Composite Data Types

Oracle Database Server supports the following composite data types:

- Are of two types:
  - PL/SQL RECORDs
  - PL/SQL Collections
    - INDEX BY Table
    - Nested Table
    - VARRAY
- Contain internal components
- Are reusable

A composite type contains components within it. A variable of a composite type contains one or more scalar components.

PL/SQL Records

- Must contain one or more components of any scalar, RECORD, or INDEX BY table data type, called FIELDs.
- Are similar in STRUCTUREs (C/C++) to records in a 3GL
- Are NOT the SAME AS ROWs in a database table
- Treat a COLLECTION OF FIELDs as a logical unit
- Are convenient for fetching a row of data from a table for processing.

Create a PL/SQL Record

Syntax:

```
-- Defining the RECORD
TYPE type_name IS RECORD
  (field_declaration[, field_declaration]…);

-- Declaring the RECORD
identifier type_name;
```

Where field_declaration is: (similar to variable declartion)

```
field_name {field_type | variable%TYPE
   | table.column%TYPE | table%ROWTYPE} [[NOT NULL] {:= | DEFAULT} expr]
```

Example:
Declare variables to store the first name, job, and salary of an employee.

```
TYPE empRecType IS RECORD
(
      first_name  VARCHAR2(20),
      job         employees.job_id%TYPE,
      salary            employees.salary%TYPE
);

empRec            empRecType;
```

PL/SQL Tables

A PL/SQL table is:

- a ONE DIMENSIONAL, unbounded, sparse collection of HOMOGENOUS elements
- Indexed by INTEGERs

In technical terms, a PL/SQL table:
• is like an ARRAY
• is like a SQL table; yet it is not precisely the same as either of those
  data structures
• is one type of collection structure
• is PL/SQL's way of providing arrays

INDEX BY Tables OR PL/SQL Tables
~~~~~~~~~~~~~~~
• Are composed of two components:
  – Primary key of data type BINARY_INTEGER
  – Column of scalar or record data type
• Can increase in size dynamically because they are unconstrained

Creating an INDEX BY Table OR PL/SQL Table
~~~~~~~~~~~~~~~~~~~~~~~~~
Syntax:
```
-- Definition of the TABLE
TYPE type_name IS TABLE OF
   {column_type | variable%TYPE
   | table.column%TYPE} [NOT NULL]
   | table.%ROWTYPE
   [INDEX BY BINARY_INTEGER];

-- Declare TABLE
identifier type_name;
```

Example:
Declare an INDEX BY table to store names.

```
TYPE enameTable IS TABLE OF
      employees.first_name%TYPE
  INDEX BY BINARY_INTEGER;

names       enameTable;
```

Example-2:
```
TYPE genderTable IS TABLE OF char(6)
      INDEX BY BINARY_INTEGER;

custGender  genderTable;
```

* These tables are unconstrained tables.
* You cannot initialize a PL/SQL table in its declaration.

For example:
```
custGender := ('Male', 'Female');  -- Error
```

Referencing PL/SQL Tables
~~~~~~~~~~~~~~~~~~~~~~~~
To assign values to specific rows, the following syntax is used:
```
PLSQL_table_name( primary_key_value ) := PLSQL_expression
```

Example:
```
names(1) := 'Apoorva';            <-- C/C++   strcpy(names[1],
"Apoorva");
ages(1)  := 23;             <-- C/C++   ages[1] = 23;
```
~~~~~~~~~~~~~~~~~~~~~~~~

User-Defined SubTypes

User defined SUBTYPES are subtypes based on an existing type.
They can be used to give an alternate name to a type.

NOTE: Similar to TYPEDEF in C/C++

Syntax:
```
SUBTYPE New_Type IS original_type;
```

Example:
```
SUBTYPE CounterType  IS    NUMBER;
v_counter        CounterType;

SUBTYPE string IS VARCHAR2;
v_custName       string(20);
```

Basic PL/SQL : Lesson-03
Cursors and Exception Handling

Cursors

Every SQL statement executed by the Oracle Server has an individual cursor
associated with it:
- Implicit cursors: Declared for all DML and PL/SQL SELECT statements
- Explicit cursors: Declared and named by the programmer

A cursor is a “handle” or “name” for a private SQL area.

For queries that return “more than one row”, you must declare an explicit cursor.

Thus the two types of cursors are:
- Implicit
- Explicit

Implicit Cursor

- The PL/SQL engine takes care of automatic processing.
- PL/SQL implicitly declares cursors for all DML statements.
- They are simple SELECT statements and are written in the BEGIN block (executable section) of the PL/SQL.
- They are easy to code, and they RETRIEVE EXACTLY ONE row

Processing Implicit Cursors

- Oracle implicitly opens a cursor to process each SQL statement that is not associated with an explicitly declared cursor.
- This implicit cursor is (AKA) known as SQL Cursor.
- Program cannot use the OPEN, FETCH, and CLOSE statements to control the SQL cursor. PL/SQL implicitly does those operations .

• You can use CURSOR ATTRIBUTES to get information about the most recently
  executed SQL statement.
      SQL%NOTFOUND
      SQL%ROWCOUNT
      SQL%FOUND
      SQL%ISOPEN
• Implicit cursor attributes are used to verify the outcome of DML statements
• Implicit Cursor is used to process INSERT, UPDATE, DELETE, and single row
  SELECT INTO statements.

Examples: Refer PPTs

Explicit Cursors
==========
• The set of rows returned by a query can consist of zero, one, or multiple
  rows, depending on how many rows meet your search criteria.
• When a query returns multiple rows, you can explicitly declare a cursor to
  process the rows.
• You can declare a cursor in the declarative part of any PL/SQL block,
  subprogram, or package.
• Processing has to be done by the user.

Processing Explicit Cursors
~~~~~~~~~~~~~~~~~~~~~~~~~~
[1] Declare
      Create a named SQL area
[2] Open
      Identify the active area
[3] Fetch
      Load the current row into variables
[4] Test
      Test for existing rows
      Return to Fetch if rows are found
[5] Close
      Release the active area

Declare a Cursor
~~~~~~~~~~~~~~
Syntax:
CURSOR cursor_name IS
   select_statement;

Example:
CURSOR empCursor IS
   SELECT first_name, job_id, salary
   FROM employees
   WHERE department_id IN (100, 110);

NOTE: If processing rows in a specific sequence is required, use the
      ORDER BY clause in the query.

Opening the Cursor
~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~

Syntax:
```
OPEN cursor_name;
```

Example:
```
OPEN  empCursor;
```

- Open the cursor to execute the query and identify the active set.
- If the query returns no rows, no exception is raised.
- Use cursor attributes to test the outcome after a fetch.

Fetching Data from the Cursor
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Syntax:
```
FETCH cursor_name INTO [variable1, variable2, ...] | record_name];
```

- Retrieve the current row values into variables.
- Include the same number of variables.
- Match each variable to correspond to the columns positionally.
- Test to see whether the cursor contains rows.

Example:
```
LOOP
  FETCH emp_cursor INTO v_empno,v_ename;
  EXIT WHEN ...;
  ...
  -- Process the retrieved data
  :
END LOOP;
```

Closing the Cursor
~~~~~~~~~~~~~~~~~
Syntax:
```
CLOSE cursor_name;
```

Example:
```
CLOSE empCursor;
```

- Close the cursor after completing the processing of the rows.
- Reopen the cursor, if required.
- Do not attempt to fetch data from a cursor after it has been closed.

Explict Cursor Attributes
=========================

Attribute	Type	Description
%ISOPEN	Boolean	Evalutes to TRUE if the cursor is open
%NOTFOUND	Boolean	Evalutes to TRUE if the most recent fetch does not return a row
%FOUND	Boolean	
%ROWCOUNT	Number	Evalutes the total number of rows returned so far.

Cursor FOR Loop
===============
Syntax:
```
FOR record_name IN cursor_name LOOP
  statement1;
```

```
  statement2;
. . .
END LOOP;
```

Advantages:
- The cursor FOR loop is a shortcut to process explicit cursors.
- Implicit open, fetch, exit, and close occur.
- The record is implicitly declared.

Cursor with Parameters

Syntax:

```
CURSOR cursor_name[(parameter_name datatype, ...)]
IS select_statement;
```

- Pass parameter values to a cursor when the cursor is opened and the query is executed.
- Open an explicit cursor several times with a different active set each time.

```
OPEN cursor_name(parameter_value,.....) ;
```

Example:
Pass the department number to the WHERE clause, in the cursor SELECT statement.

```
CURSOR empCursor(p_deptID NUMBER) IS
   SELECT first_name, job_id, salary
   FROM employees
   WHERE department_id = p_deptID;
```

Thus will opening the cursor we specify the department_id as follows:

```
OPEN empCursor( 100 );
```

We can even use a Cursor FOR loop to process a parameterized cursor.

Example:

```
      FOR empRec IN empCursor( v_deptID )  LOOP
            -- Processing the rows
            DBMS_OUTPUT.PUT_LINE('First Name : '|| empRec.first_name);
            DBMS_OUTPUT.PUT_LINE('Job ID     : '|| empRec.job_id);
            DBMS_OUTPUT.PUT_LINE('Salary     : '|| empRec.salary);
      END LOOP;
```

The FOR UPDATE Clause

- The method of locking records which are selected for modification, consists of two parts:
- The FOR UPDATE clause in CURSOR declaration.
- The WHERE CURRENT OF clause in an UPDATE or DELETE statement.

Syntax:

```
SELECT ...
```

```
FROM ...
FOR UPDATE [OF column_reference][NOWAIT];
```

- Use explicit locking to deny access for the duration of a transaction.
- Lock the rows before the update or delete.

Example:
Retrieve the employees who work in department 80 and update their salary.

```
DECLARE
CURSOR emp_cursor IS
  SELECT employee_id, last_name, department_name
  FROM employees,departments
  WHERE employees.department_id = departments.department_id
  AND employees.department_id = 80
FOR UPDATE OF salary NOWAIT;
```

The WHERE CURRENT OF Clause
==========

Syntax:

```
WHERE CURRENT OF cursor ;
```

- Use cursors to update or delete the current row.
- Include the FOR UPDATE clause in the cursor query to lock the rows first.
- Use the WHERE CURRENT OF clause to reference the current row from an explicit cursor.

Example:

```
DECLARE
  CURSOR sal_cursor IS
    SELECT e.department_id, employee_id, last_name, salary
    FROM employees e, departments d
    WHERE d.department_id = e.department_id
    and d.department_id = 60
    FOR UPDATE OF salary NOWAIT;
BEGIN
    FOR emp_record IN sal_cursor LOOP
      IF emp_record.salary < 5000 THEN
          UPDATE employees
            SET salary = emp_record.salary * 1.10
            WHERE CURRENT OF sal_cursor;
        END IF;
    END LOOP;
END;
/
```

Refer to yet another example in the PPTs

==========

Handling Exceptions
==========

If we desire to handle exception in PL/SQL it is possible

What is an exception?
~~~~~~~~~~
~~~~~~~~~~

• An exception is an identifier in PL/SQL that is raised during execution.
• How is it raised?
  – An Oracle error occurs.
  – You raise it explicitly.
• How do you handle it?
  – Trap it with a handler.
  – Propagate it to the calling environment.

Examples of internally defined exceptions:
ZERO_DIVIDE Divide by zero error
STORAGE_ERROR Out of memory error

Handling Exceptions
~~~~~~~~~~~~~~~~~~~
When an exception occurs in the execution part of the PL/SQL block, the control is transferred to the exception handler in the EXCEPTION section (if it exists)

The SQL Developer can write code to handle the exception in the EXCEPTION section of the PL/SQL code

If the exception is not handled in the EXCEPTION section, the exception is propogates to the calling environment.

Exception Types
===============
Exception types can be classified as follows:

[1] Implicitly raised

  Pre-defined Oracle Server exceptions
    They are Named IDENTIFIERS
  Non-predefined Oracle Server exceptions
    Don't have a name
    Also known as NUMBERED Exceptions

[2] Explicitly raised

  User-define exceptions

Trapping Exceptions
===================
Syntax:

```
EXCEPTION
EXCEPTION
  WHEN exception1 [OR exception2 . . .] THEN
    statement1;
    statement2;
    . . .
  [WHEN exception3 [OR exception4 . . .] THEN
     statement1;
     statement2;
     . . .]
  [WHEN OTHERS THEN
     statement1;
     statement2;
```
~~~~~~~~~~~~~~~~~~~

...]

• The EXCEPTION keyword starts exception-handling section.
• Several exception handlers are allowed.
• Only one handler is processed before leaving the block.
• WHEN OTHERS is the last clause

Trapping Predefined Oracle Server Exceptions

• Reference the standard name in the exception handling routine.
• Sample predefined exceptions:
  – NO_DATA_FOUND
  – TOO_MANY_ROWS
  – INVALID_CURSOR
  – ZERO_DIVIDE
  – DUP_VAL_ON_INDEX

Trapping Non-predefined Exceptions

AKA - Numbered Exceptions

• An exception name can be associated with an ORACLE error.
• This gives us the ability to trap the error specifically to ORACLE errors
• This is done with the help of “Compiler Directives”
• PRAGMA EXCEPTION_INIT

i.e. In the declarative section do the following:
(a) Name the exception
(b) Code the PRAGMA EXCEPTION_INIT

In the exception handling section
   Handle the exception

Trapping User-Defined Exception

(a) Name the exception
      Done in the DECLARative section
(b) Explictily raise the exception using the
      RAISE statement in the execution section
(c) Handle the raised exception
      EXCEPTION section

Other Exceptions

AKA as Wildcard Exception

When we are not sure of the exception type, we can use the WHEN OTHERS option
in the EXCEPTION section.

Moreover, the SQLCODE function will give the Error Code number
and SQLERRM function will return the Error message.

We can use them to our advangtage.