-Wesley 2000

1. What are the three database design options with Oracle8?

2. What are the three benefits of objects?

3. Name three different types of database objects.

4. What is a major difference between a nested table and a varying array?

5. Can a varying array be accessed using SQL?

Answers:

1. The three database design options are:
Relational
Object-relational
Object-oriented

2. The three benefits of objects are:
Object reuse
Standards
Control Data Access

3. Types of database objects are:
ADT (object types)
Nested tables
Varying arrays
Large objects

4. The size of a varying array must be given when the array is created. The size of a nested table is not set.

5. No. It must be accessed using PL/SQL.