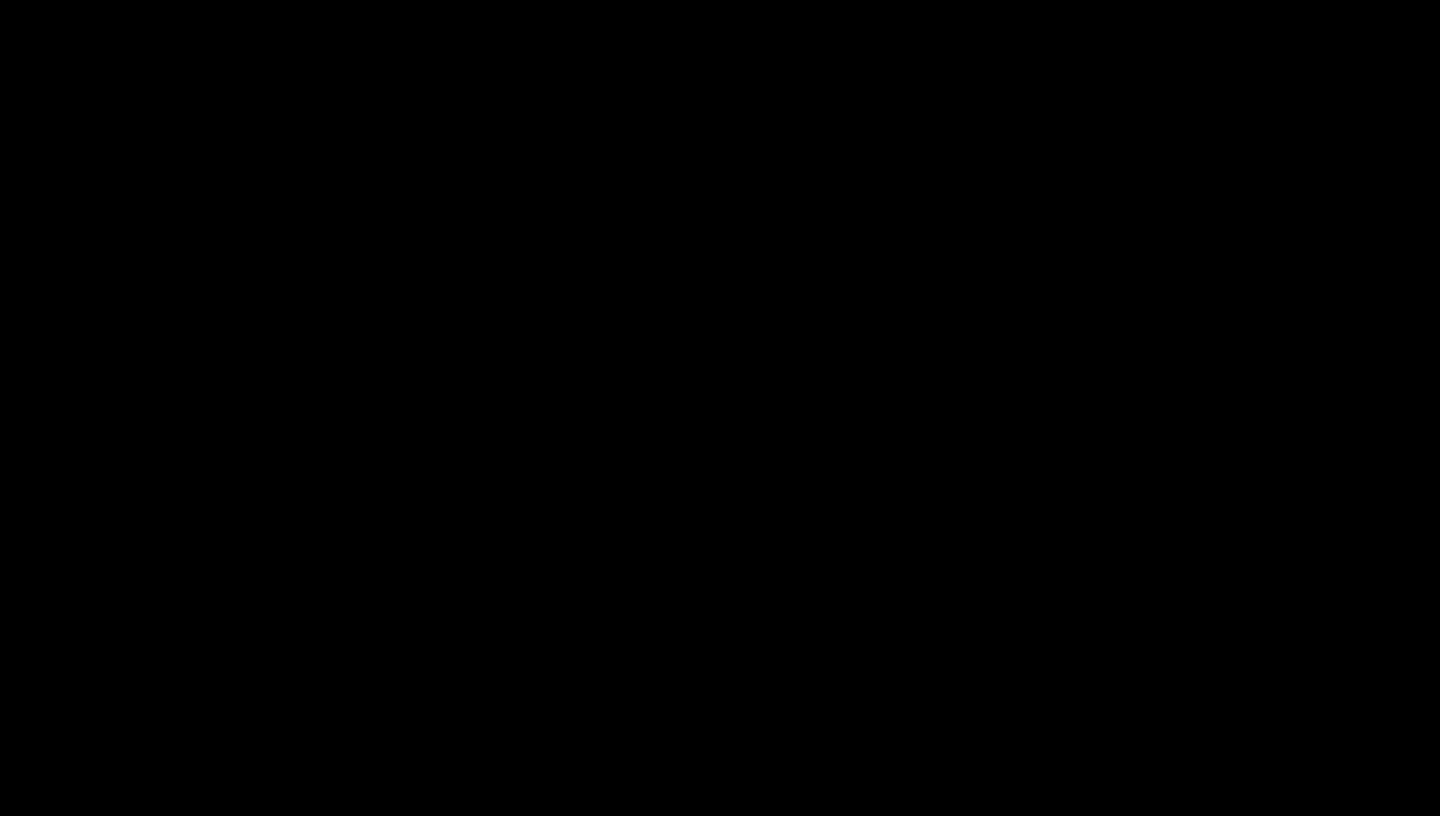
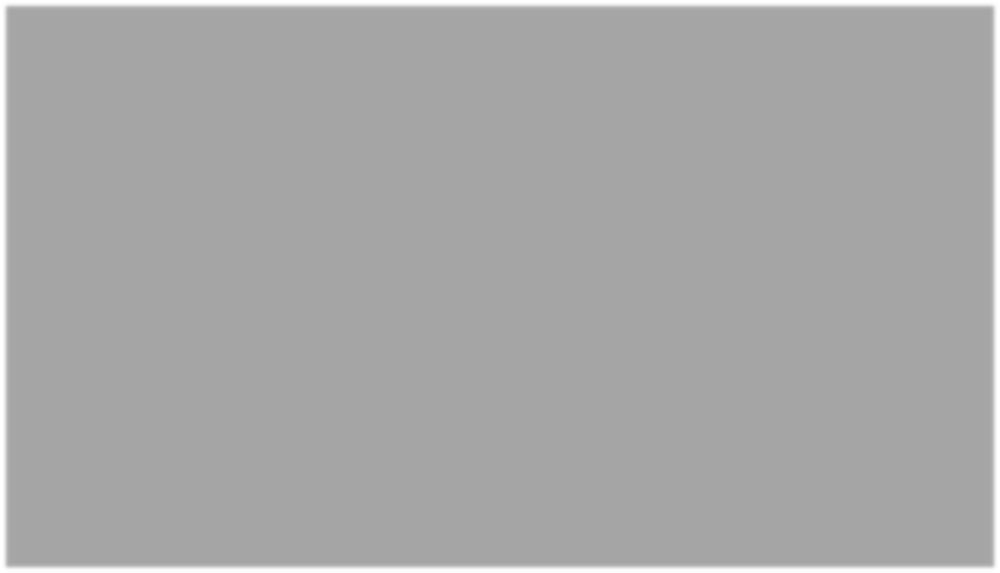
Implementation of Complex



Types in Oracle

**Abstract Data Type, Varray, Nested Tables, Methods, Inheritance, Reference, Overloading, Overriding, Object Views**

## Complex Data Types

* Relational database management systems (RDBMSs) are the standard tool for managing business data.
* They provide reliable access to huge amounts of data for millions of businesses around the world every day.
* Oracle is an **object-relational** database management system (ORDBMS), which means that users can define additional kinds of data--specifying both the structure of the data and the ways of operating on it--and use these types within the relational model.

## Complex Data Types

* This approach adds value to the data stored in a database.
* User-defined datatypes make it easier for application developers to work with complex data such as images, audio, and video.
* Object types store structured business data in its natural form and allow applications to retrieve it that way.
* For that reason, they work efficiently with applications developed using object-oriented programming techniques.

## Complex Data Types

* **Applications**
  + computer-aided design,
  + computer-aided software engineering
  + multimedia and image databases,
  + document/hypertext databases.
* **CLOB, BLOB**

## Complex Data Types

* Abstract Data Type
* **Varray**
* Nested Tables
* **Methods**
* Inheritance
* **Reference**
* Overloading, Overriding
* **Object Views**

## Large Objects (LOBs)

* Large Objects (LOBs) are a set of data types that are designed to hold large amounts of data.
* A LOB can hold up to a maximum size ranging from 8 terabytes to 128 terabytes depending on how your database is configured.
* Storing data in LOBs enables you to access and manipulate the data efficiently in your application.

Large Objects (LOBs)

* The built-in LOB data types BLOB, CLOB and NCLOB (stored internally), and BFILE (stored externally), can store large and unstructured data such as text, images and spatial data up to 4 gigabytes in size.
* BLOB
  + The BLOB data type stores binary large objects. BLOB can store up to 4 gigabytes of binary data.
* CLOB
  + The CLOB data type stores character large objects. CLOB can store up to 4 gigabytes of character data.
* NCLOB
  + The NCLOB data type stores character large objects in multibyte national character set. NCLOB can store up to 4 gigabytes of character data.
* BFILE
  + The BFILE data type enables access to binary file LOBs that are stored in file systems outside the Oracle database. A BFILE column stores a locator, which serves as a pointer to a binary file on the server's file system. The maximum file size supported is 4 gigabytes.

## Large Objects (LOBs)

* The **DBMS\_LOB** package provides subprograms to operate on BLOBs, CLOBs, NCLOBs, BFILEs, and temporary LOBs.
* You can use DBMS\_LOB to access and manipulate specific parts of a LOB or complete LOBs.
* DBMS\_LOB can read and modify BLOBs, CLOBs, and NCLOBs; it provides read-only operations for BFILEs.
* The bulk of the LOB operations are provided by this package.

## Large Objects (LOBs)

* SUBSTR functions
* This function returns a specified number of bytes (for a BLOB) or characters (for a CLOB or NCLOB), starting at a specified offset from the beginning of a specified LOB

## Large Objects (LOBs) - CLOB

* + Create the myClob table with a CLOB column

**CREATE TABLE myClob**

**(id NUMBER PRIMARY KEY,**

**clob\_content CLOB);**

## Large Objects (LOBs) - CLOB

* + Insert data into myClob Table

INSERT INTO myClob (ID,CLOB\_CONTENT)

**VALUES (101,to\_clob('hello'));**

INSERT INTO myClob (ID,CLOB\_CONTENT) VALUES(102,EMPTY\_CLOB());

**INSERT INTO myClob (ID,CLOB\_CONTENT) VALUES(105,’symca’);**

## Large Objects (LOBs) - CLOB

* view table myClob

**select id,DBMS\_LOB.SUBSTR(clob\_content) from myClob;**

* Update table myClob

**UPDATE myClob**

**SET CLOB\_CONTENT=TO\_CLOB('TYMCA') WHERE ID=101;**

## Large Objects (LOBs) - BLOB

* + Create the blob\_exm table with a BLOB column

**create table blob\_exm**

**(id int, blob\_content blob);**

* Insert data into blob\_exm Table

**insert into blob\_exm (id,blob\_content)**

**values(106,'1101111111');**

## Large Objects (LOBs) - BLOB

* Update table blob\_exm

**select id, DBMS\_LOB.SUBSTR(blob\_content) from blob\_exm;**

* Update table blob\_exm

**update blob\_exm**

**set blob\_content='1111010001' where id=104;**

## Large Objects (LOBs) - BFILE

* Create table using BFILE datatype

##### CREATE TABLE myTable

**(id number(4) PRIMARY KEY, bfile\_column BFILE NOT NULL);**

* Insert data into myTable Table

**insert into myTable(id,bfile\_column) values (1,BFILENAME('exm','test.txt'));**

Large Objects (LOBs) - BFILE

* BFILENAME() Function
* BFILENAME returns a BFILE locator that is associated with a physical LOB binary file on the

server file system.

* Syntax:

BFILENAME('directory', 'filename’)

* + 'directory' is a database object that serves as an alias for a full path name on the server file system where the files are actually located.
  + 'filename' is the name of the file in the server file system.
* You must create the directory object and associate a BFILE value with a physical file before you can use them as arguments to BFILENAME in a SQL or PL/SQL statement, DBMS\_LOB package, or OCI operation.

#### Collection Types

* A collection is an ordered group of elements having the same data type.
* Each element is identified by a unique subscript that represents its position in the collection.
* Collections are ideal for representing one-to-many relationships between data.
* Oracle supports two basic types of collections: VARRAY arrays and nested tables.
  + Nested table
  + Variable-size array or Varray

#### Variable-Sized Array (VARRAY)

* The VARRAY array is an ordered collection of data. Each element in this array has a specific index, which is used to access it.
* Items of type VARRAY are called *varrays*.
* They allow you to associate a single identifier with an entire collection.
* This association lets you manipulate the collection as a whole and reference individual elements easily.
* To reference an element, you use standard subscripting syntax
* A varray has a maximum size, which you must specify in its type definition.
* Its index has a fixed lower bound of 1 and an extensible upper bound.
* Thus, a varray can contain a varying number of elements, from zero (when empty) to the maximum specified in its type definition.

Variable-Sized Array (VARRAY)

* To create a VARRAY type, you use this syntax:

CREATE [OR REPLACE ] TYPE type\_name AS | IS

**VARRAY(max\_elements) OF element\_type [NOT NULL];**

#### Variable-Sized Array (VARRAY)

**Create VARRAY of phone\_no**

**CREATE OR REPLACE TYPE phone\_no AS VARRAY(10) OF VARCHAR2(128);**

**Create Table Phone\_list and data into it**

**CREATE TABLE phone\_list (id number, phone phone\_no);**

**INSERT INTO phone\_list VALUES (100,phone\_no('91','02352','224335'));**

## Variable-Sized Array (VARRAY)

###### Selecting elements from table

**SELECT p1.id, p2.column\_value**

**FROM phone\_list p1, TABLE(p1.phone) p2;**

**updating elements from table**

**update phone\_list**

**set phone=phone\_no('9422429442','23445’) where id=100;**

## Nested Tables

* Within the database, nested tables can be considered one-column database tables.
* Oracle stores the rows of a nested table in no particular order.
* But, when you retrieve the nested table into a PL/SQL variable, the rows are given consecutive subscripts starting at 1.
* That gives you array-like access to individual rows.
* PL/SQL nested tables are like one-dimensional arrays.
* You can model multi-dimensional arrays by creating nested tables whose elements are also nested tables.

## Nested Tables

* Nested tables differ from arrays in two important ways:
  + Arrays have a fixed upper bound, but nested tables are unbounded.
    - So, the size of a nested table can increase dynamically.
  + Initially, nested tables are dense, but they can become *sparse*

(have nonconsecutive subscripts).

* + - So, you can delete elements from a nested table using the built-in procedure DELETE.
    - That might leave gaps in the index, but the built-in function NEXT lets you iterate over any series of subscripts.

## Nested Tables

* To create a nested table, use following syntax

CREATE [OR REPLACE] TYPE nested\_table\_type AS TABLE OF element\_datatype [NOT NULL];

## Nested Tables

**Create Nested Table type CourseList**

**CREATE TYPE CourseList AS TABLE OF VARCHAR2(10);**

**Create Table Student**

**CREATE TABLE college**

**(id number,cname varchar2(10),course CourseList) NESTED TABLE course STORE AS CourseList\_Tab;**

## Nested Tables

###### insert into college

**values(1,’FAMT',CourseList('ADTA'));**

###### insert into college values(2,’GJC',CourseList('ADTA','GIS'));

* Selecting elements from table

**select c1.id,c2.column\_value**

**from college c1,TABLE(c1.course) c2;**

## Abstract Data Types

* Central to object-oriented programming in Oracle are abstract types, also called object types.
* Unlike conventional data types, abstract types contain not only the data structure but also the functions and procedures needed to manipulate them, combining data and behavior.
* **ADT** (Abstract DataType) is a user defined data type (also referred to as UDT's).
* Abstract Datatypes are data types that consist of one or more subtypes.
* Rather than being constrained to the standard Oracle data types of NUMBER, DATA, and VARCHAR2, abstract data types can more accurately describe your data.

## Abstract Data Types

* Object types are created by users and stored in a database like other Oracle data types, for example, VARCHAR2.
* The CREATE TYPE command allows you to create an abstract template corresponding to the real-world object.
* Syntax

**CREATE OR REPLACE TYPE type\_name AS OBJECT**

**(data\_type attribute, ...**

**MEMBER procedure or function signature [, PRAGMA clause], ...**

**);**

## Abstract Data Types

* Below there is the syntax for the object type implementation (in its basic version):
* The type body is created to define the member function, and the member method is invoked:

**CREATE TYPE BODY type\_name AS**

**MEMBER procedure or function implementation; ... END;**

## Abstract Data Types

* Below is an example of how to use this command:

**To Create Abstract datatype ADDRESS**

**CREATE OR REPLACE TYPE address AS OBJECT**

**( street char(20), city char(20), state char(2), zip char(5)**

**);**

## Abstract Data Types

**CREATE TABLE person**

**(**

**first\_name**

**last\_name full\_address**

**varchar(20),**

**varchar(20), address**

**);**

**Create Table person**

## Abstract Data Types

###### Insert data into Table person

**INSERT INTO person**

**VALUES ('Harshada','Salvi',address('41 Cherise Ave.', 'Ratangiri','MH','66654'));**

**INSERT INTO person**

**VALUES ('Rahul','Gupta',address('51 Tortor. Street', 'Ratangiri','MH','66654'));**

**INSERT INTO person**

**VALUES ('James','Massey',address('Viverra. Avenue', 'Surat','GJ','44454'));**

**INSERT INTO person**

**VALUES ('Paresh','Patel',address('Lodhi Road', 'Surat','GJ','44454'));**

**INSERT INTO person**

**VALUES ('Suresh','Sharma',address('Link Road', 'Mumbai','MH','66689'));**

## Abstract Data Types

###### Display data from Table person

**SELECT last\_name,p.full\_address.zip,p.full\_address.city FROM person p;**

* **Object Table**

## Abstract Data Types

– Object tables contain objects. Below is an example of how to create an object table:

**CREATE TYPE student AS object (name varchar2(30),**

**phone varchar2(20));**

**CREATE TABLE student\_table OF student;**

## Abstract Data Types

* There are no unambiguous (single-value) columns in object tables like in conventional Oracle tables: all columns are types and can therefore store multiple values.
* Object tables can be used to view data both as a single-column table and as a table with multiple columns consisting of object type components.
* The example below shows how you can insert data into an object table:

**INSERT INTO student\_table**

**VALUES ('James Massey', '1-800-555-4444’);**

### Abstract Data Types

**create type CustomerType as object( name VARCHAR2(10),**

**address VARCHAR2(10), dob date);**

**create a table whose rows are a user-defined type**

**create table CUST of CustomerType;**

### Abstract Data Types - Methods

###### Create Abstract datatype employee\_t with method

**CREATE TYPE employee\_t AS OBJECT (name VARCHAR2(30),**

###### ssn VARCHAR2(11),

**salary NUMBER,**

**MEMBER FUNCTION raise\_sal RETURN NUMBER);**

# Abstract Data Types - Methods

###### Create Method

CREATE TYPE BODY employee\_t AS

**MEMBER FUNCTION raise\_sal RETURN NUMBER IS BEGIN**

RETURN salary \* 2; END;

**END;**

# Abstract Data Types - Methods

**Create Table emp2 and insert record in ti**

**CREATE TABLE emp2 OF employee\_t;**

**INSERT INTO emp2**

**VALUES ( employee\_t('Frank', '12345', 1000) );**

# Abstract Data Types - Methods

**SELECT \* FROM emp2;**

**SELECT e.raise\_sal() from emp2 e;**

## Inheritance

* SQL object inheritance is based on a family tree of object types that forms a type hierarchy.
* The type hierarchy consists of a **parent object type, called a supertype**, and
* **one or more levels of child object types**, called **subtypes**, which are derived from the parent.

## Inheritance

* Inheritance is the mechanism that connects subtypes in a hierarchy to their supertypes.
* Subtypes automatically inherit the attributes and methods of their parent type.
* Also, the inheritance link remains alive.
* Subtypes automatically acquire any changes made to these attributes or methods in the parent: any attributes or methods updated in a supertype are updated in subtypes as well.
* Subtypes can have new attributes and new methods that its parent supertype does not have
* With object types in a type hierarchy, you can model an entity such as a employee, and also define different specializing subtypes of employees under the original type.
* You can then perform operations on a hierarchy and have each type implement and execute the operation in a special way.

## Inheritance

Employee

(eid,ename,emp\_addr, doj, basic, cal\_sal())

Engineer (eng\_typ())

Secretary (secretary\_type())

Technician (tech\_type())

## Inheritance

* + Creating a Parent or Supertype Object
    - You can create a parent or supertype object using the CREATE TYPE statement.
  + Creating a Subtype Object
    - A subtype inherits the attributes and methods of the supertype.
    - These are inherited:
      * All the attributes declared in or inherited by the supertype.
      * Any methods declared in or inherited by supertype.
    - Subtypes are created using the keyword UNDER as follows:

**CREATE TYPE eng UNDER emp\_typ**

## Inheritance

* + **FINAL and NOT FINAL Types and Methods for Inheritance**
* Object types can be inheritable and methods can be overridden if they are so defined.
* For an object type or method to be inheritable, the definition must specify that it is inheritable.
* For both types and methods, the keywords FINAL or NOT FINAL are used are used to determine inheritability.
  + Object type: For an object type to be inheritable, thus allowing subtypes to be derived from it, the object definition must specify this. NOT FINAL means subtypes can be derived. FINAL, (default) means that no subtypes can be derived from it.
  + Method: The definition must indicate whether or not it can be overridden. NOT FINAL (default) means the method can be overridden. FINAL means that subtypes cannot override it by providing their own implementation.

## Inheritance

Datatype: Super type Emp

create type emp as object

**(eid number,ename varchar2(10),emp\_addr varchar2(10),doj date,salary number,MEMBER FUNCTION cal\_sal RETURN number) not final;**

## Inheritance

* Create body of member function

CREATE TYPE BODY emp AS

**MEMBER FUNCTION cal\_sal RETURN number IS BEGIN**

RETURN salary; END;

**END;**

## Inheritance

* Create table employee of type emp

CREATE TABLE employee OF emp;

**INSERT INTO employee VALUES (emp(1,'Supriya','Ganpatipule','14-AUG-2008',20000) );**

## Inheritance

Display Data

select \* from employee;

**select e.cal\_sal() from employee e;**

## Inheritance

* Create subtype eng of emp data type

create type eng under emp (eng\_type varchar2(10),

**MEMBER FUNCTION emp\_type RETURN varchar2);**

## Inheritance

Create body of member function emp\_type

CREATE TYPE BODY eng AS

**MEMBER FUNCTION emp\_type RETURN varchar2 IS BEGIN**

RETURN 'Engineer'; END;

**END;**

## Inheritance

* Create table engineer of type eng

CREATE TABLE engineer OF eng;

**insert into engineer values (2,'anita','Ratnagiri','12-DEC-2009',15000,'senior’);**

## Inheritance

* Display Data

**select e.cal\_sal() from engineer e; select e.emp\_type() from engineer e;**

Inheritance

* Create Super type Musician

**create type Musician as object (mname varchar2(10),**

**dob date,**

**age number(4), addr varchar2(10), Type varchar2(20),**

**MEMBER FUNCTION artist\_type RETURN varchar2) not final;**

Inheritance

* Create Member Function Body

**CREATE TYPE BODY Musician AS**

**MEMBER FUNCTION artist\_type RETURN varchar2 IS BEGIN**

**RETURN 'Musician'; END;**

**END;**

## Inheritance

* Create table Musician\_tab and insert data into it.

**CREATE TABLE Musician\_tab OF Musician;**

**INSERT INTO Musician\_tab VALUES**

###### ( Musician('Harshada','14-AUG-1985',32,'Ratnagiri') );

**select m.artist\_type() from Musician\_tab m;**

## Inheritance

* Create subtype Vocals Under Musician

create type Vocals under Musician (vocal\_type varchar2(10),

**OVERRIDING MEMBER FUNCTION artist\_type RETURN varchar2) not final;**

## Inheritance

###### Define Body of overrinding member function

CREATE TYPE BODY Vocals AS

OVERRIDING MEMBER FUNCTION artist\_type RETURN varchar2 IS BEGIN

RETURN 'Vocals'; END;

END;

## Inheritance

CREATE TABLE Vocals\_tab OF Vocals;

**INSERT INTO Vocals\_tab VALUES**

( Vocals('Anita','14-AUG-1985', 32, 'Ratnagiri', 'Classical') );

**select v.artist\_type() from Vocals\_tab v;**

Object Identifiers & REF types

* Every row object in an object table has an associated logical object identifier (OID).
* Oracle assigns a unique system-generated identifier of length 16 bytes as the OID for each row object by default.
* The OID column of an object table is a hidden column.
* Although the OID value in itself is not very meaningful to an object-relational application, Oracle uses this value to construct object references to the row objects.
* You cannot directly access object identifiers, but you can make references (REFs) to the object identifiers and directly access the REFs

#### Object Identifiers & REF types

* Applications need to be concerned with only object references that are used for fetching and navigating objects.
* The purpose of the OID for a row object is to uniquely identify it in an object table.

## Object Identifiers & REF types

References to Row Objects

* A REF is a logical pointer or reference to a row object that you can construct from an object identifier (OID).
* You can use the REF to obtain, examine, or update the object. You can change a REF so that it points to a different object of the same object type hierarchy or assign it a null value.
* REFs are Oracle Database built-in data types. REFs and collections of REFs model associations among objects, particularly many-to-one relationships, thus reducing the need for foreign keys. REFs provide an easy mechanism for navigating between objects.

## Object Identifiers & REF types

* Scoped REF types require less storage space and allow more efficient access than unscoped REF types.
* You can constrain a column type, collection element, or object type attribute to reference a specified object table. Use the SQL constraint subclause SCOPE IS when you declare the REF.

## Object Identifiers & REF types

* Create object theater\_t

**create type theater\_t as object (tno number(4),**

**tname varchar2(20), address varchar2(20), phone number(10)) ;**

Object Identifiers & REF types

* Create Table Theaters

**create table Theaters of theater\_t ;**

**Insert into Theaters values(1,'CityPride','KCNagar',223344); Insert into Theaters values(2, 'IMAX','MGRoad',223344);**

## Object Identifiers & REF types

CREATE TABLE Nowshowing (movie\_id number(4),

**theater REF theater\_t SCOPE is Theaters, mstart date,mend date);**

## Object Identifiers & REF types

**insert into Nowshowing values (1,**

**(SELECT REF(t) FROM Theaters t WHERE t.tno = 1),**

**'06-apr-2016','12-apr-2016’);**

**select n.movie\_id,n.theater.tname from Nowshowing n;**

