**PL/SQL** (Procedural Language/Structured Query Language) is Oracle Corporation's procedural extension language for SQL and the Oracle relational database. PL/SQL's syntax resembles that of Ada.

PL/SQL is one of three key language paradigms embedded in the Oracle Database, along with SQL itself and also Java.

Introduction

PL/SQL supports variables, conditions, loops and exceptions. Arrays are also supported, though in a somewhat unusual way, involving the use of PL/SQL collections. PL/SQL collections are a slightly advanced topic.

Implementations from version 8 of Oracle Database onwards have included features associated with object-orientation.

PL/SQL program units (essentially code containers) can be compiled into the Oracle database. Programmers can thus embed PL/SQL units of functionality into the database directly. They also can write scripts containing PL/SQL program units that can be read into the database using the Oracle SQL\*Plus tool.

Once the program units have been stored into the database, they become available for execution at a later time.

While programmers can readily embed Data Manipulation Language (DML) statements directly into their PL/SQL code using straight forward SQL statements, Data Definition Language(DDL) requires more complex "Dynamic SQL" statements to be written in the PL/SQL code. However, DML statements underpin the majority of PL/SQL code in typical software applications.

In the case of PL/SQL dynamic SQL, early versions of the Oracle Database required the use of a complicated Oracle DBMS\_SQL package library. More recent versions have however introduced a simpler "Native Dynamic SQL", along with an associated EXECUTE IMMEDIATE syntax.

Oracle Corporation customarily extends package functionality with each successive release of the Oracle Database.

PL/SQL program units

**Anonymous Blocks**

Anonymous blocks form the basis of the simplest PL/SQL code, and have the following structure:

<<label>>

DECLARE

TYPE / item / FUNCTION / PROCEDURE declarations

BEGIN

Statements

EXCEPTION

EXCEPTION handlers

END label;

The <<label>> and the DECLARE and EXCEPTION sections are optional.

Exceptions, errors which arise during the execution of the code, have one of two types:

1. Predefined exceptions
2. User-defined exceptions.

User-defined exceptions are always raised explicitly by the programmers, using the RAISE or RAISE\_APPLICATION\_ERROR commands, in any situation where they have determined that it is impossible for normal execution to continue. RAISE command has the syntax:

RAISE <exception name>;

Oracle Corporation has pre-defined several exceptions like NO\_DATA\_FOUND, TOO\_MANY\_ROWS, *etc.* Each exception has a SQL Error Number and SQL Error Message associated with it. Programmers can access these by using the SQLCODE and SQLERRM functions.

The DECLARE section defines and (optionally) initialises variables. If not initialised specifically, they default to NULL.

The major datatypes in PL/SQL include NUMBER, INTEGER, CHAR, VARCHAR2, DATE, TIMESTAMP, TEXT *etc.*

**Functions**

Functions in PL/SQL are a collection of SQL and PL/SQL statements that perform a task and should return a value to the calling environment.

CREATE OR REPLACE FUNCTION <function\_name> [(input/output variable declarations)] RETURN return\_type

<IS|AS>

[declaration block]

BEGIN

<PL/SQL block WITH RETURN statement>

[EXCEPTION

EXCEPTION block]

END;

There are three types of parameter: IN, OUT and IN OUT. An IN parameter is used an input only. An IN parameter cannot be changed by the called program. An OUT parameter is initially NULL. The program assigns the parameter a value and that value is returned to the calling program. An IN OUT parameter may or may not have an initial value. That initial value may or may not be modified by the called program. Any changes made to the parameter are returned to the calling program.

**Procedures**

Procedures are the same as Functions, in that they are also used to perform some task with the difference being that procedures cannot be used in a SQL statement and although they can have multiple out parameters they do not return a value.

Procedures are traditionally the workhorse of the coding world and functions are traditionally the smaller, more specific pieces of code. PL/SQL maintains many of the distinctions between functions and procedures found in many general-purpose programming languages, but in addition, functions can be called from SQL, while procedures cannot.

**Packages**

Packages are groups of conceptually linked Functions, Procedures,Variable,Constants & Cursors etc. The use of packages promotes re-use of code. Packages usually have two parts, a specification and a body, although sometimes the body is unnecessary. The specification (spec for short) is the interface to your applications; it declares the types, variables, constants, exceptions, cursors, and subprograms available for use. The body fully defines cursors and subprograms, and so implements the spec.

Variables

**Numeric variables**

variable\_name NUMBER(P[,S]) := VALUE;

To define a numeric variable, the programmer appends the variable type **NUMBER** to the name definition. To specify the (optional) precision(P) and the (optional) scale (S), one can further append these in round brackets, separated by a comma. ("Precision" in this context refers to the number of digits which the variable can hold, "scale" refers to the number of digits which can follow the decimal point.)

A selection of other datatypes for numeric variables would include: binary\_float, binary\_double, dec, decimal, double precision, float, integer, int, numeric, real, smallint, binary\_integer

**Character variables**

variable\_name varchar2(L) := 'Text';

To define a character variable, the programmer normally appends the variable type VARCHAR2 to the name definition. There follows in brackets the maximum number of characters which the variable can store.

Other datatypes for character variables include:

varchar, char, long, raw, long raw, nchar, nchar2, clob, blob, bfile

**Date variables**

variable\_name date := '01-Jan-2005';

Oracle provides a number of data types that can store dates (DATE, DATETIME, TIMESTAMP *etc.*), however DATE is most commonly used.

Programmers define date variables by appending the datatype code "DATE" to a variable name. The TO\_DATE function can be used to convert strings to date values. The function converts the first quoted string into a date, using as a definition the second quoted string, for example:

TO\_DATE('31-12-2004','dd-mm-yyyy')

or

TO\_DATE ('31-Dec-2004','dd-mon-yyyy', 'NLS\_DATE\_LANGUAGE = American')

To convert the dates to strings one uses the function TO\_CHAR (date\_string, format\_string).

PL/SQL also supports the use of ANSI date and interval literals.[3] The following clause gives an 18-month range:

WHERE dateField BETWEEN DATE '2004-12-31' - INTERVAL '1-6' YEAR TO MONTH

AND DATE '2004-12-31'

**Datatypes for specific columns**

Variable\_name Table\_name.Column\_name%type;

This syntax defines a variable of the type of the referenced column on the referenced tables.

Programmers specify user-defined datatypes with the syntax:

type data\_type is record (field\_1 type\_1 :=xyz, field\_2 type\_2 :=xyz, ..., field\_n type\_n :=xyz);

Conditional Statements

The following code segment shows the IF-THEN-ELSIF construct. The ELSIF and ELSE parts are optional so it is possible to create simpler IF-THEN or, IF-THEN-ELSE constructs.

IF x = 1 THEN

sequence\_of\_statements\_1;

ELSIF x = 2 THEN

sequence\_of\_statements\_2;

ELSIF x = 3 THEN

sequence\_of\_statements\_3;

ELSIF x = 4 THEN

sequence\_of\_statements\_4;

ELSIF x = 5 THEN

sequence\_of\_statements\_5;

ELSE

sequence\_of\_statements\_N;

END IF;

The CASE statement simplifies some large IF-THEN-ELSE structures.

CASE

WHEN x = 1 THEN sequence\_of\_statements\_1;

WHEN x = 2 THEN sequence\_of\_statements\_2;

WHEN x = 3 THEN sequence\_of\_statements\_3;

WHEN x = 4 THEN sequence\_of\_statements\_4;

WHEN x = 5 THEN sequence\_of\_statements\_5;

ELSE sequence\_of\_statements\_N;

END CASE;

CASE statement can be used with predefined selector:

CASE x

WHEN 1 THEN sequence\_of\_statements\_1;

WHEN 2 THEN sequence\_of\_statements\_2;

WHEN 3 THEN sequence\_of\_statements\_3;

WHEN 4 THEN sequence\_of\_statements\_4;

WHEN 5 THEN sequence\_of\_statements\_5;

ELSE sequence\_of\_statements\_N;

END CASE;

Array handling

PL/SQL refers to arrays as "collections". The language offers three types of collections:

1. Index-by tables (associative arrays)
2. Nested tables
3. Varrays (variable-size arrays)

Programmers must specify an upper limit for varrays, but need not for index-by tables or for nested tables. The language includes several collection methods used to manipulate collection elements: for example FIRST, LAST, NEXT, PRIOR, EXTEND, TRIM, DELETE, etc. Index-by tables can be used to simulate associative arrays, as in this example of a memo function for Ackermann's function in PL/SQL.

Looping

As a procedural language by definition, PL/SQL provides several iteration constructs, including basic LOOP statements, WHILE loops, FOR loops, and Cursor FOR loops.

**LOOP statements**

Syntax:

<<parent\_loop>>

LOOP

statements

<<child\_loop>>

LOOP

statements

EXIT parent\_loop WHEN <condition>; *-- Terminates both loops*

EXIT WHEN <condition>; *-- Returns control to parent\_loop*

END LOOP;

EXIT WHEN <condition>;

END LOOP parent\_loop;

Loops can be terminated by using the EXIT keyword, or by raising an exception.

**FOR loops**

**Cursor FOR loops**

FOR RecordIndex IN (SELECT person\_code FROM people\_table)

LOOP

DBMS\_OUTPUT.PUT\_LINE(RecordIndex.person\_code);

END LOOP;

Cursor-for loops automatically open a cursor, read in their data and close the cursor again

As an alternative, the PL/SQL programmer can pre-define the cursor's SELECT-statement in advance in order (for example) to allow re-use or to make the code more understandable (especially useful in the case of long or complex queries).

DECLARE

CURSOR cursor\_person IS

SELECT person\_code FROM people\_table;

BEGIN

FOR RecordIndex IN cursor\_person

LOOP

DBMS\_OUTPUT.PUT\_LINE(RecordIndex.person\_code);

END LOOP;

END;

The concept of the person\_code within the FOR-loop gets expressed with dot-notation ("."):

RecordIndex.person\_code

**Example**

DECLARE

var NUMBER;

BEGIN

*/\*N.B. for loop variables in pl/sql are new declarations, with scope only inside the loop \*/*

FOR var IN 0 .. 10 LOOP

DBMS\_OUTPUT.put\_line(var);

END LOOP;

IF (var IS NULL) THEN

DBMS\_OUTPUT.put\_line('var is null');

ELSE

DBMS\_OUTPUT.put\_line('var is not null');

END IF;

END;

Output:

0

1

2

3

4

5

6

7

8

9

10

var is null

|  |
| --- |
| 1 UPDATE statement can be used within PL/SQL programs to update a  row or a set of rows |
| |  | | --- | |  | |
| |  | | --- | | SQL> SQL> -- **create**demo table SQL> **create**table Employee(   2    ID                 VARCHAR2(4 BYTE)         NOT NULL,   3    First\_Name         VARCHAR2(10 BYTE),   4    Last\_Name          VARCHAR2(10 BYTE),   5    Start\_Date         DATE,   6    End\_Date           DATE,   7    Salary             Number(8,2),   8    City               VARCHAR2(10 BYTE),   9    Description        VARCHAR2(15 BYTE)  10  )  11  /  Table created.  SQL> SQL> -- prepare data SQL> **insert**into Employee(ID,  First\_Name, Last\_Name, Start\_Date,End\_Date,salary,  City,       Description)   2               values ('01','Jason',    'Martin',  to\_date('19960725','YYYYMMDD'),  to\_date('20060725','YYYYMMDD'), 1234.56, 'Toronto',  'Programmer')   3  /  1 row created.  SQL> **insert**into Employee(ID,  First\_Name, Last\_Name, Start\_Date,  End\_Date,                       Salary,  City,       Description)   2                values('02','Alison',   'Mathews', to\_date('19760321','YYYYMMDD'),  to\_date('19860221','YYYYMMDD'), 6661.78, 'Vancouver','Tester')   3  /  1 row created.  SQL> **insert**into Employee(ID,  First\_Name, Last\_Name, Start\_Date,  End\_Date,                       Salary,  City,       Description)   2                values('03','James',    'Smith',   to\_date('19781212','YYYYMMDD'),  to\_date('19900315','YYYYMMDD'), 6544.78, 'Vancouver','Tester')   3  /  1 row created.  SQL> **insert**into Employee(ID,  First\_Name, Last\_Name, Start\_Date,  End\_Date,                       Salary,  City,       Description)   2                values('04','Celia',    'Rice',    to\_date('19821024','YYYYMMDD'),  to\_date('19990421','YYYYMMDD'), 2344.78, 'Vancouver','Manager')   3  /  1 row created.  SQL> **insert**into Employee(ID,  First\_Name, Last\_Name, Start\_Date,  End\_Date,                       Salary,  City,       Description)   2                values('05','Robert',   'Black',   to\_date('19840115','YYYYMMDD'),  to\_date('19980808','YYYYMMDD'), 2334.78, 'Vancouver','Tester')   3  /  1 row created.  SQL> **insert**into Employee(ID,  First\_Name, Last\_Name, Start\_Date,  End\_Date,                       Salary, City,        Description)   2                values('06','Linda',    'Green',   to\_date('19870730','YYYYMMDD'),  to\_date('19960104','YYYYMMDD'), 4322.78,'New York',  'Tester')   3  /  1 row created.  SQL> **insert**into Employee(ID,  First\_Name, Last\_Name, Start\_Date,  End\_Date,                       Salary, City,        Description)   2                values('07','David',    'Larry',   to\_date('19901231','YYYYMMDD'),  to\_date('19980212','YYYYMMDD'), 7897.78,'New York',  'Manager')   3  /  1 row created.  SQL> **insert**into Employee(ID,  First\_Name, Last\_Name, Start\_Date,  End\_Date,                       Salary, City,        Description)   2                values('08','James',    'Cat',     to\_date('19960917','YYYYMMDD'),  to\_date('20020415','YYYYMMDD'), 1232.78,'Vancouver', 'Tester')   3  /  1 row created.  SQL> SQL> SQL> SQL> -- display data in the table SQL> **select**\* **from**Employee   2  / Hit a key to **continue**  ID   FIRST\_NAME LAST\_NAME  START\_DAT END\_DATE      SALARY CITY       DESCRIPTION ---- ---------- ---------- --------- --------- ---------- ---------- --------------- 01   Jason      Martin     25-JUL-96 25-JUL-06    1234.56 Toronto    Programmer 02   Alison     Mathews    21-MAR-76 21-FEB-86    6661.78 Vancouver  Tester 03   James      Smith      12-DEC-78 15-MAR-90    6544.78 Vancouver  Tester 04   Celia      Rice       24-OCT-82 21-APR-99    2344.78 Vancouver  Manager 05   Robert     Black      15-JAN-84 08-AUG-98    2334.78 Vancouver  Tester 06   Linda      Green      30-JUL-87 04-JAN-96    4322.78 New York   Tester 07   David      Larry      31-DEC-90 12-FEB-98    7897.78 New York   Manager 08   James      Cat        17-SEP-96 15-APR-02    1232.78 Vancouver  Tester  8 rows selected.  SQL> SQL> SQL> SQL> SQL> --**UPDATE**statement can be used within PL/SQL programs to **update**a row or a set of  rows with a **new**-- SQL> --value. SQL> SQL> **BEGIN**   2     **UPDATE**employee   3     SET salary = salary \* 2;   4  **END**;   5  /  PL/SQL **procedure**successfully completed.  SQL> SQL> SQL> **select**\* **from**employee; Hit a key to **continue**  ID   FIRST\_NAME LAST\_NAME  START\_DAT END\_DATE      SALARY CITY       DESCRIPTION ---- ---------- ---------- --------- --------- ---------- ---------- --------------- 01   Jason      Martin     25-JUL-96 25-JUL-06    2469.12 Toronto    Programmer 02   Alison     Mathews    21-MAR-76 21-FEB-86   13323.56 Vancouver  Tester 03   James      Smith      12-DEC-78 15-MAR-90   13089.56 Vancouver  Tester 04   Celia      Rice       24-OCT-82 21-APR-99    4689.56 Vancouver  Manager 05   Robert     Black      15-JAN-84 08-AUG-98    4669.56 Vancouver  Tester 06   Linda      Green      30-JUL-87 04-JAN-96    8645.56 New York   Tester 07   David      Larry      31-DEC-90 12-FEB-98   15795.56 New York   Manager 08   James      Cat        17-SEP-96 15-APR-02    2465.56 Vancouver  Tester  8 rows selected. | |

|  |
| --- |
| 2 The basic four arithmetic operators in action |
| |  | | --- | |  | |
| |  | | --- | | SQL> SQL> --The basic four arithmetic operators in action. SQL> SET SERVEROUTPUT ON SQL> **BEGIN**   2     DBMS\_OUTPUT.PUT\_LINE(4 \* 2);  --Multiplication   3     DBMS\_OUTPUT.PUT\_LINE(24 / 3); --Division   4     DBMS\_OUTPUT.PUT\_LINE(4 + 4);  --Addition   5     DBMS\_OUTPUT.PUT\_LINE(16 - 8); --Subtraction   6  **END**;   7  / 8 8 8 8  PL/SQL **procedure**successfully completed.  SQL> | |

|  |
| --- |
| 3: the concatenation operator |
| |  | | --- | |  | |
| |  | | --- | | SQL> SQL> -- Use of the concatenation operator. SQL> SET SERVEROUTPUT ON SQL> DECLARE   2     a     VARCHAR2(30);   3     b     VARCHAR2(30);   4     c     VARCHAR2(30);   5  **BEGIN**   6     -- Concatenate several string constants.   7     c := 'A' || ' AND ' || 'B';   8     DBMS\_OUTPUT.PUT\_LINE(c);   9     -- Concatenate both string variables and constants.  10     a := 'A';  11     b := 'B';  12     DBMS\_OUTPUT.PUT\_LINE(a || ' ' || b || ',');  13     -- Concatenate two string variables.  14     a := 'A ';  15     b := 'B';  16     c := a || b;  17     DBMS\_OUTPUT.PUT\_LINE(c);  18  **END**;  19  / A AND B A B, A B  PL/SQL **procedure**successfully completed.  SQL> | |

|  |
| --- |
| 4 While Loop with condition |
| |  | | --- | |  | |
| |  | | --- | | SQL> SQL> SET SERVEROUTPUT ON SQL> DECLARE   2      counter INTEGER := 2;   3  **BEGIN**   4   5   6      counter := 0;   7      WHILE counter < 6 LOOP   8        counter := counter + 1;   9        DBMS\_OUTPUT.PUT\_LINE(counter);  10      **END**LOOP;  11  **END**;  12  / 1 2 3 4 5 6  PL/SQL **procedure**successfully completed.  SQL> | |
| 5 Your first FOR loop |
| |  | | --- | |  | |
| |  | | --- | | SQL> SQL> --  Your first FOR loop. SQL> set echo on SQL> **BEGIN**   2       FOR i IN 1..5 LOOP   3            DBMS\_OUTPUT.PUT\_LINE('Loop counter is ' || i);   4       **END**LOOP;   5  **END**;   6  / Loop counter is 1 Loop counter is 2 Loop counter is 3 Loop counter is 4 Loop counter is 5  PL/SQL **procedure**successfully completed. | |
| 6 Call EXIT to exit a for loop |
| |  | | --- | |  | |
| |  | | --- | | SQL> SQL> **BEGIN**   2FOR v\_loopcounter IN 1..20 LOOP   3 IF MOD(v\_loopcounter,2) = 0 THEN   4  DBMS\_OUTPUT.PUT\_LINE('The AREA of the circle is ' ||v\_loopcounter\*v\_loopcounter );   5         **END**IF;   6         IF v\_loopcounter = 10 THEN   7              EXIT;   8         **END**IF;   9   **END**LOOP;  10  **END**;  11  / The AREA of the circle is 4 The AREA of the circle is 16 The AREA of the circle is 36 The AREA of the circle is 64 The AREA of the circle is 100  PL/SQL **procedure**successfully completed. | |
| 7 Print command: print out a variable |
| |  | | --- | |  | |
| |  | | --- | | SQL> SQL> SQL> -- Print command SQL> SQL> SQL> VARIABLE g\_result VARCHAR2(100) SQL> **BEGIN**   2     :g\_result :='Hello';   3  **END**;   4  /  PL/SQL **procedure**successfully completed.  SQL> SQL> PRINT g\_result  G\_RESULT ------------------- Hello | |

|  |
| --- |
| 8 Use in parameter to pass value and insert value to a table |
| |  | | --- | |  | |
| |  | | --- | | SQL> SQL> **create**table t(   2      n number   3  )   4  /  Table created.  SQL> SQL> **create**or replace   2    **procedure**insert\_into\_t( p\_parm1 in number,  p\_parm2 in number ) is   3    **begin**   4      **insert**into t values ( p\_parm1 );   5      **insert**into t values ( p\_parm2 );   6    **end**insert\_into\_t;   7    /  Procedure created.  SQL> SQL> SQL> exec insert\_into\_t( p\_parm1 => 101, p\_parm2 => 102 );  PL/SQL **procedure**successfully completed.  SQL> SQL> **select**\* **from**t;           N ----------        101        102 | |

|  |
| --- |
| 9 Effects of nulls on boolean expressions: = |
| |  | | --- | |  | |
| |  | | --- | | SQL> SQL> --  Effects of nulls on **boolean**expressions. SQL> SET SERVEROUTPUT ON SQL> SQL> DECLARE   2     a     INTEGER;   3     n     INTEGER;     -- be our **null**value.   4  **BEGIN**   5     -- Assign a value to the variable A, but leave N **null**.   6     a := 2;   7     -- Note that the test **for**A=N fails.   8     IF a = n THEN   9       DBMS\_OUTPUT.PUT\_LINE('a = n is **true**');  10     ELSE  11       DBMS\_OUTPUT.PUT\_LINE('a = n is not **true**');  12     **END**IF;  13  **END**;  14  / a = n is not **true**  PL/SQL **procedure**successfully completed.  SQL> | |

|  |
| --- |
| 10 Effects of nulls on boolean expressions: >, < (2) |
| |  | | --- | |  | |
| |  | | --- | | SQL> SQL> --  Effects of nulls on **boolean**expressions. SQL> SET SERVEROUTPUT ON SQL> SQL> DECLARE   2     a     INTEGER;   3     n     INTEGER;     -- be our **null**value.   4  **BEGIN**   5     -- Assign a value to the variable A, but leave N **null**.   6     a := 2;   7     -- But also note that the test **for**a <> n fails.   8     IF a <> n THEN   9       DBMS\_OUTPUT.PUT\_LINE('a <> n is **true**');  10     ELSE  11       DBMS\_OUTPUT.PUT\_LINE('a <> n is not **true**');  12     **END**IF;  13  14  **END**;  15  / a <> n is not **true**  PL/SQL **procedure**successfully completed.  SQL> | |

|  |
| --- |
| 11 Use and to link two boolean expressions |
| |  | | --- | |  | |
| |  | | --- | | SQL> SQL> --  Effects of nulls on **boolean**expressions. SQL> SET SERVEROUTPUT ON SQL> SQL> DECLARE   2     a     INTEGER;   3     n     INTEGER;     -- be our **null**value.   4  **BEGIN**   5     -- Assign a value to the variable A, but leave N **null**.   6     a := 2;   7     -- TRUE and NULL = NULL   8     IF (a = 2) AND (a <> n) THEN   9       DBMS\_OUTPUT.PUT\_LINE('TRUE and NULL = TRUE');  10     ELSE  11       DBMS\_OUTPUT.PUT\_LINE('TRUE and NULL = NULL');  12     **END**IF;  13  **END**;  14  / TRUE and NULL = NULL  PL/SQL **procedure**successfully completed.  SQL> | |
| 12 Constants are compared using blank-padded comparison semantics |
| |  | | --- | |  | |
| |  | | --- | | SQL> SQL> -- Demonstration of string comparison. SQL> SET SERVEROUTPUT ON SQL> DECLARE   2    fixed\_length\_10  CHAR(10);   3    fixed\_length\_20  CHAR(20);   4    var\_length\_10    VARCHAR2(10);   5    var\_length\_20    VARCHAR2(20);   6  **BEGIN**   7    --Constants are compared **using**blank-padded comparison semantics,   8    -- so the trailing spaces won't affect the result.   9    IF 'Abc' = 'Abc          ' THEN  10       DBMS\_OUTPUT.PUT\_LINE('Constant: ''Abc'' = ''Abc          ''');  11    **END**IF;  12  13  **END**;  14  / Constant: 'Abc' = 'Abc          '  PL/SQL **procedure**successfully completed.  SQL> | |

|  |
| --- |
| 13 Comparison of a fixed length string and a literal |
| |  | | --- | |  | |
| |  | | --- | | SQL> SQL> -- Demonstration of string comparison. SQL> SET SERVEROUTPUT ON SQL> DECLARE   2    fixed\_length\_10  CHAR(10);   3    fixed\_length\_20  CHAR(20);   4    var\_length\_10    VARCHAR2(10);   5    var\_length\_20    VARCHAR2(20);   6  **BEGIN**   7   8    --Comparison of a fixed length string and a literal also   9    -- results in the use of blank-padded comparison semantics.  10    IF fixed\_length\_10 = 'Donna' THEN  11       DBMS\_OUTPUT.PUT\_LINE('Char and constant:' || fixed\_length\_10 ||' = ' || 'Donna');  12    ELSE  13       DBMS\_OUTPUT.PUT\_LINE('Not Donna');  14    **END**IF;  15  **END**;  16  / Not Donna  PL/SQL **procedure**successfully completed.  SQL> | | 14 Print out value using DBMS\_OUTPUT.PUT\_LINE | | |  | | --- | |  | | | |  | | --- | | SQL> SQL> --Basic loop SQL> SQL> SQL> SET SERVEROUTPUT ON SQL> DECLARE   2    x NUMBER := 1;   3  **BEGIN**   4    LOOP   5      DBMS\_OUTPUT.PUT\_LINE('This loop has executed'||TO\_CHAR(x)||' time(s)');   6      x := x +1;   7      EXIT WHEN x > 5;   8    **END**LOOP;   9  **END**;  10  11  12  / This loop has executed1 time(s) This loop has executed2 time(s) This loop has executed3 time(s) This loop has executed4 time(s) This loop has executed5 time(s)  PL/SQL **procedure**successfully completed. | | |

|  |
| --- |
| 15.DBMS\_OUTPUT package |
| |  | | --- | |  | |
| |  | | --- | | SQL> SQL> -- DBMS\_OUTPUT **package** SQL> SQL> SQL> SQL> SET SERVEROUTPUT ON SQL> **BEGIN**   2     DBMS\_OUTPUT.PUT\_LINE('Hello');   3     DBMS\_OUTPUT.PUT\_LINE('there');   4  **END**;   5  / Hello there  PL/SQL **procedure**successfully completed.  SQL> SQL> | |