

PLSQL Notes

The PL/SQL programming language was developed by Oracle Corporation in the late 1980s as procedural extension language for SQL and the Oracle relational database. Following are certain notable facts about PL/SQL –

- PL/SQL is a completely portable, high-performance transaction-processing language.
- PL/SQL provides a built-in, interpreted and OS independent programming environment.
- PL/SQL can also directly be called from the command-line **SQL*Plus interface**.
- Direct call can also be made from external programming language calls to database.
- PL/SQL's general syntax is based on that of ADA and Pascal programming language.
- Apart from Oracle, PL/SQL is available in **TimesTen in-memory database** and **IBM DB2**.

Features of PL/SQL

PL/SQL has the following features –

- PL/SQL is tightly integrated with SQL.
- It offers extensive error checking.
- It offers numerous data types.
- It offers a variety of programming structures.
- It supports structured programming through functions and procedures.
- It supports object-oriented programming.
- It supports the development of web applications and server pages.

Advantages of PL/SQL

PL/SQL has the following advantages –

- SQL is the standard database language and PL/SQL is strongly integrated with SQL. PL/SQL supports both static and dynamic SQL. Static SQL supports DML operations and transaction control from PL/SQL block. In Dynamic SQL, SQL allows embedding DDL statements in PL/SQL blocks.
- PL/SQL allows sending an entire block of statements to the database at one time. This reduces network traffic and provides high performance for the applications.
- PL/SQL gives high productivity to programmers as it can query, transform, and update data in a database.
- PL/SQL saves time on design and debugging by strong features, such as exception handling, encapsulation, data hiding, and object-oriented data types.
- Applications written in PL/SQL are fully portable.
- PL/SQL provides high security level.
- PL/SQL provides access to predefined SQL packages.

- PL/SQL provides support for Object-Oriented Programming.
- PL/SQL provides support for developing Web Applications and Server Pages.

Every PL/SQL statement ends with a semicolon (;). PL/SQL blocks can be nested within other PL/SQL blocks using **BEGIN** and **END**. Following is the basic structure of a PL/SQL block –

```
DECLARE
    <declarations section>
BEGIN
    <executable command(s)>
EXCEPTION
    <exception handling>
END;
```

The 'Hello World' Example

```
DECLARE
    message varchar2(20) := 'Hello, World!';
BEGIN
    dbms_output.put_line(message);
END;

/
```

Comments

```
DECLARE
    -- variable declaration
    message varchar2(20) := 'Hello, World!';
BEGIN
    /*
    *   PL/SQL executable statement(s)
    */
    dbms_output.put_line(message);

END;

/
```

Following is a valid declaration –

```
DECLARE
    num1 INTEGER;
    num2 REAL;
    num3 DOUBLE PRECISION;
BEGIN
    null;
END;

/
```

PL/SQL User-Defined Subtypes

A subtype is a subset of another data type, which is called its base type. A subtype has the same valid operations as its base type, but only a subset of its valid values.

PL/SQL predefines several subtypes in package **STANDARD**. For example, PL/SQL predefines the subtypes **CHARACTER** and **INTEGER** as follows –

```
SUBTYPE CHARACTER IS CHAR;
SUBTYPE INTEGER IS NUMBER(38,0);
```

You can define and use your own subtypes. The following program illustrates defining and using a user-defined subtype –

```
DECLARE
    SUBTYPE name IS char(20);
    SUBTYPE message IS varchar2(100);
    salutation name;
    greetings message;
BEGIN
    salutation := 'Reader ';
    greetings := 'Welcome to the World of PL/SQL';
    dbms_output.put_line('Hello ' || salutation || greetings);
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result –

```
Hello Reader Welcome to the World of PL/SQL
```

```
PL/SQL procedure successfully completed.
```

Variable Declaration in PL/SQL

PL/SQL variables must be declared in the declaration section or in a package as a global variable. When you declare a variable, PL/SQL allocates memory for the variable's value and the storage location is identified by the variable name.

The syntax for declaring a variable is –

```
variable_name [CONSTANT] datatype [NOT NULL] [:= | DEFAULT initial_value]
```

Where, *variable_name* is a valid identifier in PL/SQL, *datatype* must be a valid PL/SQL data type or any user defined data type which we already have discussed in the last chapter. Some valid variable declarations along with their definition are shown below –

```
sales number(10, 2);
pi CONSTANT double precision := 3.1415;
name varchar2(25);
address varchar2(100);
```

When you provide a size, scale or precision limit with the data type, it is called a **constrained declaration**. Constrained declarations require less memory than unconstrained declarations. For example –

```
sales number(10, 2);
name varchar2(25);
address varchar2(100);
```

Initializing Variables in PL/SQL

Whenever you declare a variable, PL/SQL assigns it a default value of NULL. If you want to initialize a variable with a value other than the NULL value, you can do so during the declaration, using either of the following –

- The **DEFAULT** keyword
- The **assignment** operator

For example –

```
counter binary_integer := 0;
greetings varchar2(20) DEFAULT 'Have a Good Day';
```

You can also specify that a variable should not have a **NULL** value using the **NOT NULL** constraint. If you use the NOT NULL constraint, you must explicitly assign an initial value for that variable.

It is a good programming practice to initialize variables properly otherwise, sometimes programs would produce unexpected results. Try the following example which makes use of various types of variables –

```
DECLARE
    a integer := 10;
    b integer := 20;
    c integer;
    f real;
BEGIN
    c := a + b;
    dbms_output.put_line('Value of c: ' || c);
    f := 70.0/3.0;
    dbms_output.put_line('Value of f: ' || f);
END;
/
```

When the above code is executed, it produces the following result –

```
Value of c: 30
Value of f: 23.3333333333333333
```

PL/SQL procedure successfully completed.

Variable Scope in PL/SQL

PL/SQL allows the nesting of blocks, i.e., each program block may contain another inner block. If a variable is declared within an inner block, it is not accessible to the outer block. However, if a variable is declared and accessible to an outer block, it is also accessible to all nested inner blocks. There are two types of variable scope –

- **Local variables** – Variables declared in an inner block and not accessible to outer blocks.

- **Global variables** – Variables declared in the outermost block or a package.

Following example shows the usage of **Local** and **Global** variables in its simple form –

```
DECLARE
    -- Global variables
    num1 number := 95;
    num2 number := 85;
BEGIN
    dbms_output.put_line('Outer Variable num1: ' || num1);
    dbms_output.put_line('Outer Variable num2: ' || num2);
    DECLARE
        -- Local variables
        num1 number := 195;
        num2 number := 185;
    BEGIN
        dbms_output.put_line('Inner Variable num1: ' || num1);
        dbms_output.put_line('Inner Variable num2: ' || num2);
    END;
END;
/
```

Assigning SQL Query Results to PL/SQL Variables

You can use the **SELECT INTO** statement of SQL to assign values to PL/SQL variables. For each item in the **SELECT list**, there must be a corresponding, type-compatible variable in the **INTO list**. The following example illustrates the concept. Let us create a table named CUSTOMERS –

The following program assigns values from the above table to PL/SQL variables using the **SELECT INTO clause** of SQL –

```
DECLARE
    c_id customers.id%type := 1;
    c_name customers.name%type;
    c_addr customers.address%type;
    c_sal customers.salary%type;
BEGIN
    SELECT name, address, salary INTO c_name, c_addr, c_sal
    FROM customers
    WHERE id = c_id;
    dbms_output.put_line
    ('Customer ' || c_name || ' from ' || c_addr || ' earns ' || c_sal);
END;
/
```

When the above code is executed, it produces the following result –

Customer Ramesh from Ahmedabad earns 2000

PL/SQL procedure completed successfully

Declaring a Constant

A constant is declared using the **CONSTANT** keyword. It requires an initial value and does not allow that value to be changed. For example –

```
PI CONSTANT NUMBER := 3.141592654;
DECLARE
    -- constant declaration
    pi constant number := 3.141592654;
    -- other declarations
    radius number(5,2);
    dia number(5,2);
    circumference number(7, 2);
    area number (10, 2);
BEGIN
    -- processing
    radius := 9.5;
    dia := radius * 2;
    circumference := 2.0 * pi * radius;
    area := pi * radius * radius;
    -- output
    dbms_output.put_line('Radius: ' || radius);
    dbms_output.put_line('Diameter: ' || dia);
    dbms_output.put_line('Circumference: ' || circumference);
    dbms_output.put_line('Area: ' || area);
END;
/
```

Labeling a PL/SQL Loop

PL/SQL loops can be labeled. The label should be enclosed by double angle brackets (<< and >>) and appear at the beginning of the LOOP statement. The label name can also appear at the end of the LOOP statement. You may use the label in the EXIT statement to exit from the loop.

The following program illustrates the concept –

```
DECLARE
    i number(1);
    j number(1);
BEGIN
    << outer_loop >>
    FOR i IN 1..3 LOOP
        << inner_loop >>
        FOR j IN 1..3 LOOP
            dbms_output.put_line('i is: ' || i || ' and j is: ' || j);
        END loop inner_loop;
    END loop outer_loop;
END;
/
```

Declaring String Variables

Oracle database provides numerous string datatypes, such as CHAR, NCHAR, VARCHAR2, NVARCHAR2, CLOB, and NCLOB. The datatypes prefixed with an 'N' are '**national character set**' datatypes, that store Unicode character data.

If you need to declare a variable-length string, you must provide the maximum length of that string. For example, the VARCHAR2 data type. The following example illustrates declaring and using some string variables –

```
DECLARE
    name varchar2(20);
    company varchar2(30);
    introduction clob;
    choice char(1);
BEGIN
    name := 'John Smith';
    company := 'Infotech';
    introduction := ' Hello! I'm John Smith from Infotech.';
    choice := 'y';
    IF choice = 'y' THEN
        dbms_output.put_line(name);
        dbms_output.put_line(company);
        dbms_output.put_line(introduction);
    END IF;
END;
/
```

PL/SQL String Functions and Operators

PL/SQL offers the concatenation operator (||) for joining two strings. The following table provides the string functions provided by PL/SQL –

```
DECLARE
    greetings varchar2(11) := 'hello world';
BEGIN
    dbms_output.put_line(UPPER(greetings));

    dbms_output.put_line(LOWER(greetings));

    dbms_output.put_line(INITCAP(greetings));

    /* retrieve the first character in the string */
    dbms_output.put_line ( SUBSTR (greetings, 1, 1));

    /* retrieve the last character in the string */
    dbms_output.put_line ( SUBSTR (greetings, -1, 1));

    /* retrieve five characters,
       starting from the seventh position. */
    dbms_output.put_line ( SUBSTR (greetings, 7, 5));

    /* retrieve the remainder of the string,
       starting from the second position. */
    dbms_output.put_line ( SUBSTR (greetings, 2));

    /* find the location of the first "e" */
    dbms_output.put_line ( INSTR (greetings, 'e'));
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result –

```
HELLO WORLD
hello world
```

```
Hello World
h
d
World
ello World
2
```

PL/SQL procedure successfully completed.

Creating a Procedure

A procedure is created with the **CREATE OR REPLACE PROCEDURE** statement. The simplified syntax for the CREATE OR REPLACE PROCEDURE statement is as follows –

```
CREATE [OR REPLACE] PROCEDURE procedure_name
[(parameter_name [IN | OUT | IN OUT] type [, ...])]
{IS | AS}
BEGIN
    < procedure_body >
END procedure_name;
```

Where,

- *procedure-name* specifies the name of the procedure.
- [OR REPLACE] option allows the modification of an existing procedure.
- The optional parameter list contains name, mode and types of the parameters. **IN** represents the value that will be passed from outside and OUT represents the parameter that will be used to return a value outside of the procedure.
- *procedure-body* contains the executable part.
- The AS keyword is used instead of the IS keyword for creating a standalone procedure.

Example

The following example creates a simple procedure that displays the string 'Hello World!' on the screen when executed.

```
CREATE OR REPLACE PROCEDURE greetings
AS
BEGIN
    dbms_output.put_line('Hello World!');
END;
/
```

When the above code is executed using the SQL prompt, it will produce the following result –

Procedure created.

Executing a Standalone Procedure

A standalone procedure can be called in two ways –

- Using the **EXECUTE** keyword
- Calling the name of the procedure from a PL/SQL block

The above procedure named '**greetings**' can be called with the EXECUTE keyword as –


```
EXECUTE greetings;
```

The above call will display –

Hello World

PL/SQL procedure successfully completed.

The procedure can also be called from another PL/SQL block –

```
BEGIN
    greetings;
END;
/
```

The above call will display –

Hello World

PL/SQL procedure successfully completed.

In this chapter, we will discuss the functions in PL/SQL. A function is same as a procedure except that it returns a value. Therefore, all the discussions of the previous chapter are true for functions too.

Creating a Function

A standalone function is created using the **CREATE FUNCTION** statement. The simplified syntax for the **CREATE OR REPLACE PROCEDURE** statement is as follows –

```
CREATE [OR REPLACE] FUNCTION function_name
[(parameter_name [IN | OUT | IN OUT] type [, ...])]
RETURN return_datatype
{IS | AS}
BEGIN
    < function_body >
END [function_name];
```

Where,

- *function-name* specifies the name of the function.
- [OR REPLACE] option allows the modification of an existing function.
- The optional parameter list contains name, mode and types of the parameters. IN represents the value that will be passed from outside and OUT represents the parameter that will be used to return a value outside of the procedure.
- The function must contain a **return** statement.
- The *RETURN* clause specifies the data type you are going to return from the function.
- *function-body* contains the executable part.
- The AS keyword is used instead of the IS keyword for creating a standalone function.

Example

The following example illustrates how to create and call a standalone function. This function returns the total number of CUSTOMERS in the customers table.

```
CREATE OR REPLACE FUNCTION totalCustomers
RETURN number IS
    total number(2) := 0;
BEGIN
    SELECT count(*) into total
```

```

        FROM customers;

        RETURN total;
END;
/

```

When the above code is executed using the SQL prompt, it will produce the following result

—

```
Function created.
```

Calling a Function

While creating a function, you give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task. When a program calls a function, the program control is transferred to the called function.

A called function performs the defined task and when its return statement is executed or when the **last end statement** is reached, it returns the program control back to the main program.

To call a function, you simply need to pass the required parameters along with the function name and if the function returns a value, then you can store the returned value. Following program calls the function **totalCustomers** from an anonymous block —

```

DECLARE
    c number(2);
BEGIN
    c := totalCustomers();
    dbms_output.put_line('Total no. of Customers: ' || c);
END;
/

```

Example

The following example demonstrates Declaring, Defining, and Invoking a Simple PL/SQL Function that computes and returns the maximum of two values.

```

DECLARE
    a number;
    b number;
    c number;
FUNCTION findMax(x IN number, y IN number)
RETURN number
IS
    z number;
BEGIN
    IF x > y THEN
        z := x;
    ELSE
        z := y;
    END IF;
    RETURN z;
END;
BEGIN
    a := 23;
    b := 45;
    c := findMax(a, b);

```

```
        dbms_output.put_line(' Maximum of (23,45): ' || c);
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result –

Maximum of (23,45): 45

Implicit Cursors

The following program will update the table and increase the salary of each customer by 500 and use the **SQL%ROWCOUNT** attribute to determine the number of rows affected –

```
DECLARE
    total_rows number(2);
BEGIN
    UPDATE customers
    SET salary = salary + 500;
    IF sql%notfound THEN
        dbms_output.put_line('no customers selected');
    ELSIF sql%found THEN
        total_rows := sql%rowcount;
        dbms_output.put_line( total_rows || ' customers selected ');
    END IF;
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result –

6 customers selected

PL/SQL procedure successfully completed.

Explicit Cursors

Explicit cursors are programmer-defined cursors for gaining more control over the **context area**. An explicit cursor should be defined in the declaration section of the PL/SQL Block. It is created on a SELECT Statement which returns more than one row.

The syntax for creating an explicit cursor is –

```
CURSOR cursor_name IS select_statement;
```

Working with an explicit cursor includes the following steps –

- Declaring the cursor for initializing the memory
- Opening the cursor for allocating the memory
- Fetching the cursor for retrieving the data
- Closing the cursor to release the allocated memory

Declaring the Cursor

Declaring the cursor defines the cursor with a name and the associated SELECT statement. For example –

```
CURSOR c_customers IS
    SELECT id, name, address FROM customers;
```

Opening the Cursor

Opening the cursor allocates the memory for the cursor and makes it ready for fetching the rows returned by the SQL statement into it. For example, we will open the above defined cursor as follows –

```
OPEN c_customers;
```

Fetching the Cursor

Fetching the cursor involves accessing one row at a time. For example, we will fetch rows from the above-opened cursor as follows –

```
FETCH c_customers INTO c_id, c_name, c_addr;
```

Closing the Cursor

Closing the cursor means releasing the allocated memory. For example, we will close the above-opened cursor as follows –

```
CLOSE c_customers;
```

Example

Following is a complete example to illustrate the concepts of explicit cursors &minua;

```
DECLARE
    c_id customers.id%type;
    c_name customers.No.ame%type;
    c_addr customers.address%type;
    CURSOR c_customers is
        SELECT id, name, address FROM customers;
BEGIN
    OPEN c_customers;
    LOOP
        FETCH c_customers into c_id, c_name, c_addr;
        EXIT WHEN c_customers%notfound;
        dbms_output.put_line(c_id || ' ' || c_name || ' ' || c_addr);
    END LOOP;
    CLOSE c_customers;
END;
/
```

Syntax for Exception Handling

The general syntax for exception handling is as follows. Here you can list down as many exceptions as you can handle. The default exception will be handled using ***WHEN others THEN*** –

```
DECLARE
```

```

    <declarations section>
BEGIN
    <executable command(s)>
EXCEPTION
    <exception handling goes here >
    WHEN exception1 THEN
        exception1-handling-statements
    WHEN exception2 THEN
        exception2-handling-statements
    WHEN exception3 THEN
        exception3-handling-statements
    .....
    WHEN others THEN
        exception3-handling-statements
END;

```

Example

Let us write a code to illustrate the concept. We will be using the CUSTOMERS table we had created and used in the previous chapters –

```

DECLARE
    c_id customers.id%type := 8;
    c_name customers.Name%type;
    c_addr customers.address%type;
BEGIN
    SELECT name, address INTO c_name, c_addr
    FROM customers
    WHERE id = c_id;
    DBMS_OUTPUT.PUT_LINE ('Name: ' || c_name);
    DBMS_OUTPUT.PUT_LINE ('Address: ' || c_addr);

EXCEPTION
    WHEN no_data_found THEN
        dbms_output.put_line('No such customer!');
    WHEN others THEN
        dbms_output.put_line('Error!');
END;
/

```

When the above code is executed at the SQL prompt, it produces the following result –

No such customer!

PL/SQL procedure successfully completed.