User Defined Datatypes in Oracle8

User defined datatypes are objects in which users formalize the data structures and operations that appear in their applications. This is best illustrated by an example

Assume you have the following relation table structure

Student_info Table

```
(st_no  number(5),
 st_name        varchar2(50),
 st_age number(2),
 pobox varchar2(15),
 street  varchar2(30),
 city            varchar2(30),
 country   varchar2(30))
```

If you look at the last four fields, you recognize that they make up the address of the student. In Oracle8, we can create an object called Address based on the fields that appeared in the Student_info as follows:

```
SQL>Create Type Address AS OBJECT (
        pobox  varchar2(15),
        street   varchar2(30),
        city            varchar2(30),
        country   varchar2(30)
         );
```

```
SQL> DESC ADDRESS
 Name                          Null?           Type
 ----------------------------  --------        ---------------------
 POBOX                                         VARCHAR2(15)
 STREET                                        VARCHAR2(30)
 CITY                                          VARCHAR2(30)
 COUNTRY                                       VARCHAR2(30)
```

Now, we can rewrite the CREATE TABLE statement for the original statement as follows:

```
SQL> CREATE TABLE STUDENT_INFO (
        st_no           number(5),
      st_name           varchar (50),
        st_address      address );
```

st_address is now called **Object Column**

```
SQL> desc student_info
 Name                           Null?            Type
 ------------------------------ -------- ----------------
 ST_NO                                   NUMBER(5)
 ST_NAME                                 VARCHAR2(50)
 ST_ADDRESS                              ADDRESS
```

The new datatype ADDRESS was used as if it is built-in datatype. Ofcourse, it is a user defined datatype.

Inserting Data into STUDENT_INFO :

The INSERT command is a little different when it comes to an object column:

```
SQL> INSERT INTO STUDENT_INFO
        VALUES (1000,'AMMAR SAJDI',
        ADDRESS('17187','ALI ASOUH', 'AMMAN',
        'JORDAN'));

1 row created.
```

Note the underlined word in the above SQL statement. When we insert a new record, we write the name of the user-defined type before inserting the data for it

```
ADDRESS('17187','ALI NASOUH', 'AMMAN',
        'JORDAN')
```

The expression ADDRESS is called *CONSTRUCTOR METHOD.* It is an activation of the constructor function for the type ADDRESS. The CONSTRUCTOR METHOD is what makes a new object according the object type's specification. The name of the constructor method is the name of the object type.

```
SQL> SELECT * FROM STUDENT_INFO;
ST_NO  ST_NAME       ST_ADDRESS(POBOX, STREET, CITY, COUNTRY)
---------- --------------      ----------------------------------------------------------------
  1000 AMMAR SAJDI    ADDRESS('17187', 'ALI NASOUH', 'AMMAN',
                         'JORDAN')
```

To Select the POBOX only, Use the following Syntax

```
SQL> select  d.st_address.pobox  from student_info  d;
```

ST_ADDRESS.POBO
---------- ---------- ---------- ---------- ---------- ---------- ---------- ---------- ---------- ----------
17187

Note the use of the required Alias for the table_name

If you try to remove the Alias "d" in the above statement, you will get an error

Creating an index on student_info:

It is required that you create an index on the country field of the student_info. Remember that this field is embedded within the st_address object column

SQL> *Create index IND1 on Student_Info (st_address.country);*

We can also define what is called *Object Tables.* An object table is a special kind of table that holds objects and provides a relational view of the attributes of those objects.

For example: we can create an Object type called family_member as follows:

```
SQL> CREATE OR REPLACE TYPE FAMILY_MEMBER AS OBJECT
   (MEM_ID   NUMBER,
    MEM_NAME VARCHAR2(30),
    MEM_AGE  NUMBER(2),
   MEM_SEX  CHAR(1));
   /
```

Then define an Object table for the objects of the *familyl_member* type:

```
SQL>  CREATE TABLE FAMILY OF FAMILY_MEMBER;
Table created.
```

You can now insert values in the FAMILY table as you would do in the relation model. i.e.

```
SQL> INSERT INTO FAMILY VALUES (1,'AMMAR SAJDI', 34,'M');

1 row created.
```

To View the data in *Object Tables,* you can either view it as multi-column record:

```
SQL> SELECT * FROM FAMILY;

 MEM_ID        MEM_NAME                       MEM_AGE  M
```

| --- | --- | --- | --- |
| *1* | *AMMAR SAJDI* | *34* | *M* |

or as a single column table

SQL> SELECT VALUE (X) FROM FAMILY X;

VALUE(P)(MEM_ID, MEM_NAME, MEM_AGE, MEM_SEX)

FAMILY_MEMBER(1, 'AMMAR SAJDI', 34, 'M')

Definition: Objects that appear in object tables are called row objects and objects that appear in table columns or as attributes of other objects are called column objects

To Drop a type, Use the Drop command as follows:-

SQL> Drop Type ADDRESS

Important Note: You cannot drop a user defined TYPE if there is a table or another type that uses it. You need to drop the depended object or table first. Then drop the TYPE. Since table STUDENT_INFO depends on the TYPE ADDRESS. an attempt to delete ADDRESS will generate the following error

ERROR at line 1:
ORA-02303: cannot drop or replace a type with type or table dependents

COLLECTION TYPES

The collection types are *array types* and *table types*, Each describes a data unit made up of an indefinite number of elements, all of the same datatype. The corresponding data units are called *VARRAYS* and *Nested Tables.*

ARRAY TYPES:

An array is an *ordered* set of data elements. All elements of a given array are of the same datatype. Each element has an *index*, which is a number corresponding to the element's position in the array. The number of elements in an array is the size of the array. Oracle allows arrays to be of variable size, which is why they are called VARRAYs. You must specify a maximum size when you declare the array type.

Assume you have the following Table structure

Contact_Name Varchar2(200),
Phone1 Varchar2(15),
Phone2 Varchar2(15),
Phone3 Varchar2(15),
Phone4 Varchar2(15)

Using Oracle8 Object option, you can replace the four repitions of Phone number with the new VARRAY type as follows:

SQL> CREATE TYPE phones AS VARRAY(4) OF VARCHAR2(15);

The preceding statment defines a new type and does not allocate any space for it

Now, you can create the previous structure as follows:

```
SQL> CREATE TABLE contacts
        (Contact_Name       varchar2(200),
         Phone_list         Phones);

SQL> desc contacts
 Name                           Null?    Type
 ------------------------------ -------- ----------------------
 CONTACT_NAME                            VARCHAR2(200)
 PHONE_LIST                              PHONES
```

To insert a record into the contacts table, use

```
 SQL>  insert into contacts values
  ('ammar',phones('962-6-826601','962-6-5661356'))
```

1 row created.

Note: All Elements of VARRAY are stored in a single column.

To Query the phones

SQL> Select phone_list from contacts;

```
PHONE_LIST
----------------------------------------
PHONES('962-6-826601', '962-6-5661356')
```

Querying individual elements of the VARRAY requires an operation called CASTING. It will be introduced later when we discuss *Nested Tables*.

Note again that the Phone_list array was filled using the Constructor Function *PHONES.* Remember again that the name of the constructor method is the name of the object type, and it I s this method that makes a new instance of this object.

IMPORTANT NOTE:

Once this array type is defined you can use in the following situations:

1) A datatype of a column in the CREATE TABLE clause.
2) A PL/SQL variable or function return type
3) An object type attribute

NESTED TABLES:

A *nested table* is an unordered set of data elements, all of the same datatype. It has a single column, and the type of that column is a built-in type or an object type. If an object type, the table can also be viewed as a multi-column table, with a column for each attribute of the object type.

Example:

We want to represent two tables. The first table is a master table that describes a company, and the second table is a detailed table that describes the employees in this table. In a relational point of view the following structures describe this representation.

Company
Comp_id Number,
Comp_name varchar2(100),
Capital Number

Employees
Emp_no Number,
Comp_id Number,
Emp_name varchar2(30),
Emp_title Varchar2(30)

Now a new type that defines an employee is going to be created.

```
SQL> Create type employee as object
        (emp_no          Number(3),
        Comp_id          Number(3),
        emp_name     varchar2(30),
        title        varchar2(30))
```

Type created.

The next step is to Create an *Table Type*, which is a type used for the nested table employees

SQL> Create type employees as TABLE of Employee;

Type created

A table type does not allocate space. It merely defines a type, which can be used as:

The datatype of a column of a relational table.
The object type attribute.
A PL/SQL variable of function return.

The last step is to create the Company table (the master table) with the employees as a nested table.

```
SQL> Create table company
        (comp_id            number,
        comp_name           varchar2(100),
        capital                 number,
        employee_list       employees
)
   nested table employee_list STORE as employees_table;
```

Table created.

SQL> desc company

Name	Null?	Type
COMP_ID		NUMBER
COMP_NAME		VARCHAR2(100)
CAPITAL		NUMBER
EMPLOYEE_LIST		EMPLOYEES

The last SQL statement create COMPANY table with a nested table called employees_table of type Employees.

Therefore, the rows of a nested table are stored in a separate storage table. We supplied a storage tablename (employees_table) when we defined the table containing the nested table. For each nested table in the table definition, the associated storage table contains the rows of all instances of the given nested table in the rows of the parent table.

if You execute the command "Select * from tab" you will find that there exists a table called employees_table. Now try to select from that table

*SQL> Select * from Employees_table;*
*select * from employees_table*
**
ERROR at line 1:*
ORA-22812: cannot reference nested table column's storage table

Indexing a column in the nested table

SQL>Create Index ind1 on employees_table (empno);

INSERTING DATA

Inserting a record into the Compnay table with no employees.

SQL> Insert into company values
(1,'Palestine_engineering',100,000,employees());

1 row created.

Inserting a record into the Nested Table

SQL> Insert into THE (
select p.employee_list from company p
where p.comp_id=1)
values (1,1,'AMMAR SAJDI','EXECUTIVE MANAGER');

THE clause informs Oracle that the column value returned by the subquery is a nested table and not a scalar. A Subquery prefixed by THE is called a flattend Subquery. More about that later.

OR

you can insert into the *COMPANY* table and the employess nested table at once.

SQL>Insert into company values (1, 'PALESTINE ENGINEERING',90000,employees(employee(1,1,'AMMAR SAJDI','EXECUTIVE_MANAGER')))

OR

*SQL> Insert into Company values (1,'PALCO',1000 ,CAST(MULTISET(SELECT * FROM EMP) AS EMPLOYEES));*

The above statement will insert the records in the EMP table in the nested table of Company

Flattened Subquery:

To manipulate the individual rows of a nested table stored in a database column, use the keyword THE. You must prefix THE to a subquery that returns a single column value or an expression that yields a nested table. If the subquery returns more than a single column value, a run-time error results. Because the value is a nested table, not a scalar value, Oracle must be informed, which is what
THE does.
The following example adds a new row to department 40's nested table stored in column PROJECTS:

```
INSERT INTO
THE(SELECT projects FROM dept WHERE deptno = 40)
VALUES(33, 'Install new email system', 14875);
```

This example increases the budgets for two projects assigned to department 70:

```
UPDATE
THE(SELECT projects FROM dept WHERE deptno = 70)
SET budget = budget + 1000
WHERE projno IN (24, 25);
```

SELECTING FROM NESTED TABLES

SQL> Select p.comp_id, p.comp_name , employee_list
from company p;

COMP_ID COMP_NAME EMPLOYEE_LIST(EMP_ID, COMP_ID, EMP_NAME, TITLE)
---
1 PALESTINE ENGINEERING EMPLOYEES(EMPLOYEE(1, 1, 'AMMAR SAJDI', 'EXECUTIVE_MANAGER'))

To Get the Inforamtion from the nested table in tabular format

*SQL> Select * from The (Select d.employee_list from Company d*
Where d comp_id = 2);

To Find a particular column from the nested table:-

SQL> Select T.emp_name from The (Select d.employee_list from Company d
Where d comp_id = 2) T;

Type Casting

CAST allows you to convert collection-typed values of one type into another collection type. You can cast an unnamed collection (such as the result set of a subquery) or a named collection (such as a VARRAY or a nested table) into a type-compatible named collection. The type_name must be the name of a collection type and the operand must evaluate to a collection value.

The following statement will create a nested table of the same type as the PHONES VARRAY created earlier.

SQL> Create type nested_phone as table of varchar2(15);
/

Type created.

Then, the following statement converts the Phone_list attribute of the contacts table into a nested table type using the CAST operator. As stated earlier in the VARRAY section, this method allows as to select individaul records existing in VARRAY

*SQL> Select * from the (select cast(d.phone_list as nested_phone) from contacts d);*

Please note that the usage of the alias for the table name is essential

COLUMN_VALUE

962-6-826601
962-6-5661356

When to Use *Nested Tables* and When to Use *Varrays*

Nested Tables **are used when**:

Querying the contents of the the data is required. As shown by the CAST example, it is not easy to query the individual records of VARRAY.

Indexing is required. It is not possible to index VARRAYs.

The ordering of information is not required as nested tables are unordered set of records

Upper value of the number of records is unkown. VARRAYs require the specification of an upper bound on the number of elements.

VARRAYs **are used when:**

The order of the information might be important. VARRAYs are ordered, while Nested tables are not ordered.

Upper bound for the number of records is small. VARRAYs force you to specify a maximum number of elements in advance *(4 Phone numbers in our previous VARRAY example)*. They use storage more efficiently than nested tables.

There is no need to query the individual elements of the VARRAY.

RELATIONSHIPS and REFs

In the relational model, foreign keys are used to express the one-to-many relationship. In the object option of Oracle8, other means are used to express the one-to-many relationships when the one side of the relationship is a row Object (See definition of row object presented earlier). Oracle gives every row object a unique identifier called *object identifier(*sort of a pointer). An object identifier allows the corresponding row object to be referred to from other objects or from relational tables. A built is datatype called REF represents such reference. A REF encapuslates a reference to a row object of a specified object type

The following is an example that establishes a relationship between a table called COLLEGES and another table called STUDENTS. Each college consists of many students.

```
SQL> Create or replace type college_info as object
        (college_no          number(4),
         college_name        varchar2(20),
        college_location     varchar2(20));
```

The following statement create an *object table* of type *college_info*

```
SQL> Create table COLLEGES of college_info;
```

Table created.

To insert one record into the COLLEGES table

```
SQL> Insert into COLLEGES values (college_info(1,'Electrical Eng','AMMAN'));
```

1 row created.

To create the STUDENTS table with Student_college of Type REF so that it can set a pointer to Students table

```
Create table STUDENTS
(Student_no          number(4),
```

Student_college REF College_info,
Student_name varchar2(20),
Student_sex Char(1),
Student_age number(2))

Now, Insert a record into the STUDENT table

SQL> *INSERT into students*
SELECT 1 , ref (C),
'Ammar Sajdi','M','22'
FROM colleges C
WHERE C.college_no=1;

1 row created.

The preceding statement constructs a REF given to the record whose college_no columm is = 1 in the COLLEGES table and inserts this value in the second column which is the STUDENT_COLLEGE column.

To have an idea about what typical value for REF might be, you can execute the following statement,

SQL> select student_college from students

STUDENT_COLLEGE

0000220208C20383C6D2C411D18B1E0040054A9C55C20383C5D2C411D18B1E0040054A9C55

The following statement will join the Student record for student_no 1 with its matching college. The matching college information was constructed using the DEREF function on the column *Student_college* which contains number structure shown above.

SQL> SELECT p.student_name, deref(p.student_college)
from students p
where p.student_no=1;

STUDENT_NAME

DEREF(P.STUDENT_COLLEGE)(COLLEGE_NO, COLLEGE_NAME, COLLEGE_LOCATION)

Ammar Sajdi
COLLEGE_INFO(1, 'Electrical Eng', 'AMMAN')

OR

```
SQL>  SELECT p.student_name, P.student_college.college_location
  from students p
  where p.student_no=1
```

Note: Despite the fact that table *STUDENTS* references table *COLLEGES*, you can still delete table *COLLEGES* when there are students enrolling in that college.

Scoped REFs

In declaring a column type, collection element, or object type attribute to be a REF, you can constrain it to contain only references to a specified object table. Such a REF is called a *scoped REF*. Scoped REFs require less storage space and allow more efficient access that unscoped REF's.

Syntax for creating a *SCOPED* REF

The following is a statement that defines a scope for the Student_college column in the STUDENTS table:

```
Create table STUDENTS
(Student_no          number(4),
 Student_college     REF College_info,
 Student_name        varchar2(20),
 Student_sex         Char(1),
 Student_age         number(2),
SCOPE for (student_college is COLLEGES)
```

OR you can *ALTER* an existing table as follows:

```
SQL> ALTER TABLE STUDENTS
        ADD (SCOPE FOR (Student_College) IS      COLLEGES);
```

METHODS

It was mentioned earlier that each *OBJECT TYPE* has a *name* which identifies it uniquely. It has *Attributes* which are either built-in types or other user-defined types that can describe the sturcture of a table.

Now, we are going to introduce the last component of *OBJECT TYPES* which is *METHODS. Methods* are real function or procedures written in PL/SQL and stored in

the database, or written in a language like C and stored externally. Methods implement operations the application can perform

Lets go back to our COLLEGE_INFO Object and modify by adding a method to it:

```
SQL>  create or replace type college_info as object
  (college_no   number(4),
   college_name  varchar2(20),
   college_location  varchar2(20),
  member function total return
   number,
 PRAGMA RESTRICT_REFERENCES (TOTAL, WNDS, WNPS)
```

NOTE: If it fails to create you need to drop the dependent objects ie. DROP TABLE COLLEGES, DROP TABLE STUDENTS.

Now recreate the Tables COLLEGES AND STUDENTS as show above and insert some records in them.

Now, we will create the function *TOTAL*

```
SQL>create or replace type body college_info as
MEMBER FUNCTION  TOTAL RETURN NUMBER IS
  X  NUMBER;
BEGIN
  SELECT COUNT(*) INTO X FROM STUDENTS ;
  RETURN (X);
END;
END;
```

The following Statement will invoke the *TOTAL* method function:

```
SQL> SELECT C.COLLEGE_NO,  C.TOTAL ( ) FROM COLLEGES C;

COLLEGE_NO        C.TOTAL()
------------------        --------------
       1                  2
```

Comparison methods

In the previous examples, we have seen how to implement methods function that can be used to process certain functionality. Methods also play a role in comparing objects. It is easy to compare two data items of a given **built-in** type, and determine whether one is greater than, equal to, or less than the other. However, one cannot compare two items of an arbitrary user-defined type without further guidance. There are two ways to define an

order relationship among objectsof a given object type: *MAP* method and *ORDER* method

MAP METHOD:

The ability to compare normal built-in types is used in determining comparison using the map method.

examine the following:

```
SQL>  CREATE TYPE RECTANGLE AS OBJECT
  ( LENGTH  NUMBER(2),
    WIDTH   NUMBER(2));
```

One can define a map method area that return a numer, namely, the product of the rectangles *Length* and *Width* attributes. Now, the two rectangles can be compared by comparing their areas

```
SQL>  CREATE OR REPLACE TYPE RECTANGLE AS OBJECT
  ( LENGTH  NUMBER(2),
    WIDTH   NUMBER(2),
 MAP MEMBER FUNCTION
   area   RETURN NUMBER
PRAGMA RESTRICT_REFERENCE(
 AREA, WNDS,WNPS,RNPS,RNDS)
);


SQL> CREATE TYPE BODY RECTANGLE IS
 MAP MEMBER FUNCTION AREA  RETURN NUMBER IS
        BEGIN
                RETURN (LENGTH*WIDTH)
        END;
END;
```

NOTE: THE MAP METHOD MUST BE DECLARED WITHOUT ANY PARAMETERS

A table will be created and filled with data:

```
SQL> CREATE TABLE SHAPES OF RECTANGLE

Table created.

SQL> INSERT INTO SHAPES VALUES    (RECTANGLE(1,10));
```

1 row created.

SQL> INSERT INTO SHAPES VALUES (RECTANGLE(2,3))

1 row created.

SQL> INSERT INTO SHAPES VALUES (RECTANGLE(3,3))

1 row created.

SQL> INSERT INTO SHAPES VALUES (RECTANGLE(2,6))

1 row created.

SQL> COMMIT;

Commit complete.

SQL> SELECT P.LENGTH, P.WIDTH FROM SHAPES P
ORDER BY VALUE(P); --MAP INVOKED EXPLICITLY

LENGTH	WIDTH
2	3
3	3
1	10
2	6

OR

SQL> SELECT P.LENGTH, P.WIDTH FROM SHAPES P ORDER BY p.area()

LENGTH	WIDTH
2	3
3	3
1	10
2	6

Note that the the output is ordered according to the area ascending.

SQL> select * from shapes p
where p.area() > 7;

LENGTH	WIDTH
1	10
3	3
2	6

Order Method

Order method are more general. An order method uses its own internal logic to compare two objects of a fiven object type. It returns a value that encodes the order relationship. It our defined type called *address*, the terms greater than and less than may have no meaning for addresses in your application, but you may need to perform complex computations to determine when two addresses are equal

Constraints and Object tables

You can define constraints on an object table just as you can on other tables;

SQL>ALTER TABLE COLLEGES ADD CONSTRAINT COLL_PK primary key (college_no);

Indexes

You can define indexes on an object table or on the storage table for a nested table column or attribute just as you can with other tables. Earlier, we created a type called *Address* as follows:

```
SQL>Create Type Address AS OBJECT (
        pobox  varchar2(15),
        street   varchar2(30),
        city            varchar2(30),
        country   varchar2(30)
         );

SQL> CREATE TABLE STUDENT_INFO (
        st_no           number(5),
      st_name           varchar (50),
        st_address      address );
```

The following example defines an inde on an attribute of an object column

SQL> *Create index IND1 on Student_Info (st_address.country);*

Privileges

System privileges

CREATE TYPE
CREATE ANY TYPE
ALTER ANY TYPE
DROP ANY TYPE
EXECUTE ANY TYPE

NOTE: The CONNECT and RESOURCE roles include the CREATE TYPE system privilege.

Object Privileges

The only object privilege that applies to user_Defined types is EXECUTE.

EXECUTE on a user defined types allows you to use the type to

- Define a table
- Define a column in a relational table.
- Declare a variable or parameter of the named type

Execute lets you invoke the type's methods, including the constructor