**PL/SQL Language Fundamentals**

(Notes #2)

Oracle Database PL/SQL Language Reference. 12c Release 2 (12.2). January 2019

<https://docs.oracle.com/en/database/oracle/oracle-database/12.2/lnpls/database-pl-sql-language-reference.pdf>

**Topics**:

1. Overview of PL/SQL

2. Language (Block) Structure

3. Data Types

4. Declaring Variables

5. Variable Scope

6. The %TYPE attributes

7. PL/SQL Static SQL, %ROWTPE

Appendix: concepts about schema, package

1. **Overview of PL/SQL**

PL/SQL is a **P**rocedural **L**anguage extension to SQL.

It provides the processing power of procedural language.

It offers the combination of power and ease of use:

better performance, less over header in network environment, portable, ...

You can use SQL commands:

insert, update, delete, select, cursor

commit, rollback, savepoint, etc.

plus:

Declare variables

Conditional Control (if then else (exit) goto)

Iteration Control (for... loop, while loop end loop)

Assignment statement: x := y + z ;

But you cannot use create, alter, rename, connect, grant, revoke in static SQL. Except the last class, all we will discuss are about static, not dynamic SQL.

In PL/SQL, all queries must have an INTO clause containing host variables and /or PL/SQL variables that correspond to items in the select list.

In Oracle, stored procedures, functions, database trigger and packages, all the coding is done using PL/SQL.

1. **Language Structure**

PL/SQL is a block-structured language.

A block is a logical section of program which performs a special task.

A block contains three components:

Declaration: starting with (keyword) declare.

Procedure: starting with begin, then executable statements, then end.

Exception: error handling. starting with (keyword) exception.

Nested (sub) block is allowed.

<< label\_name >> (optional)

DECLARE (optional)

-- Declarations of local types, variables, cursors, subprograms

BEGIN(required)

-- Statements (which can use items declared in the declarative part)

[EXCEPTION] -- (optional)

/\* Exception handlers for exceptions (errors) raised in the executable part.

In PL/SQL, an error or warning is called an exception. Errors in the executable

section can be handled in this section \*/

END [label\_name];

Statements terminated by ";". But not for DECLARE, BEGIN, and EXCEPTION.

Comments: /\* Multiline comments \*/

-- only for a single line.

Comparing to stored programs (usually which are created by “create [or replace]” clause), the above codes is called anonymous PL/SQL block (even if it has a label).

1. **PL/SQL Data Types**

Every PL/SQL constant, variable, parameter, and function return value has a data type

that determines its storage format and its valid values and operations.

The PL/SQL data types include the SQL data types. All information about data types, subtypes, data type comparison rules, data conversion, literals, and format models apply to both SQL and PL/SQL, except a few special cases that are noted in Oracle manual, such as the max size of Varchar2 is different, 32,767 bytes in PL/SQL while 4,000 in SQL. etc. (substantially, they are same to ordinary users, as we will never exceed these limits).

**Scalar** data types have no internal components.

The PL/SQL scalar data types are:

The SQL data types (CHAR, VARCHAR2, Date, Raw, Number, etc.)

BOOLEAN (it stores the logical values TRUE and FALSE and the non-value NULL).

Conditional expressions use the logical operators AND, OR and the unary

operator NOT to check the variable values.

Arithmetic, character, and date expressions can be used to return a Boolean value.

PLS\_INTEGER and BINARY\_INTEGER, here we treat them identically:

base type for signed integers. - 231.. 231-1. (-2,147,483,648 .. 2,147,483,647).

predefined subtypes: Natural (0, 231-1), positive (1, 231-1), etc.

They have the advantages over the NUMBER data type and its subtypes:

PLS\_INTEGER values require less storage.

PLS\_INTEGER operations use hardware arithmetic, so they are faster than

NUMBER operations, which use library arithmetic.

(For our course programming, we will not tell the difference; as nothing is slow.)

A **subtype** is a data type that is a subset of another data type, which is its base type.

A subtype has the same valid operations as its base type. A data type and its subtypes comprise a data type family. For examples:

PLS\_INTEGER: it has the subtypes: natural, naturaln, positive, postiven, signtype,

simple\_integer. (page 3-11 manual)

Number: its predefined subtypes: integer, int, dec, decimal, doubleprecision, float,

numeric, real, smallint.

Char: its predefined subtype - Character,

Varchar2: has predefined subtype Varchar and string (same range).

**User-Defined Subtypes**

PL/SQL allows users define their own subtypes. The base type can be any scalar or

user-defined subtypes.

Syntax:

SUBTYPE subtype\_name IS base\_type

**Example:**

Declare

SUBTYPE Balance IS NUMBER;

SUBTYPE Counter IS NATUREAL;

Checking\_account Balance (8,2);

Accounts Counter := 1;

SUBTYPE Word IS CHAR(6);

Verb Word := 'Study';

Composite data type: has internal component: such as PL/SQL Collections and Records, we will discuss these later (Note6a\_Collections, Note6b\_Records).

1. **Declaring Variables**

Unlike SQL, PL/SQL lets users declare variables and constants, and then use them wherever users can use an expression\*. As the program runs, the values of variables can change, but the values of constants cannot.

\* An expression is a combination of one or more values, operators, and SQL functions that evaluate to a value. An expression generally assumes the datatype of its components.

Samples of expressions:

2, 'Adams', (2+3), Upper (emp.ename), To\_char (sysdate, ‘DD-MMM-YY’).

# [Declaration]

A variable declaration always specifies the name and data type of the variable.

For most data types, a variable declaration can also specify an initial value.

variable\_name [constant] Datatype [NOT NULL] := initial\_value;

-- if defined NOT NULL, or constant, the initial value must be defined.

Example of declaration:

DECLARE

part\_number NUMBER(6); -- SQL data type

part\_name VARCHAR2(20); -- SQL data type

in\_stock BOOLEAN; -- PL/SQL-only data type

part\_price NUMBER(6,2); -- SQL data type

part\_description VARCHAR2(50); -- SQL data type

BEGIN

NULL;

END;

\* Variable acts like a container, what it stores can change (but the same data type).

Variables of a special data type are for storing values of that special data type.

\* Names of variables (identifiers) are case-insensitive. (thus, lastname, Lastname,

LASTNAME are the same. ("Quoted User-Defined Identifiers" are exceptions)

\* Do not use ":" to prefix the variable name (refer to note8\_Trigger).

\* Names of variable can be the same of that of the columns, but you are strongly

recommended not to do so.

The local variable name having the same name in the database takes precedence over

the later.

An ordinary user-defined variable name, same as in SQL (not with double quotes):

* Begins with a letter
* Can include letters, digits,
* Can include some special characters ($, #, \_ )
* Limit 128 bytes
* Not a reserved word

**[NOT NULL Constraint]**

The default initial value for a scalar variable is NULL, all the variables declared below are initialized to NULL.

DECLARE

Address VARCHAR2(80);

User\_name VARCHAR2(80);

IS\_valid BOOLEAN;

Counter Integer;

NOT NULL constraint on the variable prevents a null value to that item.

Variable declared with “not null” must be assigned an initial value.

#### acct\_id INTEGER(4) NOT NULL := 9999;

**[Declaring Constants]**

A constant holds a value that does not change, and you must assign the initial value.

#### DECLARE

#### credit\_limit CONSTANT NUMBER(9, 2) := 5000.00; -- SQL data type

#### max\_days\_in\_year CONSTANT INTEGER := 366; -- SQL data type

#### urban\_legend CONSTANT BOOLEAN := FALSE; -- PL/SQL-only data type

**[Initial values of variables and Constants]**

In a variable declaration, the initial value is optional unless you specify the NOT NULL

constraint or Constant.

User may assign an initial value to a variable while declaring it, either use the assignment operator (:=) or the keyword DEFAULT.

Example:

#### DECLARE

#### bonus\_rate CONSTANT NUMBER (3, 2) := 0.10;

#### pi CONSTANT NUMBER := 3.14159;

#### radius NUMBER := 1;

#### area NUMBER(9, 2):= (pi \* radius\*\*2);

#### state VARCHAR2(2) DEFAULT 'IL';

1. **Variables Scope**

The scope of an identifier is that region of a program unit from which you can reference the identifier.

The **visibility** of an identifier is the region of a PL/SQL unit from which you can reference the identifier without qualifying it. An identifier is **local** to the PL/SQL unit that declares it. If that unit has subunits, the identifier is **global** to them.

Rules:

* Identifiers declared in a block are local to that block and global to all its inner

blocks (sub-block).

* An outer block may not reference an identifier declared in one of the its inner block.
* If a global identifier is re-declared in a sub-block, then the local declaration prevails and the sub-block cannot reference the global identifier unless you use a qualified name. (see example below)

DECLARE

-- variables declared in this outer block are global to the nested blocks.

BEGIN

-- Statements (which can use items declared in the declarative part)

-- Program here does not see those will be declared in nested block,

[EXCEPTION]

declare -- starting of inner/nested/enclosed block

/\* variables declared here in this inner block are local to the inner block.

not available for statements in outer block. \*/

BEGIN

Program here sees the variables declared in outer block (like ancestor).

(of course, codes in the inner block can see the variables declared in inner block)

END;

END;

Terms related to nested block:

enclosed block, child block, inner block, sub-block;

The outer block may be called as

enclosing block, parent block.

**Example 1.**

#### SET SERVEROUTPUT ON -- this command is needed to display the output.

-- for one session (login), you only need to run it once.

<<outer\_b>>

DECLARE

x1 integer := 1;

BEGIN

DBMS\_OUTPUT.PUT\_LINE ('x1 = ' || x1 ||'.');

-- This will fail: DBMS\_OUTPUT.PUT\_LINE ('x2 = ' || x2 ||'.');

DECLARE

x2 integer := 2;

x1 integer := 9; -- this inner block “x1” has high priority.

BEGIN

DBMS\_OUTPUT.PUT\_LINE (' x2 = ' || x2 ||'.');

DBMS\_OUTPUT.PUT\_LINE (' x1 not from outer\_b = ' || x1 ||'.');

DBMS\_OUTPUT.PUT\_LINE (' x1 from outer\_b = ' || outer\_b.x1 ||'.');

END;

END outer\_b;

OUTPUT

x1 = 1.

x2 = 2.

x1 not from outer\_b = 9.

x1 from outer\_b = 1.

1. **The %TYPE Attributes**

The %TYPE provides the datatype of a column of a table or a variable (scalar anchoring).

The %TYPE attribute lets users declare a data item of the same data type as a previously

declared variable or column (without knowing what that type is). If the declaration of

the referenced item changes, then the declaration of the referencing item changes

accordingly.

Rather than hard-coding the data type and precision of a variable, user can use the %TYPE attribute to declare a variable according to another previously declared variable or database column. The %TYPE attribute is most often used when the value stored in the variables is derived from a table in the database. User needs not to know the data type of that column, and in case the table column definition changes, there is no need to change the variable. PL/SQL complies the block, it will determine the variable data type.

referencing\_item referenced\_item%TYPE;

#### variable\_name table\_name.column\_name%TYPE;

The referencing item inherits the data type and size, constraints from the referenced item.

If the referenced variable is a not a column of table, and it has the “not null” constraint, then the referencing item inherits the “not null” constraints.

username VARCHAR2 (50) Not Null := 'Jone';

v\_username username%TYPE := 'Bill'; -- it inherits the “not null” constraint

If you do not initialize that variable as below

v\_username username%TYPE;

then you will get an error message.

PLS-00218: a variable declared NOT NULL must have an initialization assignment

If the referenced variable is a column of table with “not null” constraint, then the referencing item does NOT inherit the “not null” constraints.

v\_no emp.empno%TYPE; -- it does NOT inherit the “not null” constraint

#### 

**Example 2.**

#### SET SERVEROUTPUT ON;

DECLARE

emp\_name emp.ename%TYPE;

emp\_no emp.empno%TYPE; -- not inherit the not null constraint

username VARCHAR2 (50) Not Null := 'Jone';

v\_username username%TYPE := 'Bill'; -- required initial value

BEGIN

emp\_name := 'King';

DBMS\_OUTPUT.PUT\_LINE('emp\_name = ' || emp\_name);

emp\_no := 9999;

DBMS\_OUTPUT.PUT\_LINE('emp\_no = ' || emp\_no);

DBMS\_OUTPUT.PUT\_LINE('username = ' || username);

DBMS\_OUTPUT.PUT\_LINE('v\_user\_name = ' || v\_username);

v\_username := 'Tom';

DBMS\_OUTPUT.PUT\_LINE('v\_user\_name = ' || v\_username);

END;

Output

emp\_name = King

emp\_no = 9999

username = Jone

v\_user\_name = Bill

v\_user\_name = Tom

1. **PL/SQL Static SQL**

Static SQLis a PL/SQL feature that allows SQL syntax directly in a PL/SQL statement.

In a PL/SQL block, you may use SQL statements to retrieve (SELECT INTO) data, modify data from database table (DML, INSERT, UPDATE, DELETE), and use transaction control commands (COMMIT, ROLLBACK, SAVEPOINT, etc.).

PL/SQL does not directly support data definition language (DDL) statements, such as CREATE TABLE, ALTER TABLE, and DROP TABLE, neither support data control language (DCL) statement such as GRANT and REVOKE. Later (in the last class) we will discuss about the dynamic SQL which may execute these commands.

**Example 3.** (using insert, update, delete in PL/SQL block)

desc employees

-- DROP TABLE employees\_temp;

-- skip this line if it is the first time to run the commands below

CREATE TABLE employees\_temp AS

SELECT employee\_id, first\_name, last\_name

FROM employees

WHERE employee\_id < 106;

SELECT \* from employees\_temp;

--

DECLARE

emp\_id employees\_temp.employee\_id%TYPE := 299;

emp\_first\_name employees\_temp.first\_name%TYPE := 'Bob';

emp\_last\_name employees\_temp.last\_name%TYPE := 'Henry';

BEGIN

INSERT INTO employees\_temp (employee\_id, first\_name, last\_name)

VALUES (999, 'Michael', 'Washington');

INSERT INTO employees\_temp (employee\_id, first\_name, last\_name)

VALUES (emp\_id, emp\_first\_name, emp\_last\_name);

DBMS\_OUTPUT.PUT\_LINE ('New record employee with ID = '

|| emp\_id || ', First name is ' || emp\_first\_name || ', Last name is ' || emp\_last\_name || '.');

UPDATE employees\_temp

SET first\_name = 'Robert'

WHERE employee\_id = emp\_id;

DBMS\_OUTPUT.PUT\_LINE ('After UPDATE, employee with ID = '

|| emp\_id || ', First name changed to Robert, ' || ' Last name is still as ' || emp\_last\_name ||'.');

DELETE FROM employees\_temp

WHERE employee\_id = emp\_id

**RETURNING** first\_name, last\_name **-- PL/SQL Special**

**INTO** emp\_first\_name, emp\_last\_name;

-- Manual example 5-50, page 5-61, UPDATE - RETURNING - INTO

DBMS\_OUTPUT.PUT\_LINE ('After DELETION: the record deleted had emp\_id = ' || emp\_id || ', First name: '

|| emp\_first\_name || ', Last name ' || emp\_last\_name ||'.');

END;

RESULT:

New record employee with ID = 299, First name is Bob, Last name is Henry.

After UPDATE, employee with ID = 299, First name changed to Robert, Last name is still as Henry.

After DELETION: the record deleted had emp\_id = 299, First name: Robert, Last name Henry.

**Processing Query Result** (SELECT)

If you expect the query to return only ***one row***, then use the SELECT INTO statement to

store values from that row in either one or more scalar variables, or one record variable.

The order of the columns and the order of the variables must match when using the SELECT INTO clause.

If the query return two or more than two rows, then the explicit cursor is needed, we will discuss that soon.

Basic syntax:

SELECT column1, column2, ...

**INTO** variable\_name1, variable\_name2 ...

FROM ... WHERE ...;

**Example 4.** using table employees (SELECT INTO for one row returned)

DECLARE

emp\_fname employees.first\_name%TYPE;

emp\_lname employees.last\_name%TYPE;

emp\_boss employees.manager\_id%TYPE;

BEGIN

SELECT first\_name, last\_name, manager\_id

**INTO** emp\_fname, emp\_lname, emp\_boss -- pay attention to “INTO”

FROM employees

WHERE **employee\_id = 102**; -- condition for unique row returned

DBMS\_OUTPUT.PUT\_LINE ('For employee with ID 102, his name is ' || emp\_fname || ' ' || emp\_lname ||

', his manager id is ' || emp\_boss || '.');

END;

Result

For employee with ID 102, his name is Lex De Haan, his manager id is 100.

If the variable data type is not correct to the correspondent select item, or

if there is more than one row selected, or

if there is no row selected, an Error will be reported by system.

**Example 5.** Some change: join two tables.

DECLARE

emp\_fname employees.first\_name%TYPE;

emp\_lname employees.last\_name%TYPE;

emp\_boss employees.last\_name%TYPE;

BEGIN

SELECT e.first\_name, e.last\_name, eb.last\_name

INTO emp\_fname, emp\_lname, emp\_boss

FROM employees e, employees eb

WHERE **e.employee\_id = 102** and e.manager\_id = eb.employee\_id;

-- condition for unique row returned

DBMS\_OUTPUT.PUT\_LINE ('For employee with ID 102, his name is ' || emp\_fname || ' ' || emp\_lname || ', his manager name is ' ||

emp\_boss || '.');

END;

Result

For employee with ID 102, his name is Lex De Haan, his manager name is King.

**Example 6.** Using %**ROWTYPE** variables, that will be discussed in details in later class (note4\_cursor, section 2.3, page 7, record anchoring).

DECLARE

r\_emp employees%ROWTYPE; -- type as record of table employees

BEGIN

SELECT \* INTO r\_emp -- variable r\_emp consists of all columns

FROM employees

WHERE employee\_id = 102; -- unique key guarantees one record returned only

DBMS\_OUTPUT.PUT\_LINE ('For the employee with ID =102, his name is ' || r\_emp.first\_name || ' ' || r\_emp.last\_name || '.');

-- Use dot notation qualifier to access columns of row variable

END;

Result

For the employee with ID =102, his name is Lex De Haan.

**Appendix**

**[Schema]**

Technically, a schema is a collection of database objects owned by a specific user.

Those objects include tables, indexes, views, stored procedures, etc.

In Oracle, a schema requires a user to be created. But you can create a user that has no schema (i.e., no objects). So, in Oracle, the user is the account and the schema is the collection of objects.

In SQL, you can find who you are with the commands;  
show user

SELECT user from dual;

output

USER is "CZHANG"

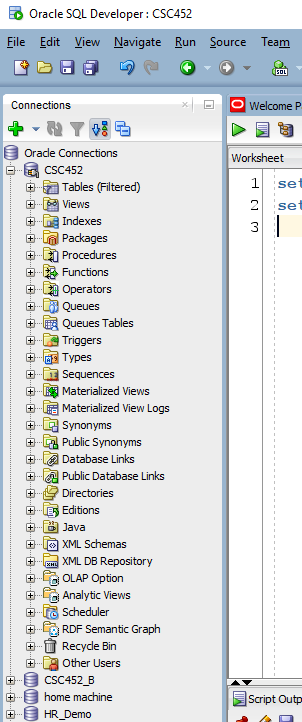
USER

-------

CZHANG

For example, my oracle user name is “czhang”. I created a connection named “CSC452”. Under that directory, I can see the tables, indexes, functions, views I created. The collection of these objects under my account is called a schema.

Below is a screenshot of my SQL\*Developer.



**[Package]**

Excerpt from Note9\_package

A **package** is a schema object that groups logically related PL/SQL types, variables, constants, subprograms, cursors, and exceptions.

A package is compiled and stored in the database, where many applications can share its contents.

Packages usually have two parts, a specification and a body, although sometimes the body is unnecessary (if there is no cursor or subprogram declared).