--# Chapter 5
--# PL/SQL Collections and Records

PL/SQL let us define two kinds of composite data types: collections and record.
A composite data type stores values that have intrnal components.
We can pass entire composite variables to subprograms as parameters and we can
access internal components of composite variable individually. Internal components
can be either Scalar or Composite.

Note: If we pass a composite variable as a parameter to a remote subprogram, then we
must create a redundant look-back DATABASE LINK, so that when the remote
subprogram compiles, the type checker that verifies the source users the same defination of the
user-defined composite type as the invoker users.

In a collection, the internal components always have the same data type, and are called
elements. We can access each element of a collection variable by its unique index,
with this syntax:
   variable_name(index).
To create a collection variable, we either defione a collection type and then create a variable
of that type or use %TYPE.

In a record, the internal components can have different data types, and are called fields.
we can access each field of a record variable by its name,
with this syntax:
   variable_name.field_name.
To Create a record variable, we either define a RECORD type and the then
create a variable of that type or use %ROWTYPE or %TYPE.

5.1 Collection Types:
   PL/SQL has three collection types
   . associative array
   . varray (variable size array),
   . nested table.

   . Number of Elements:
   If the number of elements is apecified, it is the maximum number of elements in the
   collection. If the number of elements is unspecified, the maximum number of elements in the
   collection is the upper limit of the index type.

   . Desnse or Parse:
   A dense collection has no gaps between elements-every element between the first and
   last element is defined and has a value (The value can be NULL unless the element has not
      null constraint).
   A sparse collection has a gap between elements.

   . Uninititalized Status
   An empty collection exists but has no elements. To add elements to an empty collection,
   invoke the EXTEND method.

   A null collection (also called an atomically null collection) does not exist. To change a null
   collection to an existing collection, we must initialize it, either by making it empty or by
   assigning a non-null value to it. We can not use the EXTEND method to initialize a null collection.

   . Where Defined
   A collection type defined in a PL/SQL block is a local type. It is available only in the block,
   and is stored in the database only of the block is in a standalone or package subprogram.

   A collection type defined in a package specification is a public item. we can reference it
   from outsie the package by qualifying it wit the package name (package_name.type_name).
   It is stored in the database until we drop the package.

   A collection type defined at schema level is a standalone type. We create it with the
   "CREATE TYPE statement". It is stored in the database until we drop it with the "DROP

TYPE Statemnet".

Note:
A collection type defined in a pcakge specification is incompatible with an
identically defined local or standalone collection type.

. Can be ADT attribute Data Type
To be an ADT attribute data type, a collection type must be a stand alone collection type.

ADT (Abstract DataType) is a user defined data type (also referred to as UDT's).

Examples
To define a new datatype to store a person's address:

```
CREATE OR REPLACE TYPE persons_address AS OBJECT (
streetNumber NUMBER,
streetName   VARCHAR2(30),
citySuburb   VARCHAR2(30),
state        VARCHAR2(4),
postCode     NUMBER
);
```

Define a employee_type; define a "raise_sal" member function; and create a table based on our new type:

```
CREATE TYPE employee_t AS OBJECT (
   name    VARCHAR2(30),
   ssn     VARCHAR2(11),
   salary  NUMBER,
   MEMBER FUNCTION raise_sal RETURN NUMBER)
/

CREATE TYPE BODY employee_t AS
   MEMBER FUNCTION raise_sal RETURN NUMBER IS
   BEGIN
      RETURN salary * 1.1;
   END;
END;
/

-- Test the member function
SELECT employee_t('Frank', '12345', 1000).raise_sal() from dual;

-- Create table based on employee_t
CREATE TABLE emp2 OF employee_t;
INSERT INTO emp2 VALUES ( employee_t('Frank', '12345', 1000) );
SELECT x.raise_sal() FROM emp2 x;
```

5.2 Associative Arrays

An associative array (formerly called PL/SQL table or index by table) is a set of
key-value pairs. Each eky is a unique index, used to locate the associated value with
the syntax variable_name(index).

The datatype of index can be either a string type (VARCHAR2, VARCHAR, STRING, or LONG)
or PLS_INTEGER. Indexes are stored in sort order, not creation order.

Like a database table, an associative array:
. Is empty (but not null) until we populate it
. Can hold an unspecified number of elements, which we can access without
  knowing their positions

Unlike a database table, an associative array:
. Does not need disk space or network operations
. Cannot be manipulated with DML statements

Example 5-1 Associative Array Index by String

```
DECLARE
   -- Associative array index by string

   TYPE population IS TABLE OF NUMBER       -- Associative array type
      INDEX BY VARCHAR2(64);               -- index by string

   city_population       population;          -- Associative array variable
   i                     VARCHAR2(64);       -- Scalar Variable

BEGIN
   -- Add elements (key value pairs) to associative array;

   city_population('Smallville')    := 2000;
   city_population('Midland')       := 750000;
   city_population('Megalopolis')   := 1000000;

   -- Change value associated with key 'Smallville':

   city_population('Smallville')    := 2001;

   -- Print associative array:

   i := city_population.FIRST;     -- Get first element of array

   WHILE i IS NOT NULL LOOP
      DBMS_OUTPUT.PUT_LINE
         ('Population of '|| i || ' is ' || city_population(i));
      i := city_population.NEXT(i);     -- Get next element of array
   END LOOP;
END;
/
```

Example 5-2 Function Returns Associative Array Index by PLS_INTEGER

```
DECLARE
   TYPE sum_multiples IS TABLE OF PLS_INTEGER
   INDEX BY PLS_INTEGER;

   n     PLS_INTEGER := 5;       -- number of multiples to sum for display
   sn     PLS_INTEGER := 10;       -- number of multiples to sum
   m     PLS_INTEGER := 3;        -- multiple

   FUNCTION get_sum_multiples(
      multiple IN PLS_INTEGER,
      num      IN PLS_INTEGER
   ) RETURN sum_multiples
   IS
      s sum_multiples;
   BEGIN
      FOR i IN 1..num LOOP
         s(i) := multiple * ((i * (i + 1)) / 2); -- sum of multiples
      END LOOP;
      RETURN s;
   END get_sum_multiples;

   BEGIN
      DBMS_OUTPUT.PUT_LINE(
      'Sum of the first '|| TO_CHAR(n) || ' multiple of '||
      TO_CHAR(m) || ' is '|| TO_CHAR(get_sum_multiples(m, sn) (n))
      );
   END;
```

```
/
```

5.2.1 Declaring Associative Array Constants

When declaring an associative array constant, we must create a function that populates the associative array with its initial value and then invoke the function in the constant declaration.

Example 5-3 Declaring Associative array constant:

```
CREATE OR REPLACE PACKAGE my_types AUTHID CURRENT_USER IS
   TYPE my_aa IS TABLE OF CHARCHAR2(20) INDEX BY PLS_INTEGER;
   FUNCTION init_my_aa RETURN my_aa;
END my_types;
/

CREATE OR REPLACE PACKAGE BODY my_types IS
FUNCTION init_my_aa RETURN my_aa IS
   ret my_aa;

BEGIN
    ret(-10)    := '-ten';
    ret(0)      := 'zero';
    ret(1)      := 'one';
    ret(2)      := 'two';
    ret(3)      := 'Three ';
    ret(4)    := 'Four';
    ret(9)    := 'Nine';

    RETURN ret ;
END init_my_aa;
END my_types;
/
```

5.2.2 NLS Parameter Values Affect Associative Arrays Indexed by String.

National Language Support NLS parameters such as NLS_SORT, NLS_COMP, and NLS_DATE_FORMAT afftect associative arrays indexed by string.

5.2.2.1 Changing NLS Parameter Values after Populating Assiciative arrays.

The initialization parameters NLS_SORT and NLS_COMP determine the storages order of string indexes of an assiciative array.

5.2.2.2 Indexes of Data Types Other Than VARCHAR2

In the declaration of an assiciative array indexed by string, the string type must be VARCHAR2 or one of its subtypes.

However, we can populate the associative array with indexes of any data type that the TO_CHAR function can convert to VARCHAR2.

If our indexes have data types other than VARCHAR2 and its subtypes, ensure that these indexes remains consistant and unique if the values of initialization parameters change.

. Do not use TO_CHAR(sysdate) as an index.
. Do not use NVARCHAR2 indexes that might be converted to the same VARCHAR2 value.
. Do not use CHAR or VARCHAR2 indexes that differ only in case, accented characters, or punctuation characters.

5.2.2.3 Passing Associative Array to Remote Databases

if we pass an associative array as a parameter to a remote database, and the local and the remote database have different NLS_SORT and NLS_COMP values, then:
. The collection method FIRST, LAST, NEXT or PRIOR might return unexpted values or raise exception.

. Indexes that are unique on the local database might not be unique in the remote database.

5.2.3 Appropriate Uses for Associative Arrays

An associative array is appropriate for:

. A relatively small lookup table, which can be constructed in memory each time we invoke the subprogram or initialize the package that declares it.

. Passing collections to ans from the database server

5.3 Varrays (Variable-Size Arrays)

A varray (variable-size array) is an array whose number of elements can vary from zero (empty) to the declared maximum size.

To access an element of a varray variable, use the syntax:
variable_name(index).
The lower bound of index is 1; the upper bound is the current number of elements.
The upper bound changes as we add or delete elements, but cannot exceel the maximum size.

An uninitialized varray varibale is a null collection. we must initialize it, either by making it empty or by assignning a non-NULL value to it.

Example-5-4 Varray (variable-size array)

```
DECLARE
   TYPE foursome IS VARRAY(4) OF VARCHAR2(15);   -- varray type

   -- varray varibale initialized with constructor:

   team foursome := foursome('John', 'Mary', 'Alberto', 'Junita');

   PROCEDURE print_team (heading VARCHAR2) IS
   BEGIN
      DBMS_OUTPUT.PUT_LINE(heading);

      FOR i IN 1..4 LOOP
         DBMS_OUTPUT.PUT_LINE(i || '.' || team(i));
      END LOOP;

      DBMS_OUTPUT.PUT_LINE('----');
   END;
   /

   BEGIN
      print_team('2001 Team');

      team(3) := 'Pierre';     -- Change values of two elements
      team(4) := 'Yvonne';

      print_team('2005 Team');

      -- Invoke constructor to assign new values to varray variable:

      team := foursome('Arun', 'Amitha', 'Allan', 'Mae');

      print_team('2009 Team');
   END;
   /
```

5.3.1 Appropriate Uses for Varrays

A varray is appropriate when:

. We know the maximum number of elements.

. We usually access the elements sequenctially.

Because we must store or retrive all elements at the same time, a varray might be impractical for large numbers of elements.

5.4 Nested Tabels

In the database, a nested table is a column type that stores an unspecified number of rows on a no particular order.

when we retrive a nested table value from the database into PL/SQL nested table variable, PL/SQL gives the rows consecutive indexes, statring ar 1.
Using these indexes we can access the individual rows of the nested table variable.
The syntax is:
variable_name(index).

The indexes and row order of a nested table might not remain stable as we store and retrive the nested table from the database.

The amount of memory that a nested table variable occupies can increase or decrease dynamically, as we add or delete elements.

An uninitialized nested table variable is a null collection. we must initialize it, either by making it empty or by assigning a non-NULL value to it.

Example 5-5 Nested Table of Local Type:

```
DECLARE
   TYPE Roster IS TABLE OF VARCHAR2(15);      -- nested tbale type

   -- nested tbale varibale initialized with constructor:

   name Roster := Roster('D Caruso', 'J hamil', 'D Piro', 'R Singh');

   PROCEDURE print_names (heading VARCHAR2) IS
   BEGIN
      DBMS_OUTPUT.PUT_LINE(heading);

      FOR i IN names.FIRST .. names.LAST LOOP     -- for first to last element
         DBMS_OUTPUT.PUT_LINE(names(i));
      END LOOP;

      DBMS_OUTPUT.PUT_LINE('------');
   END;

BEGIN
   print_names('Initial Vlaues: ');

   names(3) := 'P Perez';     -- changes value of one element
   print_names('Current Values');

   names := Roster('A Jansen', 'B Gupta');     -- Change entire table
   print_names('Current Values:');
END;
/
```

Example 5-6 Nested Table of Standalone Type

```
CREATE OR REPLACE TYPE nt_type IS TABLE OF NUMBER;
/

CREATE OR REPLACE PROCEDURE print_nt (nt nt_type) AUTHID DEFINER IS
   i NUMBER;
BEGIN
   i := nt.FIRST;

   IF i IS NULL THEN
```

```
      DBMS_OUTPUT.PUT_LINE('nt is empty');
    ELSE
      WHILE i IS NOT NULL LOOP
        DBMS_OUTPUT.PUT_LINE('nt. (' || i || ') =');
        DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(nt(i)), 'NULL'));
        i := nt.NEXT(i);
      END LOOP;
    END IF;

    DBMS_OUTPUT.PUT_LINE('-------');
  END print_nt;
  /

  DECLARE
    nt nt_type := nt_type();    -- nested table variabel initialized to empty
  BEGIN
    print_nt(nt);
    nt := nt_type(90, 2, 29, 58);
    print_nt(nt);
  END;
  /
```

5.4.1 Important Difference Between Nested Tables and Arrays

Conceptually, a nested table is like a one-dimensional array with an arbitrary number of elements. However, a nested table differs from an array in these important ways:

- An array has a declared number of elements, but a nested table does not. The size of a nested table can increase dynamically.
- An array is always dense. A nested array is dense initially, but it can become sparse, because you can delete elements from it.

5.4.2 Appropriate Use for Nested Tables

A nested table is appropriate when:

. THe number of elements is not set.

. Index values are not consecutive.

. We must delete or update some elements, but not all elements simultaneously.

Nested table data is stores in a separate store table, a system-generated database table. We we access a nested table, the database joind the nested table with its store table. this makes nested table suitable for queries and updates that affective only some elements of the collection.

5.5 Collection Constructors

A collection constructor is a system-defined function with the same name as a collection type, which returns a collection of that type.

The syntax of a constructor invocation is:
collection_type ([value [, value]...])

If the parameter list is empty, the constructor returns an empty collection. Otherwise, the constructor returns the specified values.

Example 5-7 Initializing Collection (Varray) Variable to Empty

```
  DECLARE
    TYPE Fouresome IS VARRAY(4) OF VARCHAR2(15);
    team Fouresome := Fouresome();    == initialize to empty

    PROCEDURE print_team(heading VARCHAR2)
      IS
    BEGIN
      DBMS_OUTPUT.PUT_LINE(heading);

      IF team.COUNT = 0 THEN
        DBMS_OUTPUT.PUT_LINE('Empty');
```

```
      ELSE
        FOR i IN 1..4 LOOP
          DBMS_OUTPUT.PUT_LINE(i || '.' || team(i));
        END LOOP;
      END IF;
    END;
  BEGIN
    print_team('Team');

    team := Fouresome('John', 'Mary', 'Alberto', 'Juanita');

    print_team('Team:');
  END;
  /
```

5.6 Qualified Expressions Overview

Qualified expressions imrpove program clarity and developer productivity by providing the ability to decalare and define a complex value in a compact form where the value is needed.

Example 5-8 Assigning Values to RECORD TYPE Variabls using Qualified Expression

```
CREATE PACKAGE pkg IS
  TYPE rec_t IS RECORD
  (year PLS_INTEGER := 2,
  name VARCHAR2(100) );
END;

DECLARE
  v_rec1 pkg.rec_t := pkg.rec_t(1847, 'ONE EIGHT FOUR SEVER');
  v_rec2 pkg.rec_t := pkg.rec_t(year => 1, name => 'ONE');
  v_rec3 pkg.rec_t := pkg.rec_t(NULL, NULL);

  PROCEDURE print_rec (pi_rec pkg.rec_t := pkg.rec_t(1847+1, 'a'||'b' ))
  IS
    v_rec1 pkg.rec_t := pkg.rec_t(2847, 'TWO EIGHT FOUR SEVEN');
  BEGIN
    DBMS_OUTPUT.PUT_LINE(NVL (v_rec1.year, 0) || ' ' || NVL(v_rec1.name, 'N/A'));
    DBMS_OUTPUT.PUT_LINE(NVL (pi_rec1.year, 0) || ' ' || NVL(pi_rec1.name, 'N/A'));
END;

BEGIN
  print_rec(v_rec1);
  print_rec(v_rec2);
  print_rec(v_rec3);
  print_rec();
END;
/
```

Example 5-9 Assigning Values to Associative array type variable using Qualified Expressions

```
CREATE FUNCTION print_bool(v IN BOOLEAN)
  RETURN VARCHAR2
IS
  v_rtn    VARCHAR2(10);
BEGIN
  CASE v
    WHEN TRUE THEN
      v_rtn := 'TRUE';
    WHEN FALSE THEN
      v_rnt := 'FALSE';
```

```
      ELSE
        v_rtn := 'NULL';
    END CASE;

    RETURN v_rtn;
  END print_bool;
```

The variable v_aa1 is initialized using index key-value pairs.

```
  DECLARE
    TYPE t_aa TABLE OF BOOLEAN INDEX BY PLS_INTEGER;
    v_aa1 t_aa := t_aa(1 => FALSE,
                       2 => TRUE,
                       3 => NULL);
  BEGIN
    DBMS_OUTPUT.PUT_LINE(print_bool(v_aa1(1)));
    DBMS_OUTPUT.PUT_LINE(print_bool(v_aa1(2)));
    DBMS_OUTPUT.PUT_LINE(print_bool(v_aa1(3)));
  END;
  /
```

5.7 Assigning Values to Collection Variables

We can assign a value to a collection varibale in these ways:

. Invoke a constructor to create a collection and assign it to the collection variable.

. Use the assignment statemet to assign it the value of another existing collection variable.

. Pass it to a subprogram as an OUT or IN OUT parameter, and then assign the value inside the subprogram.

. Use a qualified expression to assign values to an associative array

To assign a value to a scalar element of a collection variable, reference the element as collection_variable_name(index) as assign it a value.

5.7.1 Data Type Compatibility

We can assign a collection to a collection variable only if they have the same data type. Having the same elelement type is not enough.

Example 5-10 Data Type Compatibility for Collection Assignment

```
  DECLARE
    TYPE triplet IS VARRAY(3) OF VARCHAR2(15);
    TYPE trio    IS VARRAY(3) OF VARCHAR2(15);

    group1 triplet := triplet('Jones', 'Wong', 'Marceau');
    group2 triplet;
    group3 trio;
  BEGIN
    group2 := group1;   -- succeeds
    group3 := group1;    -- Fails
  END;
  /
```

5.7.2 Assigning Null Values to Varray or Nested Table Varibales

To a varray or nested table varibale, we can assign the value NULL or a null collection of the same data type. Either assignment makes the variable null.

Example 5-11 Assigning Null Value to Nested Table Variable

```
  DECLARE
    TYPE dnames_tab IS TABLE OF VARCHAR2(30);

    dept_names dnames_tab := dnames_tab(
      'shipping', 'sales', 'finance', 'payroll', 'development'); -- Initializing to non-null value

    empty_set dnames_tab;   -- Not initialized, therefore null
```

```
  PROCEDURE print_dept_names_status IS
  BEGIN
    IF dept_names IS NULL THEN
      DBMS_OUTPUT.PUT_LINE ('dept_name is null.')
    ELSE
      DBMS_OUTPUT.PUT_LINE('dept_names is not null.');
    END IF;
  END print_dept_names_status;
BEGIN
  print_dept_names_status;

  dept_names := empty_set;    -- Assign null collection to dept_names.

  print_dept_names_status;

  dept_names := dnames_tab(
    'Shipping', 'Sales', 'Finance', 'Payroll');    -- Re-initialize dept_names
  print_dept_names_status;
END;
/
```

5.7.3 Assigning Set Operation Results to Nested Table Variblas

To a nested table variable, we can assign the result of a SQL MULTISET operations or SQL SET function invocation.

The SQL MULTISET operators combine two nested tables into a single nested table. The elements of the two nested tables must have compareble data types. The SQL SET function takes a nested argument and retruns a nested table of the same data type whose elements are distinct.

Example 5-12 Assigning Set Operation Results to Nested Table Variable

```
DECLARE
  TYPE nested_type IS TABLE OF NUMBER;
  nt1   nested_type := nested_type(1, 2, 2);
  nt2 nested_typ    := nested_typ(3, 2, 1);
  nt3 nested_typ    := nested_typ(2, 3, 1, 3);
  nt4 nested_typ    := nested_typ(1, 2, 4);

  answer nested_typ;

  PROCEDURE print_nested_table(nt nested_typ) IS
    output VARCHAR2(128);
  BEGIN
    IF nt IS NULL THEN
      DBMS_OUTPUT.PUT_LINE('Result: null set');
    ELSIF nt.COUNT = 0 THEN
      DBMS_OUTPUT.PUT_LINE('Result: empty set');
    ELSE
      FOR i IN nt.FIRST .. nt.LAST LOOP    -- FOr first to last elemet
        output := output || nt(i) || ' ';
      END LOOP;
      DBMS_OUTPUT.PUT_LINE('Result: ' || output);
    END IF;
  END print_nested_table;

BEGIN
  answer   := nt1 MULTISET  UNION nt4;
  print_nested_table(answer);
  answer    := nt1 MULTISET UNION nt3;
  print_nested_table(answer);
  answer    := nt1 MULTISET UNION DISTINCT nt3;
```

```
    print_nested_table(answer);
    answer   := nt2 MULTISET INTERSECT nt3;
    print_nested_table(answer);
    answer   := nt2 MULTISET INTERSECT DISTINCT nt3;
    print_nested_table(answer);
    answer   := SET(nt3);
    print_nested_table(answer);
    answer   := nt3 MULTISET EXCEPT nt2;
    print_nested_table(answer);
    answer := nt3 MULTISET EXCEPT DISTINCT nt2;
    print_nested_table(answer);
  END;
  /
```

5.8 Multidimensional Collections

Although a collection has only one dimention, we can model a multidimensional collection with a collection whose elements are collections

Example 5-13 Two-Dimensional Varray (Varray of Varrays)

```
DECLARE
    TYPE t1 IS VARRAY(10) OF INTEGER;   -- varray of integer
    va t1 := t1(2,3,5);

    TYPE nt1 IS VARRAY(10) OF t1;       -- varray of varray of integer
    nva nt1 := nt1(va, t1(55, 6, 73), t1(2, 4), va);

    i integer;
    val t1;
BEGIN
    i := nva(2)(3);
    DBMS_OUTPUT.PUT_LINE('i = '|| i);

    nva.EXTEND;
    nva(5) := t1(56, 32);              -- replace inner varray elements
    nva(4) := t1(45, 43, 67, 43345);   -- replace an inner integer eleme t
    nva(4) (4) := 1;                   -- replace 43345 with 1

    nva(4).EXTEND;                     -- add element to 4th vrray element
    nva(4) (5) := 89;                  -- store integer 89 there
END;
/
```

Example 5-14 Nested Tables of Nested Tables and Varray of Integers

```
DECLARE
    TYPE tb1 IS TABLE OF VARCHAR2(20);      -- nested table of string
    vtb1 tb1 := tb1('one', 'three');

    TYPE ntb1 IS TABLE OF tb1;              -- nested table of nested tables of strings
    vntb1 ntb1   := ntb1(vtb1);

    TYPE tv1 IS VARRAY(10) OF INTEGER;      -- varray of integers
    TYPE ntb2 IS TABLE OF tv1;              -- nested table of vrrays of integer
    vntb2 ntb2   := ntb2(tv1(3,5), tv1(5,7,3));

BEGIN
    vntb1.EXTEND;
    vntb1(2) := vntb1(1);

    vntb1.DELETE(1);       -- delete first element of vntb1
    vntb1(2).DELETE(1);    -- delete first string from second table in nested table
END;
/
```

Example 5-15 Nested Tables of Associative Arrays and Varrays of Strings

```
DECLARE
  TYPE tb1 IS TABLE OF INTEGER INDEX BY PLS_INTEGER;       -- associative arrays
  v4   tb1;
  v5   tb1;

  TYPE aa1 IS TABLE OF tb1 INDEX BY PLS_INTEGER;          -- associative array of associative arrays
  v2 aa1;

  TYPE val IS VARRAY(10) OF VARCHAR2(10);                 -- varray of strings
  v1 val := val('hello', 'world');

  TYPE ntb2 IS TABLE OF val INDEX BY PLS_INTEGER;         -- assocoative array of varrays
  v3 ntb2;
BEGIN
  v4(1)   := 34;          -- populate associative array
  v4(2)   := 46456;
  v4(456) := 343;

  v2(23)    := v4;          -- populate associative array of associative arrays

  v3(34)    := val(33, 656, 343);       -- populate associative array varrays

  v2(35)   := v5;          -- assign empty associative array to v2(35)
  v2(35)(2)   := 78;
END;
/
```

5.9 Collection Comparisons

To determine if one collection variable is less than another,
we must define what less than means in that context and write a function that returns TRUE or FALSE.

5.9.1 Comparing Varray and Nested Table Variables to NULL

Use the IS [NOT] NULL operator when comparing to the NULL value.

we can compare varray and nested table variables to the value NULL with the "IS [NOT] NULL Operator",
but not with the relational operator equal(=) and not equal(<>, !=).

Example 5-16 Comparing Varray and Nested Table Variables to NULL

```
DECLARE
  TYPE Foursome IS VARRAY(4) OF VARCHAR2(15);       -- VARRAY type
  team Foursome;                                    -- varray variable

  TYPE Roster IS TABLE OF VARCHAR2(15);          -- nested table type
  names Roster   := Roster('Adams', 'Patel');   -- nested table variable

BEGIN
  IF TEAM IS NULL THEN
    DBMS_OUTPUT.PUT_LINE('team IS NULL');
  ELSE
    DBMS_OUTPUT.PUT_LINE('team IS NOT NULL')
  END IF;

  IF names IS NOT NULL THEN
    DBMS_OUTPUT.PUT_LINE('names IS NOT NULL');
  ELSE
    DBMS_OUTPUT.PUT_LINE('names IS NULL');
  END IF;
END;
/
```

5.9.2 Comparing Nested Tables for Equality and Inequality

Two nested tabel variables are equal if and only if they have the same set of elements (in any order).

If two nested table variables have the same nested table type, and that nested table type does not have elements of a record type, then we can compare the two variables for equality or inequality with the relational operator equal(=) or not equal(<>, !=);

Example 5-17 Comparing Nested Tables for Equality and Inequality

```
DECLARE
   TYPE dnames_tab IS TABLE OF VARCHAR2(30);       -- element type is not record type

   dept_names1 dnames_tab :=
      dnames_tab('Shipping', 'Sales', 'Finance', 'Payroll');

   dept_names2 dnames_tab :=
      dnames_tab('Sales', 'Finance', 'Shipping', 'Payroll');

   dept_names3 dnames_tab :=
      dnames_tab('Sales', 'Finance', 'Payroll');

BEGIN
   IF dept_names1 = dept_names2 THEN
      DBMS_OUTPUT.PUT_LINE('dept_names1 = dept_names2');
   END IF;

   IF dept_names2 != dept_anmes3 THEN
      DBMS_OUTPUT.PUT_LINE('dept_names2 != dept_names3');
   END IF;
END;
/
```

5.9.3 Comparing Nested Tables with SQL Multiset Conditions

We can compare nested table variables, and test some of their properties, with SQL multiset conditions.

Example 5-18 Comparing Nested Tables with SQL Multiset Conditions

```
DECLARE
   TYPE nested_typ IS TABLE OF NUMBER;
   nt1       nested_typ     := nested_typ(1, 2, 3);
   nt2       nested_typ    := nested_typ(3, 2, 1);
   nt3     nested_typ     := nested_typ(2, 3, 1, 3);
   nt4     nested_typ    := nested_typ(1, 2, 4);

   PROCEDURE testify (
      truth BOOLEAN := NULL,
      quantity NUMBER := NULL
   ) IS
   BEGIN
      IF thuth IS NOT NULL THEN
         DBMS_OUTPUT.PUT_LINE(
            CASE truth
               WHEN TRUE THEN 'True'
               WHEN FALSE THEN 'False'
            END
         );
      END IF;

      IF quantity IS NOT NULL THEN
         DBMS_OUTPUT.PUT_LINE(quantity);
      END IF;
```

```
  END;
BEGIN
  testify(truth => (nt1 IN (nt2, nt3, nt4)));
  testify(truth => (nt1 SUBMULTISET OF nt3));
  testify(truth => (nt1 NOT SUBMULTISET OF nt4));
  testify(truth => (4 MEMBER OF nt1));
  testify(truth => (nt3 IS A SET));
  testify(truth => (nt3 IS NOT A SET));
  testify(truth => (nt1 IS EMPTY));
  testify(quantity => (CARDINALITY(nt3)));
  testify(quantity => (CARDINALITY(SET(nt3))));
END;
/
```

5.10 Collection Methods

A collection method is a PL/SQL subprogram-either a function that returns information about a collection or a procedure that operates on a collection.

Note: With a null collection, EXISTS is the only collection method that does not raise the predefined expection COLLECTION_IS_NULL.

Methods

DELETE	PROCEDURE	Delete elements from collection.
TRIM	PROCEDURE	Deletes elements from end of varray or nested table.
EXTEND	PROCEDURE	Adds elements to end of varray or nested table.
EXISTS	Function	Returns TRUE if the only if specified elements of varray or nested table exists.
FIRST	FUNCTION	Returns first index in collection.
LAST	FUNCTION	Returns last index in collection.
COUNT	FUNCTION	Returns number of elements in collection.
LTRIM	FUNCTION	Returns maximum number of elements that collection can have.
PRIOR	FUNCTION	Returns index that precedes specified index.
NEXT	FUNCTION	Returns index that succeeds specified index.

5.10.1 DELETE collection Method

. DELETE deletes all elements from a collection of any type.
- DELETE(n) deletes the elements whose index is n, if that element exists.
- DELETE(m, n) deletes all elements whose indexes are in the range m..n, if both m and n exists and m <= n; otherwise it does nothing.

Example 5-19 DELETE Method with Nested Table

```
CREATE OR REPLACE TYPE nt_type IS TABLE OF NUMBER;
/

CREATE OR REPLACE PROCEDURE print_nt (nt nt_type) AUTHID DEFINER IS
  i NUMBER;
BEGIN
  i := nt.FIRST;

  IF i IS NULL THEN
    DBMS_OUTPUT.PUT_LINE('nt is empty');
  ELSE
    WHILE i IS NOT NULL LOOP
      DBMS_OUTPUT.PUT_LINE('nt. (' || i || ') =');
      DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(nt(i)), 'NULL'));
      i := nt.NEXT(i);
    END LOOP;
  END IF;

  DBMS_OUTPUT.PUT_LINE('-------');
END print_nt;
/
```

```
DECLARE
   nt nt_type := nt_type(11, 22, 33, 44, 55, 66);
BEGIN
   print_nt(nt);

   nt.DELETE(2);   -- Delete second element
   print_nt(nt);

   nt(2) := 2222;    -- Restore second element
   print_nt(nt)

   nt.DELETE(2, 4);    -- Delete range of elements
   print_nt(nt);

   nt(3) := 3333;       -- Restore third element
   print_nt(nt);

   nt.DELETE;          -- Delete all elements
   print_nt(nt);
END;
/
```

Example 5-20 DELETE Method with Associative Array Index by String

```
DECLARE
   TYPE aa_type_str IS TABLE OF INTEGER INDEX BY VARCHAR2(10);

   aa_atr    aa_type_str;

   PROCEDURE print_aa_str IS
      i VARCHAR2(10);
   BEGIN
      i   := aa_str.FIRST;

      IF i IS NULL THEN
         DBMS_OUTPUT.PUT_LINE('aa_str is empty');
      ELSE
         WHILE i IS NOT NULL LOOP
            DBMS_OUTPUT.PUT_LINE('aa_str.('||  i || ') = ');
            DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(aa_str(i)), 'NULL'));

            i := aa_str.NEXT(i);
         END LOOP;
      END IF;

      DBMS_OUTPUT.PUT_LINE('-----------');
   END print_aa_str;

BEGIN
   aa_str('M')    := 13;
   aa_str('Z')      := 26;
   aa_str('C')      := 3;

   aa_str.DELETE;      -- Delete all elements
   print_aa_str;

   aa_str('M')   := 13;
   aa_str('Z')   := 260;
   aa_str('C')   := 30;
   aa_str('W')   := 23;
   aa_str('J')   := 10;
   aa_str('N')   := 14;
   aa_str('P')   := 16;
   aa_str('W')   := 23;
```

```
    aa_str('J')    := 10;

    print_aa_str;

    aa_str.DELETE('C');
    print_aa_str

    aa_str.DELETE('N', 'W');
    print_aa_str;

    aa_str.DELETE('Z', 'M');
    print_aa_str;

END;
/
```

5.10.2 TRIM Collection Method

. TRIM removes one element from the end of the collection, if the collection has at least one element; otherwise it raises the predefined expection SUBSCRIPT_BEYOND_COUNT.

. TRIM(n) removes n elements from the end of the collection, if there are at least n elements at the end; otherwise it raises the predefined exception SUBSCRIPT_BEYOND_COUNT.

Caution:

Do not depend on interactive between TRIM and DELETE.

Treat nested tables like either fixed-size arrays and use only DELETE or stacks and use only TRIM and EXTEND.

Example 5-21 TRIM Method with Nested Table

```
CREATE OR REPLACE TYPE nt_type IS TABLE OF NUMBER;
/

CREATE OR REPLACE PROCEDURE print_nt (nt nt_type) AUTHID DEFINER IS
   i NUMBER;
BEGIN
   i := nt.FIRST;

   IF i IS NULL THEN
      DBMS_OUTPUT.PUT_LINE('nt is empty');
   ELSE
      WHILE i IS NOT NULL LOOP
         DBMS_OUTPUT.PUT_LINE('nt. (' || i || ') =');
         DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(nt(i)), 'NULL'));
         i := nt.NEXT(i);
      END LOOP;
   END IF;

   DBMS_OUTPUT.PUT_LINE('-------');
END print_nt;
/

DECLARE
   nt nt_type     := nt_type(11, 22, 33, 44, 55, 66);
BEGIN
   print_nt(nt);

   nt.TRIM;       -- Trim last element
   print_nt(nt);

   nt.DELETE(4);    -- Delete fourth element
   print_nt(nt);

   nt.TRIM(2);       -- Trim last two elements
   print_nt(nt);
```

```
END;
/
```

5.10.3 EXTEND Collection Method

. EXTEND appends one null element to the collection.
. EXTEND(n) appends n null elements to the collection.
. EXTEND(n, i) appends n copies of the ith element to the collection.

Note: EXTEND(n, i) is the only form that we can use for a collection whose elements have the NOT NULL constraint.

Example 5-22 EXTEND Method with Nested Table

```
CREATE OR REPLACE TYPE nt_type IS TABLE OF NUMBER;
/

CREATE OR REPLACE PROCEDURE print_nt (nt nt_type) AUTHID DEFINER IS
  i NUMBER;
BEGIN
  i := nt.FIRST;

  IF i IS NULL THEN
    DBMS_OUTPUT.PUT_LINE('nt is empty');
  ELSE
    WHILE i IS NOT NULL LOOP
      DBMS_OUTPUT.PUT_LINE('nt. (' || i || ') =');
      DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(nt(i)), 'NULL'));
      i := nt.NEXT(i);
    END LOOP;
  END IF;

  DBMS_OUTPUT.PUT_LINE('-------');
END print_nt;
/

DECLARE
  nt  nt_type := nt_type(11, 22, 33);
BEGIN
  print_nt(nt);

  nt.EXTEND(2, 1);     -- Append two copies of first element
  print_nt(nt);

  nt.DELETE(5);        -- Delete fifth element
  print_nt(nt);

  nt.EXTEND;           -- Append one null element
  print_nt(nt);
END;
/
```

5.10.4 EXISTS Collection Method

EXISTS(n) returns TRUE if the nth element of the collection exists and FALSE otherwise.
if n is out of range, EXISTS returns FALSE instead of raising the predefined expection SUBSCRIPT_OUTSIDE_LMIT.

For a deleted element, EXISTS(n) returns FALSE, even if DELETE kept a placeholder for it.

Example 5-23 EXISTS Method with Nested TABLE

```
DECLARE
  TYPE NumList IS TABLE OF INTEGER;
  n NumList  := NumList(1, 3, 5, 7);
BEGIN
  n.DELETE(2);  -- Delete second element
```

```
  FOR i IN 1..6 LOOP
    IF n.EXISTS(i) THEN
      DBMS_OUTPUT.PUT_LINE('n('||i||') = '|| n(i) );
    ELSE
      DBMS_OUTPUT.PUT_LINE('n('||i||') does not exists');
    END IF;
  END LOOP;
END;
/
```

5.10.5 FIRST and LAST Collection Methods

If the collection has at least one element, FIRST and LAST returns the indexs of the first and last elements, respectively (ignoring deleted elements even of DELETE kept placeholder for them). If the collection has only one element FIRST and LAST returns the same index. If the collection is empty, FIRST and LAST return NULL.

5.10.5.1 First and LAST methods for Associative array.

Example 5-24 FIRST and LAST Values for Associative ARRAY index by PLS_INTEGER

```
DECLARE
  TYPE aa_type_int IS TABLE OF INTEGER INDEX BY PLS_INTEGER;
  aa_int aa_type_int;

  PROCEDURE print_first_and_last IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE('FIRST = ' || aa_int.FIRST);
    DBMS_OUTPUT.PUT_LINE('LAST = ' || aa_int.LAST);
  END print_first_and_last;

BEGIN
  aa_int(1) := 3;
  aa_int(2) := 6;
  aa_int(3) := 9;
  aa_int(4) := 12;

  DBMS_OUTPUT.PUT_LINE('Before Deletions:');
  print_first_and_last;

  aa_int.DELETE(1);
  aa_int.DELETE(4);

  DBMS_OUTPUT.PUT_LINE('After deletions:');
  print_first_and_last;

END;
/
```

Example 5-25 FIRST AND LAST Values for Associative Array Index by String

```
DECLARE
  TYPE aa_type_str IS TABLE OF INTEGER INDEX BY VARCHAR2(10);
  aa_str aa_type_str;

  PROCEDURE print_first_and_last IS
    DBMS_OUTPUT.PUT_LINE('FIRST: '|| aa_str.FIRST);
    DBMS_OUTPUT.PUT_LINE('LAST: '|| aa_str.LAST);
  END print_first_and_last;

BEGIN
  aa_str('Z') := 52;
  aa_str('A') := 53;
```

```
    aa_str('K') := 54;
    aa_str('R') := 55;

    DBMS_OUTPUT.PUT_LINE('Before Deletions:');
    print_first_and_last;

    aa_str.DELETE('A');
    aa_str.DELETE('Z');

    DBMS_OUTPUT.PUT_LINE('After Deletions:');
    print_first_and_last;
  END;
  /
```

5.10.5.2 FIRST and LAST Method for Varray

For a varray that is not empty, FIRST always returns 1. for every varray LAST always equals COUNT.

Example 5-26 Printing Varray with FIRST and LAST in FOR LOOP

```
DECLARE
  TYPE team_type IS VARRAY(4) OF VARCHAR2(15);
  team team_type;

  PROCEDURE print_team (heading VARCHAR2)
  IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE(heading);

    IF team IS NULL THEN
      DBMS_OUTPUT.PUT_LINE('Does not exist');
    ELSIF team.FIRST IS NULL THEN
      DBMS_OUTPUT.PUT_LINE('Has no members');
    ELSE
      FOR i IN team.FIRST..team.LAST LOOP
        DBMS_OUTPUT.PUT_LINE(i || '. '|| team(i));
      END LOOP;
    END IF;

    DBMS_OUTPUT.PUT_LINE('-------');
  END print_team;
BEGIN
  print_team('Team Status:');

  team := team_type();    -- Team is funded, but nobody is on it.
  print_team('Team Status: ');

  team := team_type('John', 'Mary');    -- Put 2 members on team.
  print_team('Initial Team');

  team := team_type('Arun', 'Amitha', 'Allan', 'Mae');    -- change team
  print_team('New Team: ');
END;
/
```

5.10.5.3 FIRST and LAST Methods for Nested Table

For a nested table, LAST equals COUNT unless we delete elements from its middle, in which case LAST is larger than COUNT.

Example 5-27 Printing Nested Table with FIRST and LAST in FOR LOOP

```
DECLARE
  TYPE team_type IS TABLE OF VARCHAR2(15);
  team team_type;

  PROCEDURE print_team (heading varchar2) IS
```

```
    BEGIN
      DBMS_OUTPUT.PUT_LINE(heading);

      IF team IS NULL THEN
        DBMS_OUTPUT.PUT_LINE('Does not exist');
      ELIF team.FIRST IS NULL THEN
        DBMS_OUTPUT.PUT_LINE('Has no members');
      ELSE
        FOR i IN team.FIRST..team.LAST LOOP
          DBMS_OUTPUT.PUT(i || '. ');
          IF team.EXISTS(i) THEN
            DBMS_OUTPUT.PUT_LINE(team(i));
          ELSE
            DBMS_OUTPUT.PUT_LINE('(to be hired)');
          END IF;
        END LOOP;
      END IF;
      DBMS_OUTPUT.PUT_LINE('---------');
    END print_team;

  BEGIN
    print_team('Team Status:');

    team := team_type();    -- Team is funded, but nobody is on it.
    print_team('Team Status: ');

    team := team_type('Arun', 'Amitha', 'Allan', 'Mae');     -- add members
    print_team('Initial Team: ');

    team.DELETE(2, 3);    -- Remove 2nd and 3rd member;
    print_team('Current Team:');
  END;
  /
```

5.10.6 COUNT Collection Method

Count is a function that returns the number of elements in the collection (ignoring deleted elements, even if DELETE kept placeholders for them)

5.10.6.1 COUNT Method for Varray

Example 5-28 COUNT and LAST Values for Varray

```
  DECLARE
    TYPE NumList IS VARRAY(10) OF INTEGER;
    n NumList := NumList(1, 3, 5, 7);

    PROCEDURE print_count_and_last IS
    BEGIN
      DBMS_OUTPUT.PUT('n.COUNT = '|| n.COUNT ||', ');
      DBMS_OUTPUT.PUT_LINE('n.LAST = '|| n.LAST);

    END print_count_and_last;

  BEGIN
    print_count_and_last;

    n.EXTEND(3);
    print_count_and_last;

    n.TRIM(5);
    print_count_and_last;
  END;
  /
```

5.10.6.2 COUNT Method for Nested Table

For a nested table, COUNT equals LAST unless we delete elements from the middle of the nested table.

Example 5-29 COUNT and LAST Vaslues for Nested Table

```
DECLARE
  TYPE NumList IS TABLE OF INTEGER;
  n NumList := NumList(1, 3, 5, 7);

  PROCEDURE print_count_and_last IS
  BEGIN
    DBMS_OUTPUT.PUT('n.COUNT = '|| n.COUNT ||', ');
    DBMS_OUTPUT.PUT_LINE('n.LAST = '|| n.LAST);
  END print_count_and_last;
BEGIN
  print_count_and_last;

  n.DELETE(3);   -- Delete thirst element
  print_count_and_last;

  n.EXTEND(2);   -- Add two null elements to end
  print_count_and_last;

  FOR i IN 1..8 LOOP
    IF n.EXISTS(i) THEN
      IF n(i) IS NOT NULL THEN
        DBMS_OUTPUT.PUT_LINE('n(' || i ||') ='|| n(i));
      ELSE
        DBMS_OUTPUT.PUT_LINE('n('|| i ||') = NULL');
      END IF;
    ELSE
      DBMS_OUTPUT.PUT_LINE('n('|| i ||') does not exist');
    END IF;
  END LOOP;
END;
/
```

5.10.7 LIMIT Collection Method

LIMIT is a function that returns the maximum number of elements that the collection can have. If the collection has no maximum number of elements, LIMIT returns NULL. Only a varray has a maximum size.

Example 5-30 LIMIT and COUNT Values for Different Collection Types

```
DECLARE
  TYPE aa_type IS TABLE OF INTEGER INDEX BY PLS_INTEGER;
  aa aa_type;          -- associative array

  TYPE va_type IS VARRAY(4) OF INTEGER;
  va va_type(2, 4);          -- varray

  TYPE nt_type IS TABLE OF INTEGER;
  nt nt_type(1, 3, 5);          -- nested table

BEGIN
  aa(1) := 3;
  aa(2) := 6;
  aa(3) := 9;
  aa(4) := 12;

  DBMS_OUTPUT.PUT('aa.COUNT = ');
  DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(aa.COUNT), 'NULL'));
```

```
    DBMS_OUTPUT.PUT('aa.LIMIT = ');
    DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(aa.LIMIT), 'NULL'));

    DBMS_OUTPUT.PUT('va.COUNT = ');
    DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(va.COUNT), 'NULL'));

    DBMS_OUTPUT.PUT('va.LIMIT = ');
    DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(va.LIMIT), 'NULL'));

    DBMS_OUTPUT.PUT('nt.COUNT = ');
    DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(nt.COUNT), 'NULL'));

    DBMS_OUTPUT.PUT('nt.LIMIT = ');
    DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(nt.LIMIT), 'NULL'));
  END;
  /
```

5.10.8 PRIOR and NEXT Collection Methods

PRIOR and NEXT are functions that let us move backward and forward in the collection. These methods are useful for traversing sparse collections.

PRIOR returns the index of the preceding existing element of the collection, if one exist. Otherwise PRIOR returns NULL.

NEXT returns the index of the succeeding sxisting element of the collection, if one exists. Otherwise NEXT returns NULL.
The given index need not exist. However, if the collection c is a varray, and the index exceeds c.LIMIT, then:

- c.PRIOR(index) returns c.LAST.
- c.NEXT(index) returns NULL.

For example:

```
DECLARE
  TYPE arr_type IS VARRAY(10) OF NUMBER;
  v_numbers arr_type := arr_type();
BEGIN
  v_numbers.EXTEND(4);

  v_numbers(1)   := 10;
  v_numbers(2)   := 20;
  v_numbers(3)   := 30;
  v_numbers(4)   := 40;

  DBMS_OUTPUT.PUT_LINE(NVL(v_numbers.PRIOR(3400), -1));    -- 4
  DBMS_OUTPUT.PUT_LINE(NVL(v_numbers.NEXT(3400), -1));   -- -1

END;
/
```

Example 5-31 PRIOR and NEXT methods

```
DECLARE
  TYPE nt_type IS TABLE OF NUMBER;
  nt nt_type := nt_type(18, NULL, 36, 45, 54, 63);

BEGIN
  nt.DELETE(4);
  DBMS_OUTPUT.PUT_LINE('nt(4) was deleted');

  FOR i IN 1..7 LOOP
    DBMS_OUTPUT.PUT('nt.PRIOR(' || i ||')');
    DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(nt.PRIOR(i)), 'NULL'));

    DBMS_OUTPUT.PUT('nt.NEXT(' || i ||')');
```

```
      DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(nt.NEXT(i)), 'NULL'));
    END LOOP;
  END;
  /
```

Example 5-32 Printing Elements of Sparse Nested Table
This example prints the elements of a spase nested table from first to last, using FIRST and NEXT, and from last to first, using LAST and PRIOR

```
  DECLARE
    TYPE numlist IS TABLE OF NUMBER;

    n numlist := numlist(1, 2, null, null, 5, null, 7, 8, 9, null);
    idx INTEGER;
  BEGIN
    DBMS_OUTPUT.PUT_LINE('First to last:');
    idx := n.FIRST;

    WHILE idx IS NOT NULL LOOP
      DBMS_OUTPUT.PUT('n('||idx||') =');
      DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(n(idx)), 'NULL'));
      idx := n.NEXT(idx);
    END LOOP;

    DBMS_OUTPUT.PUT_LINE('----------------');

    DBMS_OUTPUT.PUT_LINE('Last to first:');
    idx := n.LAST;
    WHILE idx IS NOT NULL LOOP
      DBMS_OUTPUT.PUT('n(' || idx ||') =');
      DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(n(idx)), 'NULL'));
      idx := n.PRIOR(idx);
    END LOOP;
  END;
  /
```

5.11 Collection Types Defined in Package Specifications

A collection type defined in a package specification is incompatible with an identically defined local or standalone collection type.

Example 5-33 Identically Defined Package and Local Collection Types

```
  CREATE OR REPLACE PACKAGE pkg AS
    TYPE NumList IS TABLE OF NUMBER;
    PROCEDURE print_numlist (nums NumList);
  END pkg;
  /

  CREATE OR REPLACE PACKAGE BODY pkg AS
    PROCEDURE print_numlist (nums NumList) IS
    BEGIN
      FOR i IN nums.FIRST..nums.LAST LOOP
        DBMS_OUTPUT.PUT_LINE(nums(i));
      END LOOP;
    END print_numlist;
  END pkg;
  /

  DECLARE
    TYPE NumList IS TABLE OF NUMBER;   -- local type identical to package type
    n1 pkg.NumList := pkg.NumList(2, 4);   -- package type
    n2 NumList       := NumList(6, 8);       -- local type
  BEGIN
    pkg.print_numlist(n1);   -- succeeds
```

```
    pkg.print_numlist(n2);    -- fails

END;
/
```

Example 5-34 Indentical Defined Package and Standalone collection Types

```
CREATE OR REPLACE TYPE NumList IS TABLE OF NUMBER;
-- Standalone collection type identical to package type
/

DECLARE
   n1 pkg.NumList := pkg.NumList(2, 4);       -- package type
   n2 NumList       := NumList(6, 8);           -- Standalone type

BEGIN
   pkg.print_numlist(n1); -- success
   pkg.print_numlist(n2); -- fails
END;
/
```

5.12 Record Variables

We can create a record variable in any of these ways:

. Define a RECORD type and then declare a variable of that type.
. Use %ROWTYPE to declare a record variable that represents either a full or partial row of a database table or view.
. Use %TYPE to declare a record variable of the same type as a previously declared record variable.

5.12.1 Initial Values of Record Variables

For a record variable of a RECORD type, the initial value of each field is NULL unless we specify a different initial value for it when we define the type.

5.12.2 Decalring Record Constants

When declaring a record constant, we must create a function that populated the record with its initial value and then invoke the function in the constant declaration.

Example 5-35 Declaring Record Constant

This example creates a function that populate the record with its initial value and then invoke the function in the constant declaration.

```
CREATE OR REPLACE PACKAGE My_Types AUTHID CURRENT_USER IS
   TYPE My_Rec IS RECORD (a NUMBER, b NUMBER);
   FUNCTION Init_My_Rec RETURN My_Rec;
END My_Types;
/

CREATE OR REPLACE PACKAGE BODY My_Types IS
   FUNCTION Init_My_Rec RETURN My_Rec IS
      rec My_Rec;
   BEGIN
      rec.a := 0;
      rec.b := 1;
      RETURN rec;
   END Init_My_Rec;
END My_TYpes;
/

DECLARE
   r CONSTANT My_Types.My_Rec := My_Types.Init_My_Rec();
BEGIN
```

```
    DBMS_OUTPUT.PUT_LINE('r.a = '|| r.a);
    DBMS_OUTPUT.PUT_LINE('r.b = '|| r.b);
END;
/
```

5.12.3 RECORD Types

A RECORD type defined in a PL/SQL block is a local type. It is available only in the block, and is stored in the database only if the block is in a standalone or package subprogram.

A RECORD type defined in a package specification is a public item. You can reference it from outside the package by qualifying it with the package name (package_name.type_name). It is stored in the database until you drop the package with the DROP PACKAGE statement.

You cannot create a RECORD type at schema level. Therefore, a RECORD type cannot be an ADT attribute data type.

To define a RECORD type, specify its name and define its fields. To define a field, specify its name and data type. By default, the initial value of a field is NULL. You can specify the NOT NULL constraint for a field, in which case you must also specify a non-NULL initial value. Without the NOT NULL constraint, a non-NULL initial value is optional.

A RECORD type defined in a package specification is incompatible with an identically defined local RECORD type.

Example 5-36 RECORD Type Definition and Variable Declaration

```
DECLARE
   TYPE DeptRecType IS RECORD(
      dept_id       NUMBER(4) NOT NULL := 10,
      dept_name    VARCHAR2(30) NOT NULL := 'Administrator',
      mgr_id       NUMBER(6) := 200,
      loc_id       NUMBER(4) := 1700
   );

   dept_rec DeptRecType;
BEGIN
   DBMS_OUTPUT.PUT_LINE('dept_id: '|| dept.rec.dept_id);
   DBMS_OUTPUT.PUT_LINE('dept_name: '|| dept.rec.dept_name);
   DBMS_OUTPUT.PUT_LINE('mgr_id: '|| dept.rec.mgr_id);
   DBMS_OUTPUT.PUT_LINE('loc_id: '|| dept.rec.loc_id);
END;
/
```

Example 5-37 RECORD Type with RECORD Field (Nested Record)

This example defines two RECORD types, name_rec and contact. The type contact has a field of type name_rec.

```
DECLARE
   TYPE name_rec IS RECORD (
      first    employees.first_name%TYPE,
      last     employees.last_name%TYPE
   );

   TYPE contact IS RECORD (
      name name_rec,
      phone employees.phone_number%TYPE
   );

   friend contact;
BEGIN
   friend.name.first := 'Jhon';
   friend.name.last  := 'Smith';
   friend.phone := '1-650-555-1234';

   DBMS_OUTPUT.PUT_LINE(
      friend.name.first    || ' '||
      friend.name.last     || ' '||
      friend.phone
```

```
  );
END;
/
```

Example 5-38 RECORD Type with Varray Field
This defines a VARRAY type, full_name and a RECORD type contact. The type contact has a field of type full_name.

```
DECLARE
  TYPE full_name IS VARRAY(2) OF VARCHAR2(20);

  TYPE contact IS RECORD (
    name full_name := full_name('Jhon', 'Smith'), -- varray field
    phone employees.phone_number%TYPE
  );

  friend contact;
BEGIN
  friend.phone := '1-650-555-1234';

  DBMS_OUTPUT.PUT_LINE(
    friend.name(1) || ' '||
    friend.name(2) || ' '||
    friend.phone
  );
END;
/
```

Example 5-39 Identically Defined Package and Local RECORD Types

```
CREATE OR REPLACE PACKAGE pkg AS
  TYPE rec_type IS RECORD (
    f1 INTEGER,
    f2   VARCHAR2(4)
  );

  PROCEDURE print_rec_type (rec rec_type);
END pkg;
/

CREATE OR REPLACE PACKAGE BODY pkg AS
  PROCEDURE print_rec_type (rec rec_type) IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE(rec.f1);
    DBMS_OUTPUT.PUT_LINE(rec.f2);
  END print_rec_type;
END pkg;
/

DECLARE
  TYPE rec_type IS RECORD (
    f1 INTEGER,
    f2 VARCHAR2(4)
  );

  r1 pkg.rec_type;   -- package type
  r2       rec_type;   -- local type

BEGIN
  r1.f1 := 10;
  r1.f2 := 'abc';

  r2.f1 := 25;
```

```
    r2.f2 := 'xyz';

    pkg.print_rec_type(r1);   -- success
    pkg.print_rec_type(r2);   -- fails
END;
/
```

5.12.4 Declaring Items Using the %ROWTYPE Attribute

The %ROWTYPE attribute lets us declare a record variable that represents either a full or partial row of a database table or view.

5.12.4.1 Declaring a Record Variable that always Reprents Full Row

To declare a record variale that always represents a full row of a database table or view, use syntax:

```
variable_name table_or_view_name%ROWTYPE;
```

Example 5-40 %ROWTYPE Variable Reprents Full Database table row

```
DECLARE
    dept_rec departments%ROWTYPE;
BEGIN
    -- Assign values to field
    dept_rec.depertment_id := 10;
    dept_rec.department_name := 'Administrator';
    dept_rec.manager_id := 200;
    dept_rec.location_id := 1700;

    -- Print fields

    DBMS_OUTPUT.PUT_LINE('Department ID: '|| dept_rec.department_id);
    DBMS_OUTPUT.PUT_LINE('Department NAME: '|| dept_rec.department_name);
    DBMS_OUTPUT.PUT_LINE('MANAGER ID: '|| dept_rec.manager_id);
    DBMS_OUTPUT.PUT_LINE('LOCATION ID: '|| dept_rec.location_id);
END;
/
```

Example 5-41 %ROWTYPE variable Does not inherit initial values or constraints

```
DROP TABLE t1;
CREATE TABLE t1(
    c1 INTEGER DEFAULT 0 NOT NULL,
    c2 INTEGER DEFAULT 1 NOT NULL
);

DECALRE
    t1_row t1%ROWTYPE;
BEGIN
    DBMS_OUTPUT.PUT('t1.c1 = ');
    DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(t1_row.c1), 'NULL') );

    DBMS_OUTPUT.PUT('t1.c2 = ');
    DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(t1_row.c2), 'NULL') );

END;
/
```

5.12.4.2 Declaring a Record Variable that Can Represent Partial Row

To declare a record variable that can represent a partial row of a database table or view, use the syntax:

```
variable_name cursor%ROWTYPE;
```

Example 5-42 %ROWTYPE Variable Represnets Partial Database Table Row

```
DECLARE
  CURSOR c IS
    SELECT first_name, last_name, phone_number
    FROM employees;

  friends c%ROWTYPE;

BEGIN
  friends.first_name   := 'John';
  friends.last_name    := 'Smit';
  friends.phone_number:= '1-650-555-1234';

  DBMS_OUTPUT.PUT_LINE(
    friends.first_name || ' '||
    friends.last_name || ' '||
    friends.phone_number
  );
END;
/
```

Example 5-43 %ROWTYPE Variable Repreents Join ROW

```
DECLARE
  CURSOR c2 IS
    SELECT employee_id, email, employees.manager_id, location_id
    FROM employees, departments
    WHERE employees.department_id = depertments.department_id;

  join_rec c2%ROWTYPE; -- include column from two table

BEGIN
  NULL;
END;
/
```

5.12.4.3 %ROWTYPE Attribute and Virtual Columns

If we use the %ROWTYPE attribute to define a record variable that represnts a full row of a table that has a virtual column, then we can not insert that record into the table. Instead, we must insert the individual record fields into the table, exclusing the virtual column.

Example 5-44 Inserting %ROWTYPE Record into Table (Wrong)

```
DROP TABLE plch_departure;

CREATE TABLE plch_departure (
  destination      VARCHAR2(100),
  departure_time   DATE,
  delay            NUMBER(10),
  expected         GENERATED ALWAYS AS (departure_time + delay/24/60/60)
);

DECLARE
  dep_rec plch_departure%ROWTYPE;
BEGIN
  dep_rec.destination := 'X';
  dep_rec.departure_time := SYSDATE;
  dep_rec.delay := 1500;

  INSERT INTO plch_departure VALUES dep_rec;
END;
/
```

Example 5-45 Inserting %ROWTYPE Record into Table

```
DECLARE
  dep_rec plch_departure%ROWTYPE;
BEGIN
  dep_rec.destination := 'X';
  dep_rec.departure_time := SYSDATE;
  dep_rec.delay := 1500;

  INSERT INTO plch_departure (destination, departure_time, delay)
  VALUES (dep_rec.destination, dep_rec.departure_time, dep_rec.delay);
END;
/
```

5.12.4.4 %ROWTYPE Attribute and Invisible Columns

Suppose that we use the %ROWTYPE attribute to define a record variable that represent a row of a table that has an invisible column, and then we nake the invisibel column visible.

Example 5-46 %ROWTYPE affected by Making Invisible column Visible

```
CREATE TABLE t (
  a INT,
  b INT,
  c INT Invisibl
);

INSERT INTO t (a, b, c) VALUES(1, 2, 3);
COMMIT;

DECLARE
  t_rec t%ROWTYPE;    -- t_rec fields a and b but not c

BEGIN
  SELECT * INTO c_rec FROM t WHERE ROWNUM < 2;    -- t_rec(a)=1, t_rec(b)=2
  DBMS_OUTPUT.PUT_LINE('c = '|| t_rec.c);
END;
/
```

Make invisible column visible:

```
ALTER TABLE t MODIFY (c VISIBLE);
```

5.13 Assigning Values to Record Variables

A record variable means either a record variable or a record component of a composite variable.
To any record variable, we can assign a value to each field individually.

5.13.1 Assigning One Record Variable to Another

We can assign the value of one record variable to another record variable only in these case:

- The two variables have the same RECORD TYPE.
- The target variable is declared with a RECORD type, the source variable is declared with %ROWTYPE, their fields match in number and order, and correcponding fields have the same data type.

Example 5-47 Assigning Record to Another Record of Same RECORD Type

```
DECLARE
  TYPE name_rec IS RECORD (
    first    employees.first_name%TYPE DEFAULT 'John',
    last   employees.last_name%TYPE DEFAULT 'Doe'
  );
```

```
  name1 name_rec;
  name2 name_rec;
BEGIN
  name1.first := 'Jane';
  name1.last  := 'Smith';
  DBMS_OUTPUT.PUT_LINE('name1: '|| name1.first || ' ' || name1.last);

  name2 := name1;
  DBMS_OUTPUT.PUT_LINE('name2: '|| name2.first || ' ' || name2.last);
END;
/
```

Example 5-48 Assigning %ROWTYPE Record to RECORD Type Record

```
DECLARE
  TYPE name_rec IS RECORD (
    first   employees.first_name%TYPE   DEFAULT 'John',
    last    employees.last_name%TYPE    DEFAULT 'Doe'
  );

  CURSOR c IS
    SELECT first_name, last_name
    FROM employees;

  terget name_rec;
  source c%ROWTYPE;

BEGIN
  source.first_name := 'Jane';
  source.last_name  := 'Smith';

  DBMS_OUTPUT.PUT_LINE (
  'source: ' || source.first_name || ' ' || source.last_name
  );

  target := source;

  DBMS_OUTPUT.PUT_LINE(
  'target: '|| target.first || ' ' || target.last
  );
END;
/
```

Example 5-49 Assigning Nested Record to Another Record of Same RECORD Type

```
DECLARE
  TYPE name_rec IS RECORD (
    first   employees.first_name%TYPE,
    last    employees.last_name%TYPE
  );

  TYPE phone_rec IS RECORD (
    name    name_rec,       -- naested record
    email   employees.email%TYPE
  );

  phone_contact phone_rec;
  email_contact email_rec;

BEGIN
  phone_contact.name.first    := 'John';
  phone_contact.name.last     := 'Smith';
```

```
    phone_contact.phone        := '1-650-555-1234';

    email_contact.name     := phone_contact.name;
    email_contact.email := (
       email_contact.name.first || '.' ||
       email_contact.name.last || '@'||
       'example.com'
    );

    DBMS_OUTPUT.PUT_LINE(email_contact.email);
  END;
  /
```

5.13.2 Assigning Full or Partial Rows to Record Variables

If a record variable represents a full or partial row of a database or view, we can assign the represented row to the record variable.

5.13.2.1 Using SELECT INTO to Assign a Row to a Record Variable

Example 5-50 SELECT INTO Assigning Values to Record Variable

```
  DECLARE
    TYPE recordtype IS RECORD (
       last    employees.last_name%TYPE,
       id        employees.employee_id%TYPE
    );

    rec1 record_type;

  BEGIN
    SELECT last_name, employee_id INTO rec1
    FROM employees
    WHERE job_id = 'AD_PRES';

    DBMS_OUTPUT.PUT_LINE('Employee # '|| rec1.id || ' = '|| rec1.last);
  END;
  /
```

5.13.2.2 Using FETCH to Assign a Row to a Record Variable

The syntax of a simple FETCH statement is

FETCH cursor INTO record_variable_name;

Example 5-51 FETCH Assigns Values to Record that Function Returns

```
  DECALRE
    TYPE EmpRecTyp IS RECORD (
       emp_id     employees.employee_id%TYPE,
       salary     employees.salary%TYPE
    );

    CURSOR desc_salary RETURN EmpRecType IS
       SELECT employee_id, salary
       FROM employees
       ORDER BY salary DESC;

    highest_paid_emp    EmpRecTyp;
    next_highest_paid_emp     EmpRecTyp;


    FUNCTION nth_highest_salary (n INTEGER)
    RETURN EmpRecTyp IS
    emp_rec EmpRecTyp;
    BEGIN
       OPEN desc_salary;
```

```
      FOR i IN 1..n LOOP
        FETCH desc_salary INTO emp_rec;
      END LOOP;
      CLOSE desc_salary;
      RETURN emp_rec;
    END nth_highest_salary;

  BEGIN
    highest_paid_emp := nth_highest_salary(1);
    next_highest_paid_emp := nth_highest_salary(2);

    DBMS_PUTPUT.PUT_LINE(
    'Highest Paid: #'||
    highest_paid_emp.emp_id || ', $' ||
    highest_paid_emp.salary
    );

    DBMS_OUTPUT.PUT_LINE(
    'Next Highest Paid: #'||
    next_highest_paid_emp.emp_id || ', $'||
    next_highest_paid_emp.salary
    );
  END;
  /
```

5.13.2.3 Using SQL Statement to Return Rows in PL/SQL Record Variable

The SQL statements INSERT, UPDATE, and DELETE have an optional RETURNING INTO clause that can return the affected row in a PL/SQL record variable.

Example 5-52 UPDATE Statement Assigns Values to Record Variable

```
DECLARE
  TYPE EmpRec IS RECORD (
    last_name    employees.last_name%TYPE,
    salary       employees.salary%TYPE
  );

  emp_info   EmpRec;
  old_salary    employees.salary%TYPE;

BEGIN
  SELECT salary INTO old_salary
  FROM employees
  WHERE employee_id = 100;

  UPDATE employees
  SET salary = salary * 1.1
  WHERE employee_id = 100
  RETURNING last_name, salary INTO emp_info;

  DBMS_OUTPUT.PUT_LINE(
  'Salary of '|| emp_info.last_name || ' raised from ' ||
  old_salary || ' to ' || emp_info.salary
  );
END;
/
```

5.13.3 Assigning NULL to a RECORD Varibale

Assigning the value NULL to a record variable assigns the value NULL to each of its fields.

This assignment is recursive; that is if a field is a record, then its fields are also assigned the value NULL.

Example 5-53 Assigning NULL to Record Variable

```
DECALRE
   TYPE age_rec IS RECORD (
      years    INTEGER DEFAULT 35,
      months    INTEGER DEFAULT 6
   );

   TYPE name_rec IS RECORD (
      first    employees.first_name%TYPE DEFAULT 'John',
      last    employees.last_name%TYPE DEFAULT 'Doe',
      age    age_rec
   );

   name name_rec;

   PROCEDURE print_name as
   BEGIN
      DBMS_OUTPUT.PUT(NVL(name.first, 'NULL')|| ' ' );
      DBMS_OUTPUT.PUT(NVL(name.last, 'NULL')|| ' ' );
      DBMS_OUTPUT.PUT(NVL(TO_CHAR(name.age.years), 'NULL')|| ' yrs ' );
      DBMS_OUTPUT.PUT_LINE(NVL(TO_CHAR(name.age.months), 'NULL')|| ' mos' );
   END print_name;

BEGIN
   print_name;

   name := NULL;
   print_name;
END;
/
```

5.14 Record Comparisons

Records cannot be tested natievely for nullity, equality, or inequality.
These BOOLEAN expressions are illigal:

- my_record IS NULL;
- my_record_1 = my_record_2
- my_record_1 > my_record_2

5.15 Inserting Records into Tables

The PL/SQL extension to the SQL INSERT statement lets us insert a record into a table.

The record must represent a row of the table.
To efficienty insert a collection of records into a table, put the INSERT statement inside a FORALL statement.

Example 5-54 Initializing Table by Inserting Record of Default Values

```
DROP TABLE schedule;
CREATE TABLE schedule (
   week    NUMBER,
   Mon     VARCAHR2(10),
   Tue     VARCAHR2(10),
   Wed     VARCAHR2(10),
   Thu     VARCAHR2(10),
   Fri     VARCAHR2(10),
   Sat     VARCAHR2(10),
   Sun     VARCAHR2(10)
);

DECLARE
   default_week schedule%ROWTYPE;
```

```
  i    NUMBER;
BEGIN
  default_week.Mon := '0800-1700';
  default_week.Tue := '0800-1700';
  default_week.Wed := '0800-1700';
  default_week.Thu := '0800-1700';
  default_week.Fri := '0800-1700';
  default_week.Sat := 'Day Off';
  default_week.Sun := 'Day Off';

  FOR i in 1..6 LOOP
    default_week.week    := i;

    INSERT INTO schedule VALUES default_week;
  END LOOP;
END;
/
```

5.16 Updating Rows with Records

Example 5-55 Updating rows with Record

```
DECLARE
  default_week schedule%ROWTYPE;
  i    NUMBER;
BEGIN
  default_week.Mon := '0900-1800';
  default_week.Tue := '0900-1800';
  default_week.Wed := '0900-1800';
  default_week.Thu := '0900-1800';
  default_week.Fri := '0900-1800';
  default_week.Sat := 'Day Off';
  default_week.Sun := 'Day Off';

  FOR i in 1..3 LOOP
    default_week.week    := i;

    UPDATE schedule
    SET ROW = default_week
    WHERE week = i;
  END LOOP;
END;
/
```

5.17 Restrictions on Record Inserts and Updates

These restrictions apply to record inserts and updates:

- Record variables are allowed only in these places:
  - On the right side of the SET clause in an UPDATE statement
  - In the VALUES clause of an INSERT statement
  - In the INTO subclause of a RETURNING clause

Record variables are not allowed in a SELECT list, WHERE clause, GROUP BY clause, or ORDER BY clause.

- The keyword ROW is allowed only on the left side of a SET clause. Also, you cannot use ROW with a subquery.

- In an UPDATE statement, only one SET clause is allowed if ROW is used.

- If the VALUES clause of an INSERT statement contains a record variable, no other variable or value is allowed in the clause.

- If the INTO subclause of a RETURNING clause contains a record variable, no other variable or value is allowed in the subclause.

- These are not supported:
  - Nested RECORD types
  - Functions that return a RECORD type
  - Record inserts and updates using the EXECUTE IMMEDIATE statement.