

Introduction to Oracle 9i Objects

PROCASE Consulting

Mar 23, 2003

Garry Chan

Managing Partner

gchan@ProcaseConsulting.com

Object Types

- **User-defined datatypes**

- A grouping of attributes normally based on built-in datatypes
 - Can be based on other object types (called composite object types)

- **Syntax:**

```
CREATE [OR REPLACE] TYPE object_type_name AS OBJECT
(attribute_name datatype
,attribute_name datatype);
/
```

- **Example: object type that models an address**

```
CREATE OR REPLACE TYPE address_type AS OBJECT
(street      VARCHAR2(25)
,city        VARCHAR2(25)
,country     VARCHAR2(20));
/
```

- Both semicolon and slash are required to create an object

Object Methods

- **PL/SQL functions that can operate on the attributes of an object type**
 - Also called member methods
- **Constructor method is created automatically whenever an object type is created**
 - Internal method that defines how the object is created
 - Cannot be changed
 - Has the same name as the object type
 - Has one parameter for each attribute of the object type
- **Oracle also allows user-defined methods**
 - Creating member methods is a two-step process (like packages)
 1. Create object type definition, which includes the method specification
 - Declares the method type, name, parameters, and return type
 2. Create object type body, which includes the method code

Object Columns

- **Columns in relational tables based on object types**
 - Each instance of an object column is called a column object
- **A relational table can have multiple object columns**
- **Created like regular columns, except that the column definition is based on the object type rather than a built-in datatype**

```
column_name    object_type [NOT NULL]
```

- **Example:**

- Create a site table with site_id as NUMBER(2) and address as address_type

```
CREATE TABLE site
(site_id          NUMBER(2)          NOT NULL
,address         address_type);
```

Inserting Into Object Columns

- **Constructor method is used to assign values to an object column**
 - Each time a value is inserted into an object column, Oracle creates an object
 - Conceptually, each row in the `site` table has an entire object stored in the address column
- **Insert `site_id` and address into `site` table; `classroom` and `location` are unknown**

```
INSERT INTO site (site_id, address)
VALUES (1, address_type('1 Fun Way', 'YORK', 'UK'));
```

Object Tables

- **Object table is a table based on an object type**
 - Each row in the table is called a row object
 - Each column of the table corresponds to an attribute of the object type
- **Created by using the `CREATE TABLE` command with the `OF` clause**

```
CREATE TABLE table_name OF object_type;
```
- **Example:**

```
CREATE TABLE person_address OF address_type;
```
- **Object Identifier (OID) is a unique identifier created for each row object in an object table**
 - Guaranteed to be unique within the database
 - OIDs are not reused even after the table is dropped
 - OIDs are not created for object columns

Inserting Into Object Tables

- Data can be inserted into object tables using either the standard syntax or the constructor method

- **Examples:**

- Use a standard insert to add a row

```
INSERT INTO person_address  
VALUES ('101 Fun St', 'DETROIT', 'USA');
```

- Use the constructor method to insert a row

```
INSERT INTO person_address  
VALUES (address_type('101 More Fun St', 'NEW ORLEANS', NULL));
```

REF Columns

- **OID pointer to an object in an object table**

- This is similar to storing a column in a child table to point to a row in a parent table

- Defined using a new datatype called REF

```
column_name REF object_type [SCOPE IS object_table_name]
```

- **REF defines the object type**

- SCOPE is an optional clause, which limits the reference to a specific object table

- **Example: Add an address column to the instructor table that references address_type objects**

- Limit addresses to those that exist in the person_address table

```
ALTER TABLE instructor
```

```
ADD (address REF address_type SCOPE IS person_address);
```

Inserting Into REF Columns

- **A subquery must be used to insert a single row into a table that contains REF columns**
 - Subquery is used to retrieve a reference to a specific row object
 - Values for remaining columns can be included in the subquery
- **For example, insert JONES of 1 Fun Way into the instructor table**

```
INSERT INTO instructor
    (instructor_id, instructor_name, address)
SELECT 1321, 'JONES', REF(pa)
FROM person_address pa
WHERE street = '1 Fun Way';
```

Selecting Column Objects

- Column objects are retrieved the same way as normal columns
- For example, retrieve `site_ids` and addresses from the `site` table

```
SELECT site_id, address
FROM site;
```

```
SITE_ID          ADDRESS(STREET, CITY, COUNTRY)
-----          -
2              ADDRESS_TYPE('1600 Pennsylvania', 'Washington', 'USA')
3              ADDRESS_TYPE('22 Trafalgar Square', 'London', 'UK')
```

- Results show that `address` is an `address_type` object composed of `street`, `city`, and `country`

Selecting Column Object Attributes

- **Column object attributes are retrieved using the dot notation**
 - Syntax: `table_alias.column_name.attribute_name`
 - The column must be prefixed by a table alias
 - Using table name as a prefix will result in an error
- **For example, retrieve streets and cities from the `site` table**

```
SELECT site_id, s.address.street, s.address.city
FROM site s;
```

SITE_ID	ADDRESS.STREET	ADDRESS.CITY
2	1600 Pennsylvania	WASHINGTON
3	22 Trafalgar Square	LONDON

Selecting Row Objects

- The `VALUE` function has been added to Oracle8 to retrieve objects
 - Syntax:
`VALUE (table_alias)`
- Use the `VALUE` function to retrieve object values from the `person_address` table

```
SELECT VALUE(pa)
FROM person_address pa;
```

```
PERSON_ADDRESS(STREET, CITY, COUNTRY)
```

```
-----
```

```
ADDRESS_TYPE('101 Fun St', 'DETROIT', 'USA')
```

```
ADDRESS_TYPE('101 More Fun St', 'NEW ORLEANS', NULL)
```

Selecting Row Object Attributes

- Row object attributes are retrieved the same way as relational columns
- For example, retrieve addresses from the `person_address` table

```
SELECT *  
FROM person_address;
```

STREET	CITY	COUNTRY
25 MISSION WAY	SAN DIEGO	USA
5567 KNIGHTS COURT	LONDON	UK

- For example, retrieve streets and cities from the `person_address` table

```
SELECT street, city  
FROM person_address;
```

STREET	CITY
25 MISSION WAY	SAN DIEGO
5567 KNIGHTS COURT	LONDON

Selecting REF Columns

- **REF columns can be retrieved the same way as regular columns**
 - However, it is not very useful to users because it is a pointer
- **Retrieve instructor_ids and addresses from the instructor table**
 - address is a REF column in the instructor table

```
SELECT instructor_id, address
FROM instructor;
```

INSTRUCTOR_ID	ADDRESS
222	000022020889608A47F49611D087CD00A024B12E9789608A21
243	000022020889608A46F49611D087CD00A024B12E9789608A21

Selecting REF Columns: Deref Function

- **Deref** is a function that traverses the REF pointer to return the row object
 - Syntax:
`Deref(REF column)`
- For example, list instructors and their complete addresses

```
SELECT instructor_id  
       ,Deref(address)  
FROM instructor;
```

```
INSTRUCTOR_ID  Deref(ADDRESS)(STREET, CITY, COUNTRY)  
-----  
263           ADDRESS_TYPE('562 HOCKEY STREET', 'TORONTO', 'CANADA')  
628           ADDRESS_TYPE('5567 KNIGHTSBRIDGE COURT', 'LONDON', 'UK')
```

Selecting REF Attributes

- **Attributes of a row object corresponding to a REF datatype can be accessed using the dot notation**
 - Same as column objects
- **For example, list instructors and their streets and cities**

```
SELECT instructor_id
       ,i.address.street street
       ,i.address.country country
FROM instructor i;
```

INSTRUCTOR_ID	STREET	COUNTRY
263	562 HOCKEY STREET	CANADA
628	5567 KNIGHTS COURT	UK

Updating Object Columns

- **Update column objects using the constructor method**
- **For example, change the address for site 1**

```
UPDATE site
SET address = address_type('1 Fun Way', 'LONDON', 'UK')
WHERE site_id = 1;
```

- **Update column object attributes using the dot notation**

```
SET object_column.attribute_name=expression
```

- **For example, change the city for site 1 to “WASHINGTON”**

```
UPDATE site s
SET s.address.city = 'WASHINGTON'
WHERE s.site_id = 1;
```

Updating Object Tables

- **Object tables can be updated in the same way as regular tables**
 - For example, change country from USA to US

```
UPDATE person_address
SET country = 'US'
WHERE country = 'USA';
```

- **Object tables can also be updated using a constructor**
 - For example, nullify the street for all London addresses

```
UPDATE person_address pa
SET pa = address_type(NULL, 'LONDON', 'UK')
WHERE city = 'LONDON';
```

Updating REF Columns

- **A subquery must be used to update a single row in a table that contains REF columns**
 - Subquery is used to retrieve the reference to a specific row object
`REF(table_alias)`
 - Values for remaining columns can be included in the subquery
- **For example, update JONES address in the instructor table**

```
UPDATE instructor
SET  address =
      (SELECT REF(pa)
       FROM person_address pa
       WHERE street = '1 Better Way'
       AND city = 'TORONTO')
WHERE instructor_name = 'JONES';
```

Deleting Objects

- **Delete all Boston sites (object column)**

```
DELETE FROM site s
WHERE s.address.city = 'BOSTON';
```

- **Delete 1 Fun Way from person_address table (object table)**

```
DELETE FROM person_address
WHERE street = '1 Fun Way';
```

- **Delete all students from Boston, USA (REF column)**

```
DELETE FROM student s
WHERE s.address.city = 'BOSTON'
AND s.address.country = 'USA';
```

Altering Object Types

- **With Oracle9i, you can add, modify, or drop object type attributes even if they are referenced in existing tables**
- **Syntax:**

```
ALTER TYPE object_type ADD | MODIFY ATTRIBUTE  
    (attribute_name datatype, ...,attribute_name datatype)  
[CASCADE [[NOT] INCLUDING TABLE DATA]];
```

```
ALTER TYPE object_type DROP ATTRIBUTE  
    (attribute_name, ...,attribute_name)  
[CASCADE [[NOT] INCLUDING TABLE DATA]];
```

Altering Object Types (continued)

- **Rules**

- `CASCADE` clause is required to propagate the changes if any dependent types or tables exist
 - Otherwise Oracle will return an error when dependents exist
- `INCLUDING TABLE DATA` clause is used to convert data stored in all impacted objects
 - For `ADD`, the new attribute is added to each object as `NULL`
 - For `DROP`, the attribute data is removed from each object
- `NOT INCLUDING TABLE DATA` clause defers the data changes until the objects are accessed
- `MODIFY` clause is used to modify the datatype of existing attributes

Altering Object Types Examples

- **Add zip to the address_type**

```
ALTER TYPE address_type ADD ATTRIBUTE  
    (zip NUMBER(5))  
CASCADE;
```

- **Increase size of zip to 10 digits**

```
ALTER TYPE address_type MODIFY ATTRIBUTE  
    (zip NUMBER(10))  
CASCADE INCLUDING TABLE DATA;
```

- **Remove the street attribute**

```
ALTER TYPE address_type DROP ATTRIBUTE street  
CASCADE NOT INCLUDING TABLE DATA;
```

Dropping Object Types

- **Once an object type has been defined in the database, it remains there until it is explicitly dropped**
 - A type cannot be dropped if it is used in a table
- **DROP TYPE command is used to remove object types**
 - Syntax:

```
DROP TYPE object_type_name;
```
- **Example:**

```
DROP TYPE address_type;
```

VARRAYs

- **Variable-size arrays**

- Similar to PL/SQL tables but can be stored in the database
- The number of elements in an array defines its size
 - Elements are positional
- Used for small sets (cannot be indexed)

- **Created in two steps**

1. Define a VARRAY datatype

```
CREATE TYPE varray_type_name AS VARRAY(size) OF datatype;
```

- size parameter specifies the maximum number of elements

2. Define a column based on this datatype

- All elements of a VARRAY are stored in a single column

- **Example: Create a four-element VARRAY to hold instructor phone numbers**

- Positions represent the following (1: Office, 2: Cell, 3: Pager, 4: Home)

```
CREATE TYPE phone_varray_type AS VARRAY(4) OF VARCHAR2(15);
```

```
/
```

```
ALTER TABLE instructor ADD (tel_no phone_varray_type);
```

Inserting Into a VARRAY

- **Use the constructor method**

- Can define up to the maximum number of elements
 - Undefined elements can be set to null
- Example: Add a new instructor, Smith, and specify her office and pager phone numbers

```
INSERT INTO instructor (instructor_id, instructor_name, tel_no)
VALUES (12, 'SMITH'
       ,phone_varray_type('(609)345-1234', null, '(609)345-1444'));
```

- **A subquery can also be used to insert into a VARRAY**

- Query must return the same VARRAY type
- Example: Insert instructor Jones, copy the telephone numbers from Smith

```
INSERT INTO instructor (instructor_id, instructor_name, tel_no)
SELECT 13, 'JONES', tel_no
FROM instructor
WHERE instructor_id = 12;
```

Selecting From a VARRAY

- **VARRAY is treated as a single unit, collection, in SQL**
 - The result is a VARRAY constructor
- **For example, query Smith's phone numbers**

```
SELECT instructor_name, tel_no
FROM instructor
WHERE instructor_id = 12;
```

```
INSTRUCTOR_NAME          TEL_NO
-----
SMITH                    PHONE_VARRAY_TYPE(' (609)345-1234', NULL, ' (609)345-1444')
```

INSTRUCTOR_NAME	TEL_NO
SMITH	(609)345-1234 (609)345-1444
JONES	(712)555-1234 (712)555-4321 (712)555-2224 (712)555-7686

Creating Object Type VARRAYS

- VARRAYS can be defined based on object types
- For example, create a VARRAY type based on `classroom_type`
 - `classroom_type` contains `room_number`, `capacity`, and `description`

```
CREATE TYPE classroom_varray_type AS VARRAY(50)
OF classroom_type;
/
```

- Create the `site` table with a classroom VARRAY column

```
CREATE TABLE site
(site_id          NUMBER(2) NOT NULL
, address        address_type
, classroom      classroom_varray_type
, location       VARCHAR2(12));
```

Inserting Into Object Type VARRAYS

- **Insert data into `classroom` table by nesting constructor methods**
 - Nesting constructor methods is similar to nesting functions
 - Oracle constructs the inner type first, then the outer type
- **For example, insert a new site with two classrooms**

```
INSERT INTO site (site_id, classroom)
VALUES (8, classroom_varray_type(
    classroom_type(1, 10, 'Small Room')
    ,classroom_type(2, 30, 'Large Room')));
```

SITE_ID	CLASSROOM		
1	1	2	3
	30	30	20
	Maple Room	Birch Room	Oak Room
8	1	2	
	10	30	
	Small Room	Large Room	

Unnested Queries

- Used to access individual varray elements
- Syntax:

```
SELECT ...  
FROM main_table main_alias  
      ,TABLE(main_alias.varray_column_name) varray_table_alias  
WHERE ...
```

- Rules:
 - TABLE keyword is used with the varray column name to “join” to the varray
 - Aliases must be used to identify the main table and the varray
 - WHERE clause does not require a join condition but can be used to filter data

Unnested Query Example

- Display sites 1 to 5 with room numbers and capacities

```
SELECT s.site_id, c.room_number, c.capacity
FROM site s
      ,TABLE(s.classroom) c
WHERE s.site_id <= 5
ORDER BY s.site_id, c.room_number;
```

SITE_ID	ROOM_NUMBER	CAPACITY
1	1	20
1	2	24
1	3	30
2	1	20
2	2	24
3	1	20
3	2	24
3	3	24
4	1	20
5	1	20
5	2	20
5	3	20

Nested Tables

- **Unordered set of nested data**

- Similar to `VARRAYS` with the following differences

- Elements are not positional
- There is no maximum for the number of elements
- Used for large data sets
- Elements can be indexed

- **Created in two steps**

1. Define a nested table datatype

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

2. Define a column based on this datatype

```
CREATE TABLE parent_table  
(  
    ...  
    ,nested_table_column nested_table_type)  
NESTED TABLE nested_table_column  
STORE AS nested_table_name;
```

Nested Table Example

- **Create a nested table to store invoice items**
 1. Create nested table type based on `invoice_item_type` object type, which is made up of `student_lname`, `course_title`, `start_date`, `end_date`, `amount`

```
CREATE TYPE invoice_item_table_type AS TABLE
OF invoice_item_type;
/
```

2. Create an invoice table with `invoice_item` column using `invoice_item_table` datatype

```
CREATE TABLE invoice
(invoice_number NUMBER(6) NOT NULL
,company_id NUMBER(2) NOT NULL
,billing_date DATE NOT NULL
,due_date DATE
,invoice_item invoice_item_table_type)
NESTED TABLE invoice_item
STORE AS invoice_item_table;
```

Relationship Between Parent Table and Nested Table

- **The parent table contains a reference to the data in the nested table**
 - Similar to a parent-child relationship between two relational tables
- **In relational tables, a foreign key enforces the existence of a parent record before a child record**
 - However, a nested table establishes and enforces this relationship implicitly
 - Child records are deleted whenever a parent record is removed

Inserting Into Nested Tables

- **Use the constructor method to insert one row**
 - Constructor is the name of the nested table type
 - Also use the object constructor if the table is based on an object type
- **For example, insert an invoice with two invoice items**
 - One row is inserted into the `invoice` table and two rows are inserted into the `invoice_item_table`

```
INSERT INTO invoice
(invoice_number, company_id, billing_date, invoice_item)
VALUES (10007, 30, '01-aug-1997',
       invoice_item_table_type(
         invoice_item_type('GREEN', 'ORACLE', '15-JUL-1997', null, 1250)
         , invoice_item_type('ADAMS', 'UNIX', '08-JUL-1997', null, 1250)));
```

Selecting From Nested Table Columns

- Retrieve nested table columns in the same way as regular columns
- The value of nested table columns cannot be compared
- For example, select invoice items for invoice 10007

```
SELECT invoice_item  
FROM invoice  
WHERE invoice_number = 10007;
```

```
INVOICE_ITEM(STUDENT_LNAME, COURSE_TITLE, START_DATE, END_DATE, AMOUNT)  
-----  
invoice_item_table_type(  
  invoice_item_type('GREEN', 'ORACLE', '15-JUL-1997', null, 1250)  
, invoice_item_type('ADAMS', 'UNIX', '08-JUL-1997', null, 1250))
```

Unnested Queries for Nested Tables

- Unnested query can be used to select from nested tables
- For example, display invoice number and last name for C++ course

```
SELECT i.invoice_number, ii.student_lname
FROM invoice i
      ,TABLE(i.invoice_item) ii
WHERE ii.course_title = 'C++';
```

```
INVOICE_NUMBER  STUDENT_LNAME
-----
                10000 ROSE
                10000 MYERS
                10001 BROWN
                10001 TYLER
                10001 CARTER
                10001 DUDLEY
                10002 ADAMS
                10004 CRICK
```

Inserts Using Unnested Queries

- Use an unnested query to add rows to a nested table

— Syntax:

```
INSERT INTO TABLE (subquery)
VALUES (expression,...,expression) | subquery;
```

- For example, add another invoice item to invoice 10007

```
INSERT INTO TABLE
  (SELECT invoice_item
   FROM invoice
   WHERE invoice_number = 10007)
VALUES('ADAMS', 'ORACLE', '15-JUL-1997', '18-JUL-1997', 1250);
```

Updates Using Unnested Queries

- Use an unnested query to update rows in a nested table

— Syntax:

```
UPDATE TABLE (subquery)
SET column_name = expression
[WHERE column_name = expression];
```

- For example, update fees for Oracle courses to 1000 on invoice 10008

```
UPDATE TABLE
  (SELECT invoice_item
   FROM   invoice
   WHERE  invoice_number = 10008)
SET amount = 1000
WHERE course_title = 'ORACLE';
```

Deletes Using Unnested Queries

- Use an unnested query to delete rows in a nested table

— Syntax:

```
DELETE FROM TABLE (subquery)
[WHERE column_name=expression];
```

- For example, delete all Oracle courses from invoice 10008

```
DELETE FROM TABLE
  (SELECT invoice_item
   FROM invoice
   WHERE invoice_number = 10008)
WHERE course_title = 'ORACLE';
```

Example: Using Objects in PL/SQL

- Change the address of our Boston office

```
DECLARE
  -- 1. Declare an object variable
  tmp_address address_type;
BEGIN
  -- 2. Retrieve the entire address for Boston
  SELECT s.address
  INTO tmp_address
  FROM site s
  WHERE s.address.city = 'BOSTON';

  -- 3. Replace the street attribute in the variable
  tmp_address.street := '1234 Main Street';

  -- 4. Update the entire object in the database
  UPDATE site s
  SET address = tmp_address
  WHERE s.address.city = 'BOSTON';
END;
```

Accessing VARRAY Elements in PL/SQL

- **The elements of a VARRAY can be accessed using a subscript**
 - Syntax:
`VARRAY_name(subscript)`
- **Each attribute of a VARRAY element can be accessed by adding the dot notation**
 - Syntax:
`VARRAY_name(subscript).attribute_name`
- **For example, change the capacity of the first classroom to 30**
`classroom_vry(1).capacity := 30;`

VARRAY Methods

- **A method is a built-in function or procedure that operates on collections**
 - Methods help generalize code and make VARRAYs easier to use
- **A method is invoked using a dot notation**
 - Can be called from procedural statements but not from SQL statements
`VARRAY_name.method_name[(parameters)]`
- **Commonly used VARRAY methods**

Method	Type	Description
EXISTS (n)	Func	Returns TRUE if the nth element in a VARRAY exists
COUNT	Func	Returns the number of elements in a VARRAY
LIMIT	Func	Returns the maximum number of elements that a VARRAY can contain
EXTEND [(n)]	Proc	Allocates n null elements in a VARRAY, 1 by default
DELETE	Proc	Removes all elements from the collection

Example: Using VARRAYs in PL/SQL

- Increase the capacity of each classroom in London by 20 percent

```
DECLARE
    -- 1. Declare a VARRAY variable
    classroom_vry classroom_VARRAY_type;
BEGIN

    -- 2. Retrieve all London classrooms into the VARRAY
    SELECT classroom
    INTO classroom_vry
    FROM site st
    WHERE st.address.city = 'LONDON';

    -- 3. Loop through each classroom and increment capacity
    -- Use the COUNT method as the upper limit
    FOR ctr IN 1..classroom_vry.COUNT LOOP
        classroom_vry(ctr).capacity :=
            classroom_vry(ctr).capacity * 1.2;
    END LOOP;

    -- 4. Update the entire object in the database
    UPDATE site st
    SET classroom = classroom_vry
    WHERE st.address.city = 'LONDON';

END;
```

Example: Using Nested Tables in PL/SQL

- **Update invoice_amount with the total amount for each invoice**

```
DECLARE
  -- 1. Declare a variable to store the total amount
  inv_amount NUMBER(7,2);

  -- 2. Declare a cursor to retrieve all invoices with details
  CURSOR inv_cur IS
  SELECT invoice_number, invoice_item
  FROM invoice
  FOR UPDATE;
BEGIN
  -- 3. Outer loop for each invoice
  FOR inv_rec IN inv_cur LOOP
    inv_amount := 0;

    -- 4. Inner loop to sum amount for each detail
    FOR i IN 1.. inv_rec.invoice_item.COUNT LOOP
      inv_amount := inv_amount + inv_rec.invoice_item(i).amount;
    END LOOP;

    -- 5. One update per invoice
    UPDATE invoice
    SET invoice_amount = inv_amount
    WHERE CURRENT OF inv_cur;
  END LOOP;
END;
```