Database Technologies - Object Relational Databases

Reading

Object Relational Features

Oracle Rel. 19C Object-Relational Developer’s Guide

- Chapter 1 Introduction to Oracle Objects
- Chapter 2 Basic Components of Oracle Objects
- Chapter 3 Support for Collection Types

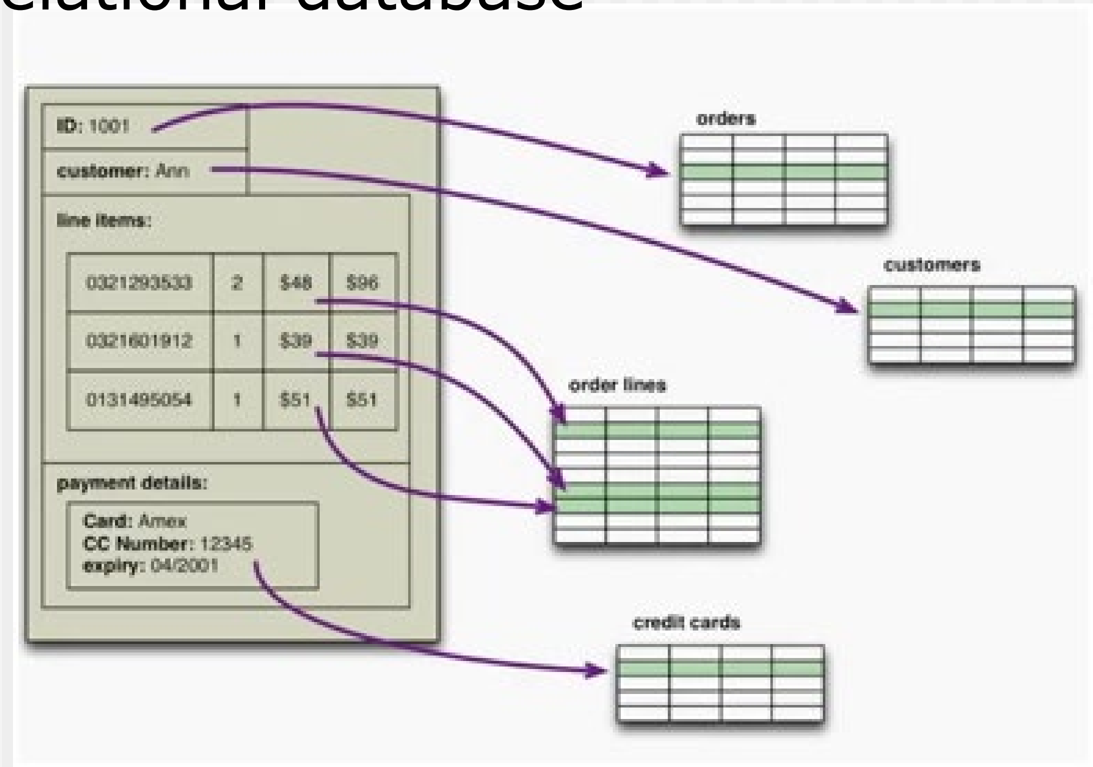
https://docs.oracle.com/en/database/oracle/oracle-database/19/adobj/object-methods.html#GUID-06384723-F483-484B-A0D0-18CE41810392

Merging Relational and Object Models

- **Object-oriented models** support interesting data types --- not just standard types files.
  - Maps, multimedia, collections etc.
  - Inheritance; Encapsulation; Polymorphism
- **The relational model** supports very-high-level queries.
- Object-relational databases are an attempt to get the best of both.

The Impedance Mismatch

Complex Object Order mapped to multiple tables in a relational database



ID: 1001

customer: Ann

line items:

0321293533	2	$48	$96
0321601912	1	$39	$39
0131495054	1	$51	$51

payment details:

Card: Amex
CC Number: 12345
expiry: 04/2001

Can we store it all together as a complex object in the database?

Evolution of DBMS’s

- Object-oriented DBMS’s “failed” because they did not offer the efficiencies of well-entrenched relational DBMS’s.
    - Maintaining OODBMS difficult relative to RDBMS
    - Declarative languagelike SQL and big investment in RDBMS
    - Lack of standards compared to SQL standards
- Object-relational extensions to relational DBMS’s capture much of the advantages of OO, yet retain the relation (or row) as the fundamental abstraction.

Object-Relational Databases

- SQL-99 (or SQL:1999, or SQL3), is an attempt to extend the relational model to include many of the common object-oriented concepts.
  - This standard forms the basis for object-relational DBMSs
  - Available from essentially all the major vendors, although these vendors differ considerably in the implementation details

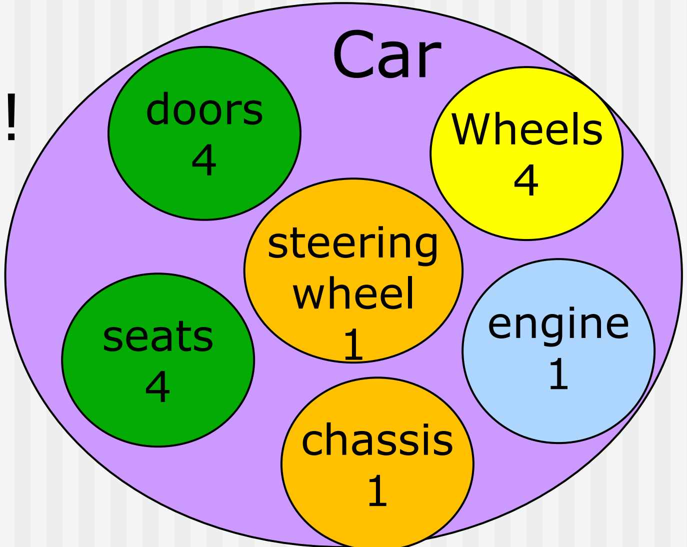
Some Object Relational Advantages

- **Objects Can Encapsulate Operations Along with Data**
  - An application can simply call the methods to retrieve the information.
- **Objects Are Efficient**
  - Object types and their methods are stored with the data in the database, so they are available for any application to use.
  - You can fetch and manipulate a set of related objects as a single unit.

Some Object Relational Advantages Continued

- **Objects Can Represent Part-Whole Relationships**
  - it is awkward to represent complex part-whole relationships.
  - E.g. Car Parts can be presented as objects that are attributes of Car the car object as opposed to complex primary-foreign Key relationships

A simple car!



Object Relational User Defined Types

- The concept of the relation is so central to SQL that objects in SQL keep relations as the core concept.
- A *user-defined type*, or UDT, is essentially a class definition, with a structure and methods.
- Two uses:
  1. As a *rowtype*, that is, the type of a relation (table).
  2. As the type of an attribute of a relation(table).
- In Oracle, first you need to define types as follows:

```
CREATE TYPE myUDT AS OBJECT
           ( list of attributes and methods );
```

Object Types and Object Tables **Exercise**

1. Create an object type to represent an **address**.
2. Next, create a “**staffMember”** object type that is made up of ID, staff_name (character) and contact address of **address** UDT.
3. Create an object table called “FullTimeEmployees” that stores “staffMember” objects

Oracles Object Type: An Example

```
CREATE OR REPLACE TYPE orderdetails AS OBJECT(
  pno number(5),
  qty integer,
  MEMBER FUNCTION  get_qty RETURN number,
  pragma RESTRICT_REFERENCES(get_qty,WNDS),
  MEMBER FUNCTION  get_cost RETURN number,
  pragma RESTRICT_REFERENCES(get_cost,WNDS));
```

What Oracle calls methods.

“Write no database state.”
That is, whatever “get_qty” method does
it won’t modify the database.

Method Definition: an example

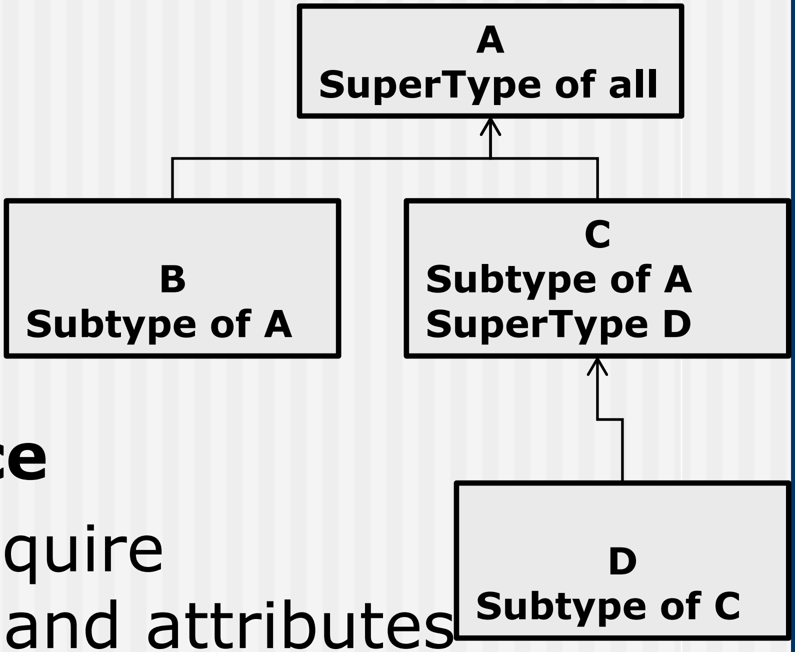
```
CREATE or REPLACE TYPE BODY orderdetails AS
MEMBER FUNCTION get_cost return NUMBER is
 p parts.price%type;
BEGIN
  SELECT price INTO p FROM parts
            WHERE pno = self.pno;
  return p * self.qty;
END;
END;
```

p is of datatype defined for column price in the parts table

- Member Function get_qty must also be defined in the body
- The member function or method is defined from order details UDT we have seen already
- Note: assumes a PARTS table with a price column exists

Types and Subtypes

- A TYPE hierarchy is a sort of family tree of object types
  - Consists of a parent base type called a **supertype** and
  - One or more levels of child object types call **subtypes**
  - Subtypes connected to Supertypes by **inheritance**
  - Subtypes automatically acquire their supertypes methods and attributes



Types and Subtypes – An example

- Let us create a person object that is a generalisation and student a specialisation of person
- We must use NOT FINAL and UNDER to allow inheritance

```
CREATE TYPE person as Object
(ID    INTEGER,
Name   VARCHAR2(30),
Phone  VARCHAR2(10),
MEMBER FUNCTION show RETURN
VARCHAR2)
NOT FINAL;
```

SUPERTYPE

Allows Inheritance

```
CREATE TYPE student UNDER person
(DeptID       INTEGER,
 major        VARCHAR2(30),
OVERRIDING MEMBER FUNCTION show RETURN VARCHAR2)
NOT FINAL;
```

SUBTYPE

Making Objects Persistent

- **Object Table (Row Object)**
  - Having declared a type, we may declare a table whose rows are of that type.

```
CREATE TABLE <tablename> OF <UDTName>;
```

  - i.e. a table of objects
- **A Column in a Relational Table (Column Object)**

```
CREATE TABLE <tablename> ( COL1 INTEGER,
                           COL2 <UDTName>);
```

- Note: Constraints such as **primary keys**, **foreign keys** may be **added now**, but apply only to this table and NOT to the UDT itself.

Making Objects Persistent Option – As a Row Object

- Simple Object type cust

```
 CREATE TYPE cust AS OBJECT
 (custId INTEGER,
  name VARCHAR2(30),
  contact_number VARCHAR2(10));
```

- An Object Table:

```
CREATE TABLE customer OF cust


INSERT INTO customer values (cust(123,'Sean','087-
   123456'));
INSERT INTO customer values (124, 'James', NULL);
INSERT INTO customer values (cust(125,'Mary',NULL));
```

--Objects in object tables are called row objects

Exercise

1. Create a object type called **custinfo** that has a customer attribute which is of **cust object type**, contact_address attribute using **address object type** (you have created these types earlier; see below)

2. Now create an object table called myCustomers based on the custinfo object type

3. Insert data into the table.

```
Object Type ADDRESS:
   ID             INTEGER,
   ADDRESS_LINE1 VARCHAR2(30),
   ADDRESS_LINE2 VARCHAR2(30),
   TOWN   VARCHAR2(30),
   COUNTY VARCHAR2(30),
   POST_CODE  VARCHAR2(10)
```

Making Objects Persistent as a Column Object

- As an attribute of a table (column objects)

```
CREATE TABLE orders_table
( order_id                  INTEGER,
  sales_rep                 VARCHAR2(40),
  cust_details            cust
);
```

“cust not a standard datatype but a user defined type UDT.

EXERCISE: Write an insert statement that populates the orders_table

Exercise

- Create a table called studentDetails that has a student attribute which is of **student UDT**, term_address, home_address using **address UDT** and a mentor that is a **person UDT**
- Insert data into the table.
- Note: we have previously created a UDT for "address" and "person" (see below) in the previous exercise.

```
Object TYPE person:
(ID      INTEGER,
Name     VARCHAR2(30),
Phone    VARCHAR2(10),
MEMBER FUNCTION show RETURN VARCHAR2)
```

Creating Constraints for Object Types(UDT’s)

```
CREATE TYPE person AS OBJECT (
id NUMBER,
name VARCHAR2(30) );
```

```
CREATE TYPE location AS OBJECT(
building_no NUMBER,
city VARCHAR2(40) );
```

Note: Cannot define Constraints on the Object Type Definitions. However, can create them on their omplementations as object tables or as columns in relational tables

Creating Constraints for Object Types(UDT’s)

On the dept table we want to create three constraints

1. Primary key on deptno
2. A default constraint on the dept_mgr column
3. A mandatory unique constraint on dept_loc

```
CREATE TYPE location AS OBJECT(
building_no NUMBER,
city VARCHAR2(40) );
```

```
CREATE TABLE dept (
deptno VARCHAR2(5) PRIMARY KEY,
dept_name VARCHAR2(20),
dept_mgr person  DEFAULT person(1,'Mr CEO') NOT NULL,
dept_loc  location,
CONSTRAINT dept_loc_cons1 UNIQUE (dept_loc.building_no, dept_loc.city),
CONSTRAINT dept_loc_cons2 CHECK (dept_loc.building IS NOT NULL) );
CONSTRAINT dept_loc_cons3 CHECK (dept_loc.city IS NOT NULL) );
```

Oracle Objects – A Reminder

- Oracle object types are **user-defined types** that make it possible to model real-world entities such as customers and purchase orders as objects in the database.
- Oracle object technology is a layer of abstraction built on Oracle relational technology.
- Underneath the object layer, data is still stored in columns and tables, but you are able to work with the data in terms of the real-world entities, such as customers and purchase orders, that make the data meaningful. Instead of thinking in terms of columns and tables when you query the database, you can simply select a customer.

Advantages of Objects

- In general, the object-type model is similar to the class mechanism found in C++ and Java. Removes the **Impedance Mismatch** problem
- No mapping layer is required between client-side objects and the relational database columns and tables that contain the data.
- Objects Can Encapsulate Operations Along with Data
- Objects Are Efficient
- All the advantages of an RDBMS e.g. Concurrency Control, Transaction Mgt, Security

Object Features – To Date

- Creating Object Types
- Objects as attributes of a table
- Object Tables
- Insertion of data into objects

How Objects are Stored in Tables -RECAP

- Objects can be stored in two types of tables:
  - Object tables: store only objects
    - In an object table, each row represents an object, which is referred to as a **row object**.
  - Relational tables: store objects with other table data
    - Objects that are stored as columns of a relational table, or are attributes of other objects, are called **column objects**.

RECAP Example 1

//Create a User Defined Type (Object Type)

```
CREATE TYPE person_typ AS OBJECT (
idno NUMBER,
name VARCHAR2(30),
phone VARCHAR2(20),
MEMBER FUNCTION get_idno RETURN NUMBER );
```

//Create an object table

```
CREATE TABLE person_obj_table OF person_typ;
```

//Insert a row into the object table – Relational Way

```
INSERT INTO person_obj_table VALUES ( 1, 'John
Smith', '1-800-555-1212');
```

RECAP Example 1 Continued

Insert a row into the object table – Object – Relational Way

```
 INSERT INTO person_obj_table VALUES
 (person_typ(2, 'Mary Murphy', '01-8318859'));
```

Select a row from the table relationally

```
SELECT * FROM person_obj_table p WHERE
p.Name = 'John Smith';
```

OR

```
SELECT idno,name,phone FROM person_obj_table p
WHERE p.name = 'John Smith';
```

Select an object from an object table

```
    SELECT VALUE(p) FROM person_obj_table p
    WHERE p.name =       'John Smith';
```

Support for Collection Types

- Oracle Supports 2 collection types:
  - **VARRAY** – and ordered collection of elements
  - **A Nested Table Type** – have any number of elements
- Choose VARRAY if have only fixed number of elements and order is important
- Choose nested table – have an arbitrary number of elements or performing mass inserts updates and deletes.

VARYING ARRAY COLLECTION TYPES (1)

- Ordered set of data elements, with each element having an index
- Set the **maximum number of elements** it can hold e.g.

```
CREATE TYPE email_list AS VARRAY(10) of VARCHAR2(80);
```

- Creating an array type does not allocate space
- It just **defines a datatype** that can be used as a
  - A datatype of a column in a relation table
  - An object type attribute
  - PL\SQL variable, function return value
- Normally **stored inline** in the row unless it is larger than 4000 bytes
- Otherwise, it is stored in a blob (binary large object)

VARYING ARRAY COLLECTION TYPES (2) An Example

- Creating a simple VARRAY

```
CREATE TYPE phones_varray_type AS VARRAY(3) OF CHAR(12);
```

- Using the VARRAY in an object type definition

```
CREATE TYPE person_type as object(
name varchar2(30),
city varchar2(40),
phones phones_varray_type);
```

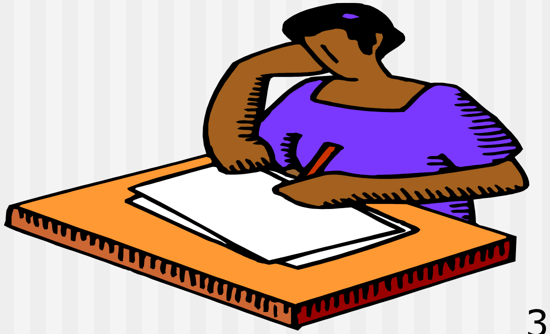
Remember this object type contains a VARRAY

- Making it Persistent in a relational table

```
CREATE TABLE employees(
eno NUMBER(4) PRIMARY KEY,
person person_type,
hire_date DATE);
```

Exercises

1. Write an INSERT statement for employees
2. Write a query that returns the phone numbers of an employee(s) that has a name sean
3. Create a VARRAY called **contacts** that allow for 100 contacts of person_type
4. Create a table called customers that has a CNO (number) and CNAME (String) and their set of contacts called CONTACTLIST (using the contacts Varray)
5. Insert a row into customers



Nested Table Collection Type (1)

- Nested Tables can support an **unlimited** number of entries per row (no maximum specified)
- An unordered set of data elements all of the same datatype
- Like Varrays it can be used as a
  - A datatype of a column of a relation table
  - An object type attribute
  - PL\SQL variable, function return value
- Elements of a nested table are stored in a **separate storage table**
- In effect, they are **tables within tables**

Nested table – Simple Example

//Create the nested table type

```
CREATE TYPE names AS TABLE OF VARCHAR2(30);
```

//create a table with the nested table in it

```
CREATE TABLE my_family(
1. family_name VARCHAR2(30),
2. names_list NAMES,
3. living INTEGER) NESTED TABLE names_list STORE AS
names_tab;
```

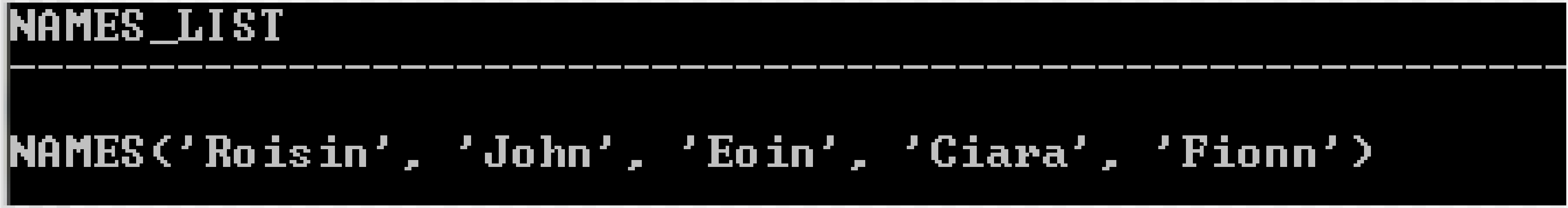
This is the column name that is to be stored as a nested table

Must give the nested table a name but you will not reference it again

//insert values

```
INSERT INTO my_family VALUES
('Murphy',names('Roisin','John','Eoin','Ciara','Fionn'), 5);
```

Output from select



```
NAMES_LIST
--------------------------------------------------------

NAMES('Roisin', 'John', 'Eoin', 'Ciara', 'Fionn')
```

Nested Tables Collection Type Using an User Defined Type (2)

- An Example Create a type to hold order details

```
CREATE TYPE odetails_type AS OBJECT
(       product_id    INTEGER,
        product_name        VARCHAR2(40),
        uom           VARCHAR2(10),
        QTY           INTEGER,
        UnitPrice     NUMBER(5,2));
```

Our nested table type. Just a definition!

Declaring the Nested table type

```
CREATE TYPE odetails_ntable_type AS TABLE OF
  odetails_type;
```

This can be included in another Object type!

```
CREATE OR REPLACE TYPE o_order_type AS OBJECT(
  ono          NUMBER(5),
  odetails     odetails_ntable_type,
  cno          NUMBER(5),
  shipped      DATE)
```

Can be used in an other object type definition!

Nested Tables in an OBJECT TABLE

- Using an Object Table

```
CREATE TABLE orders of o_order_type
(PRIMARY KEY(ono)
)NESTED TABLE odetails STORE AS odetails_tab;
```

External table

Note Nestings

```
INSERT INTO orders VALUES (
O_ORDER_TYPE(1020,
 odetails_ntable_type(odetails_type(1,'widget','kg',5,3.5),
                      odetails_type(3, 'milk','ltr',5, 1.5)),
1111,
'10-DEC-2004'
)
);
```

Nested Table as a column in a Relational Table

```
CREATE TABLE Order(
    ono NUMBER(5),
    odetails odetails_ntable_type,
    cno NUMBER(5),
    shipped DATE)
    NESTED TABLE odetails STORE AS od_nt
```

Nested table type

Declaring the storage for a nested table type

QUERYING COLLECTION OBJECTS

- Querying a collection column nests the elements of a collection in the result rows For example

```
SELECT o.odetails FROM orders o;
```

Result

```
ODETAILS(PRODUCT_ID, PRODUCT_NAME, UOM, QTY, UNITPRICE)
---------------------------------------------------------
ODETAILS_NTABLE_TYPE(ODETAILS_TYPE(1, 'widget', 'kg', 5, 3.5),
   ODETAILS_TYPE(3,'milk', 'ltr', 5, 1.5))
```

- Not all tools or applications are able to deal with results in a nested format
- If required in the conventional format you must “unnest” or “flatten” the collection attribute of a row into one or more relational rows

Flattening Results of Collection Queries

- Use the TABLE expression
  - A TABLE expression enables you to query a collection in the FROM clause like a table

```
SELECT od.*
FROM orders o, TABLE(o.odetails) od;
```

- A TABLE expression can also contain a subquery

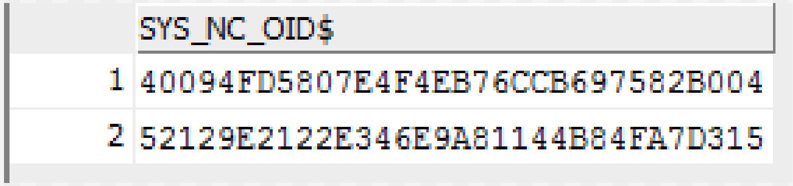
```
SELECT * FROM TABLE(SELECT o.odetails FROM orders o
 WHERE o.ono=1020);
```

  - Some restrictions
    - Subquery must return a collection type
    - SELECT List of the subquery must contain exactly one item
    - Subquery must only return only a single collection for one row. Else an error is produced

Object Identifiers: OID’s

- By default, every row object in an object table has an associated logical object identifier that uniquely identifies it in an object table
- Guaranteed to globally unique across your database environment
- Cannot directly access **OIDs** but you can make references to them and directly access them via **REF**
- Looking at OIDs

```
    SELECT SYS_NC_OID$ FROM person_obj_table;
```



	SYS_NC_OID$
1	40094FD5807E4F4EB76CCB697582B004
2	52129E2122E346E9A81144B84FA7D315

References -REFS

GO TO:
http://docs.oracle.com/cd/B28359_01/appdev.111/b28371/adobjint.htm#i458258

- **A REF** is a logical pointer to a row object that is constructed from the object identifier (OID)
- **REFs** use object identifiers (OIDs) to point to objects
  - Unlike object ID's, a REF is visible, although it is gibberish on viewing!.
  - By default, every row object has an associated logical object identifier (OID) that uniquely identifies it in an object table.
  - Globally unique OID 16 bytes long and automatically indexed
- REFs provide a more efficient means of expressing referential integrity than foreign keys, when the "one" side of the relationship is a row object.

Rules for Ref Columns and Attributes

GO TO:
http://docs.oracle.com/cd/B28359_01/appdev.111/b28371/adobjint.htm#i458258

- **REF** column is either constrained or unconstrained
- REF can be constrained by using a **SCOPE** constraint or **REFERENTIAL** constraint
- **REF** columns may contain object references that do not point to any existing object. – This is called **The dangling REF problem**
- **PRIMARY KEY** constraints cannot be specified for REF columns
- However, you can specify **NOT NULL** Constraints

Object-Identifiers Using References

- A user-defined type can also be used to specify the row types of a table:

```
      CREATE TYPE employee_type AS OBJECT
      ( name VARCHAR2(30),
       role          REF position_type );
CREATE TABLE employee_table OF employee_type;
```

ASSUMES:

position_type Object type and an Object table exists AS FOLLOWS

```
   CREATE TYPE position_type AS OBJECT
   ( position VARCHAR2(30),
     salary NUMBER(9,2),
     department VARCHAR2(30));

  CREATE TABLE position_table OF position_type;
```

Inserts using REF Function

- Insert some data

```
INSERT INTO position_table VALUES
  (position_type('accountant',100000.00, 'finance'));
```

- There is now a new row object in the position_table
- We now want to find its REF value to reference it in a row in the employee_table

```
INSERT INTO employee_table
            SELECT employee_type('Bob Jones', REF(p))
            FROM position_table p
            WHERE p.position = 'accountant';
```

- There is an alternative statement to carry out the INSERT

Queries using REF Function

- Return the Employee_Type Object

```
SELECT VALUE(e) FROM employee_table e WHERE
e.role.position= 'accountant' ;
```

```
Sample Output EMPLOYEE_TYPE('Bob Jones',
0000878546523 4A45F59876154854 7B45E)
```

  - Note the .notation in the where clause where we only object that have position “accountant” returned.

- Return the REF Value of the Object type

```
SELECT REF(e) FROM employee_table e
WHERE e.name= 'Bob Jones' ;
```

```
Sample output 00008785498 34B44E5932614585 47B23F
```

- Implicit Dereferencing using . Notation

```
SELECT e.role.salary FROM employee_table e WHERE
e.name='Bob Jones'-
```

```
sample output : John Smith
```

Explicit Dereferencing using DEREF

- **DEREF** function in a SQL statement returns the object instances corresponding to REF
- The object instance returned by DEREF may be of the declared type of the REF or any of its subtypes

```
SELECT e.name,DEREF(e.role)
FROM employee_table e
WHERE e.role IS NOT NULL;

Output
Bob Jones EMPLOYEE_TYPE(
                position_type('accountant',
                               100000,00, 'finance')
```

Using SCOPE IS

- Used to constrain the REF variable to point to a **particular table** containing an attribute of that object type.

```
CREATE TYPE address_type AS OBJECT
  (  street_address      VARCHAR2(40),
     postal_code           VARCHAR2(10),
     city                   VARCHAR2(30));

CREATE TABLE address_table OF address_type;

CREATE TABLE customer (
cust_id NUMBER,
name VARCHAR2(30),
address REF address_type  SCOPE IS address_table);
```

- NOTE If we delete the referenced object in address_table, we end up with a **dangling REF** in the customer table.

DANGLING REFS

Dangling REFs

- It is possible for the object identified by a REF to become unavailable—through either deletion of the object or a change in privileges.
  - … but it must be valid when it is stored
  - Such a REF is called *dangling*.
  - Oracle SQL provides a predicate (**called IS DANGLING**) to test for this condition.
  - Dangling REFs can be avoided by defining referential integrity constraints.

Using References to provide Referential Integrity

- A REF column constrained with a REFERENTIAL constraint similar to the specification for Foreign Keys

```
CREATE TABLE customer (
cust_id NUMBER,
name VARCHAR2(30),
address REF address_type  REFERENCES address_table);
```

- Assume a customer record references an address_type object in the address_table with a `street_address of ‘999 Tallaght’`
- If we now execute the following delete – what do think will occur?

```
 DELETE FROM address_table a
 WHERE a.street_address = ‘999 Tallaght’;
```

```
ERROR at line 1:
ORA-02292: integrity constraint (SEAN.SYS_C00361209) violated - child
    record found
```