Programando con PL/SQL en una Base de Datos Oracle

Instructor: Ing. Francisco Riccio.

OCA Oracle Database Administrator 10g

OCP Oracle Database Administrator 10g

OCP Oracle Database Administrator 11g

OCA Oracle Application Server 10g

Oracle Database 10g RAC Administrator Certified Expert

Managing Oracle on Linux Certified Expert

Oracle Database SQL Certified Expert

Oracle Database 11g Essentials For Implementors

MCTS SQL Server 2005

Email: francisco@friccio.com

Contenido

Consideraciones en la instalación del Oracle XE y configuración del Oracle SQL Developer

Descargas

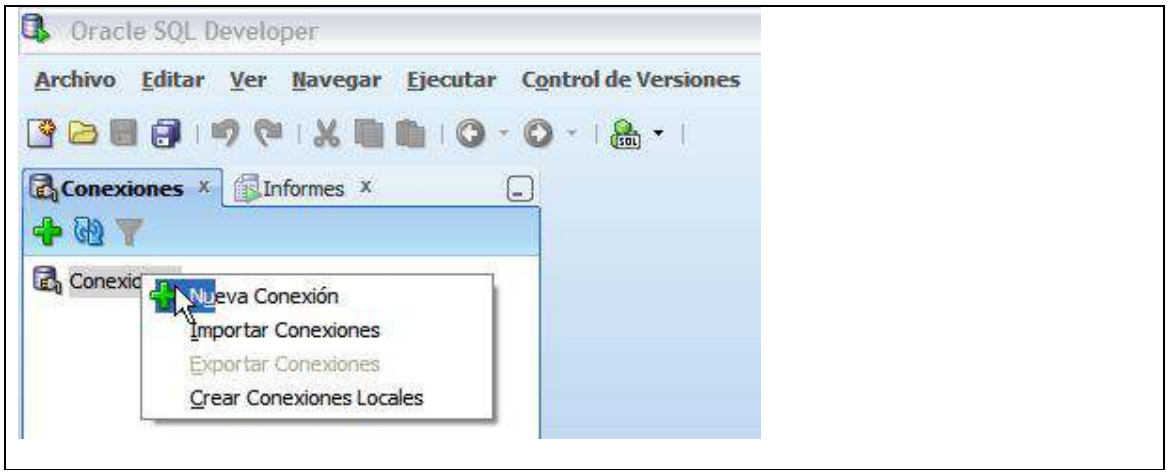
Oracle XE:

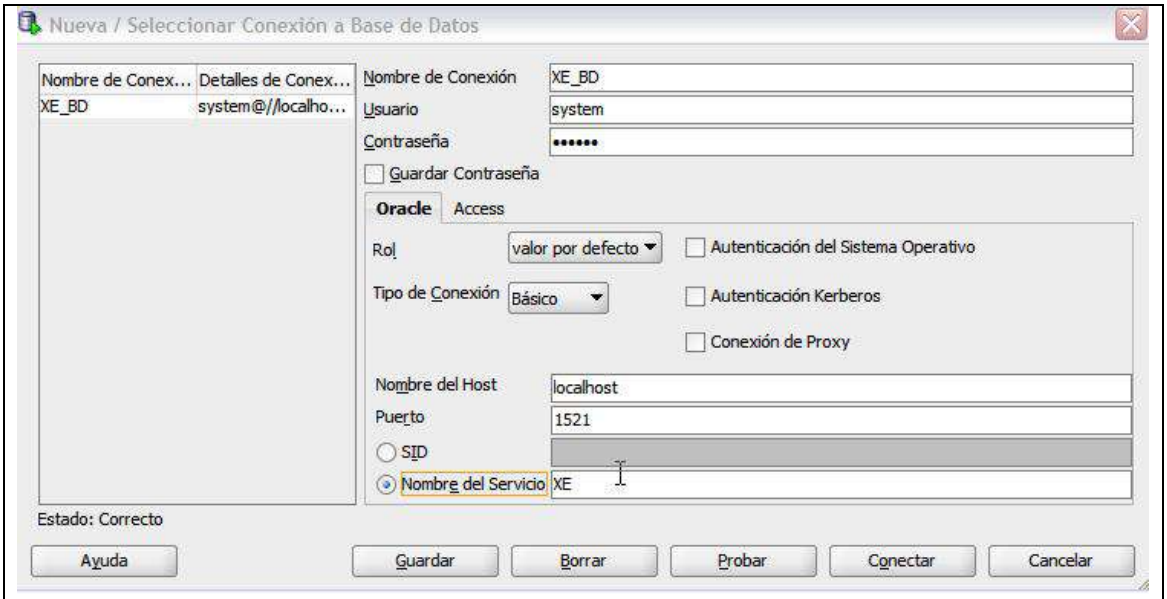
http://www.oracle.com/technetwork/database/express-edition/downloads/index.html

Oracle SQL Developer:

http://www.oracle.com/technetwork/developer-tools/sql-developer/downloads/index.html

Descargados los productos e instalados, se indica los pasos para poder generar una conexión hacia una base de datos desde el Oracle SQL Developer.





Donde en Nombre de Conexión se coloca un nombre de identificación de la conexión que pudiera ser cualquier nombre.

En usuario y contraseña debe ser un usuario válido de la base de datos con su contraseña, en este caso estamos utilizando el usuario SYSTEM que existe en la base de datos.

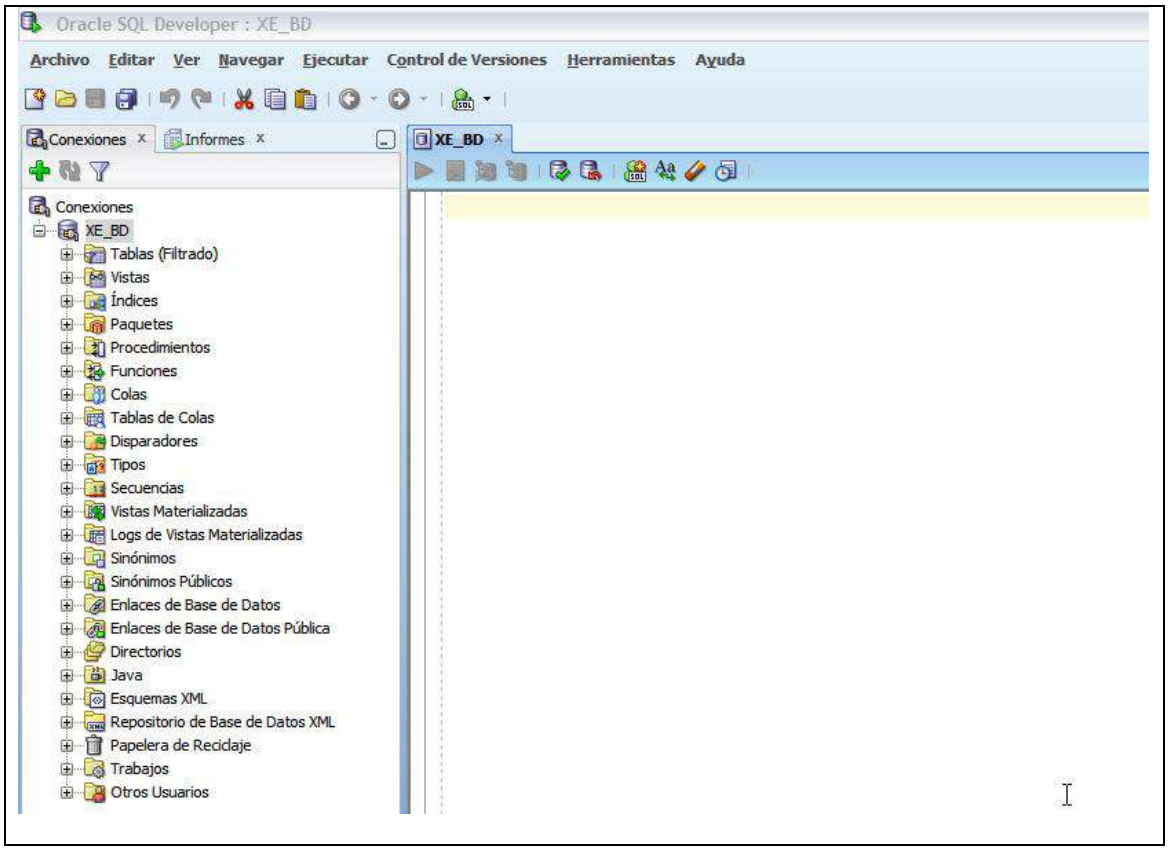
En los campos de Nombre de Host, Puerto y Nombre del Servicio, son datos que hacen referencia al listener de la base de datos. El listener es un componente de la base de datos que será responsable de realizar la conexión a la base de datos.

Estos datos pueden ser extraídos en el servidor de base de datos publicando el comando: lsnrctl status, ejemplo:

```
C:\>lsnrctl status

LSNRCTL for 32-bit Windows: Version 10.2.0.1.0 - Production on 13-NOV-2010 22:20
:29

Copyright (c) 1991, 2005, Oracle.  All rights reserved.

Conectвndose a (DESCRIPTION=(ADDRESS=(PROTOCOL=TCP)(HOST=friccio)(PORT=1521)))
ESTADO del LISTENER
------------------------
Alias                     LISTENER
Versi≤n                   TNSLSNR for 32-bit Windows: Version 10.2.0.1.0 - Produ
ction
Fecha de Inicio           13-NOV-2010 22:17:30
Tiempo Actividad    0 døas 0 hr. 3 min. 0 seg.
Nivel de Rastreo          off
Seguridad                 ON: Password or Local OS Authentication
SNMP                      OFF
Servicio por Defecto      XE
Parвmetros del Listener   C:\app\oracle\product\10.2.0\server\network\admin\list
ener.ora
Log del Listener          C:\app\oracle\product\10.2.0\server\network\log\listen
er.log
Recibiendo Resumen de Puntos Finales...
  (DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)(HOST=friccio)(PORT=1521)))
  (DESCRIPTION=(ADDRESS=(PROTOCOL=tcp)(HOST=127.0.0.1)(PORT=8080))(Presentation=
HTTP)(Session=RAW))
Resumen de Servicios...
El servicio "PLSExtProc" tiene 1 instancia(s).
  La instancia "PLSExtProc", con estado UNKNOWN, tiene 1 manejador(es) para este
 servicio...
El servicio "XE" tiene 2 instancia(s).
  La instancia "XE", con estado UNKNOWN, tiene 1 manejador(es) para este servici
o...
  La instancia "xe", con estado READY, tiene 1 manejador(es) para este servicio.

El servicio "XEXDB" tiene 1 instancia(s).
  La instancia "xe", con estado READY, tiene 1 manejador(es) para este servicio.

El servicio "XE_XPT" tiene 1 instancia(s).
  La instancia "xe", con estado READY, tiene 1 manejador(es) para este servicio.

El comando ha terminado correctamente
```

Donde lo marcado con rojo son los datos del listener que son colocados en el Oracle Developer.

Luego podemos dar click en Probar y debe salir Estado: Correcto y posteriormente guardamos los datos.


Oracle SQL Developer : XE_BD
Archivo Editar Ver Navegar Ejecutar Control de Versiones Herramientas Ayuda
Conexiones
Informes
XE_BD
Conexiones
XE_BD
Tablas (Filtrado)
Vistas
Índices
Paquetes
Procedimientos
Funciones
Colas
Tablas de Colas
Disparadores
Tipos
Secuencias
Vistas Materializadas
Logs de Vistas Materializadas
Sinónimos
Sinónimos Públicos
Enlaces de Base de Datos
Enlaces de Base de Datos Pública
Directorios
Java
Esquemas XML
Repositorio de Base de Datos XML
Papelera de Reciclaje
Trabajos
Otros Usuarios

Introducción a PL/SQL

PLSQL es una extensión de programación a SQL.

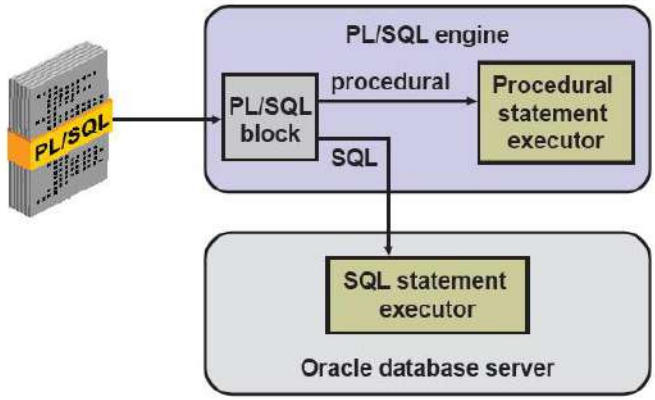
Es el lenguaje de programación de 4ta generación para base de datos Oracle.

Respecto a su arquitectura

Todo código PLSQL se compone de código PLSQL+ sentencias SQL.

Donde el código PLSQL es ejecutado en un engine llamado PLSQL y las secciones que son sentencias SQL son ejecutadas en el SQL Statement Executor (Oracle Database Server).

Toda base de datos Oracle tiene un PLSQL engine de forma inherente. Existe otros productos que cuentan con un PLSQL engine como el Oracle Application Server en sus productos (Oracle Forms, Oracle Reports).



En el caso de aplicaciones con Oracle Forms y Reports, el PLSQL Engine se ejecutará en el lado del cliente y las secciones con sentencias SQL son ejecutadas en la base de datos.

Algunos Beneficios

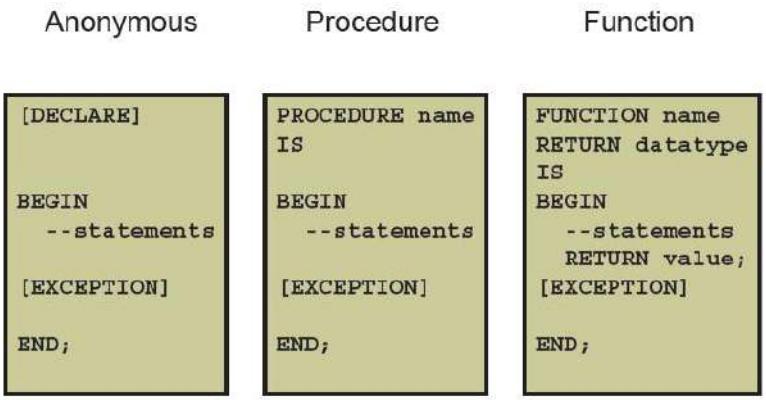
- Permite crear programas modulares.
- Integración con herramientas de Oracle.
- Portabilidad.

- Maneja Excepciones.

Un código en PLSQL puede ser de dos tipos: código anónimo y subprogramas.

Un código anónimo es básicamente aquel que el código fuente reside en el lado cliente y un subprograma reside el código fuente en el servidor. Los subprogramas pueden ser: stored procedures, funciones, triggers y paquetes.

Se muestra las diferentes secciones que componen los diferentes tipos de código PLSQL.



Declaración de variables

La sección de declaración de variables se define en la sección DECLARE ejemplo:

DECLARE

mivar <tipo_dato>

Los tipos de datos en Oracle Database 11g en PLSQL:

CHAR, VARCHAR, NUMBER, BINARY_INTEGER:

Data Type	Description
CHAR [(*maximum_length*)]	Base type for fixed-length character data up to 32,767 bytes. If you do not specify a maximum length, the default length is set to 1.
VARCHAR2 (*maximum_length*)	Base type for variable-length character data up to 32,767 bytes. There is no default size for VARCHAR2 variables and constants.
NUMBER [(*precision*, *scale*)]	Number having precision *p* and scale *s*. The precision *p* can range from 1 through 38. The scale *s* can range from –84 through 127.
BINARY_INTEGER	Base type for integers between –2,147,483,647 and 2,147,483,647

PLSQL_INTEGER, BOOLEAN, BINARY_FLOAT, BINARY_DOUBLE

Data Type	Description
PLS_INTEGER	Base type for signed integers between –2,147,483,647 and 2,147,483,647. PLS_INTEGER values require less storage and are faster than NUMBER values. In Oracle Database 10*g*, the PLS_INTEGER and BINARY_INTEGER data types are identical. The arithmetic operations on PLS_INTEGER and BINARY_INTEGER values are faster than on NUMBER values.
BOOLEAN	Base type that stores one of the three possible values used for logical calculations: TRUE, FALSE, and NULL
BINARY_FLOAT	Represents floating-point number in IEEE 754 format. It requires 5 bytes to store the value.
BINARY_DOUBLE	Represents floating-point number in IEEE 754 format. It requires 9 bytes to store the value.

DATE, TIMESTAMP, TIMESTAMP WITH TIME ZONE.

Data Type	Description
DATE	Base type for dates and times. DATE values include the time of day in seconds since midnight. The range for dates is between 4712 B.C. and A.D. 9999.
TIMESTAMP	The TIMESTAMP data type, which extends the DATE data type, stores the year, month, day, hour, minute, second, and fraction of second. The syntax is TIMESTAMP[(precision)], where the optional parameter precision specifies the number of digits in the fractional part of the seconds field. To specify the precision, you must use an integer in the range 0–9. The default is 6.
TIMESTAMP WITH TIME ZONE	The TIMESTAMP WITH TIME ZONE data type, which extends the TIMESTAMP data type, includes a time-zone displacement. The time-zone displacement is the difference (in hours and minutes) between local time and Coordinated Universal Time (UTC), formerly known as Greenwich Mean Time. The syntax is TIMESTAMP[(precision)] WITH TIME ZONE, where the optional parameter precision specifies the number of digits in the fractional part of the seconds field. To specify the precision, you must use an integer in the range 0–9. The default is 6.

TIMESTAMP WITH LOCAL TIME ZONE, INTERVAL YEAR TO MONTH, INTERVAL DAY TO SECOND

Data Type	Description
TIMESTAMP WITH LOCAL TIME ZONE	The TIMESTAMP WITH LOCAL TIME ZONE data type, which extends the TIMESTAMP data type, includes a time-zone displacement. The time-zone displacement is the difference (in hours and minutes) between local time and Coordinated Universal Time (UTC), formerly known as Greenwich Mean Time. The syntax is TIMESTAMP[(precision)] WITH LOCAL TIME ZONE, where the optional parameter precision specifies the number of digits in the fractional part of the seconds field. You cannot use a symbolic constant or variable to specify the precision; you must use an integer literal in the range 0–9. The default is 6. This data type differs from TIMESTAMP WITH TIME ZONE in that when you insert a value into a database column, the value is normalized to the database time zone, and the time-zone displacement is not stored in the column. When you retrieve the value, the Oracle server returns the value in your local session time zone.
INTERVAL YEAR TO MONTH	You use the INTERVAL YEAR TO MONTH data type to store and manipulate intervals of years and months. The syntax is INTERVAL YEAR[(precision)] TO MONTH, where precision specifies the number of digits in the years field. You cannot use a symbolic constant or variable to specify the precision; you must use an integer literal in the range 0–4. The default is 2.
INTERVAL DAY TO SECOND	You use the INTERVAL DAY TO SECOND data type to store and manipulate intervals of days, hours, minutes, and seconds. The syntax is INTERVAL DAY[(precision1)] TO SECOND[(precision2)], where precision1 and precision2 specify the number of digits in the days field and seconds field, respectively. In both cases, you cannot use a symbolic constant or variable to specify the precision; you must use an integer literal in the range 0–9. The defaults are 2 and 6, respectively.

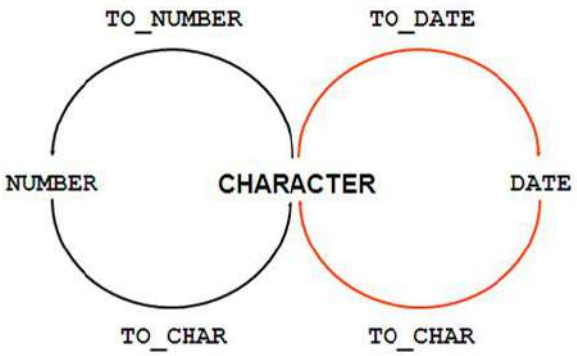
Ejemplo #1:

```
SET SERVEROUTPUT ON
DECLARE
 V_MIVARIABLE VARCHAR(20):='HOLA MUNDO';
BEGIN
 DBMS_OUTPUT.PUT_LINE(V_MIVARIABLE);
 DBMS_OUTPUT.PUT_LINE('FIN DEL PROGRAMA');
END;
/
```

Ejemplo #2

```
SET SERVEROUTPUT ON
DECLARE
 V_NUM1 NUMBER(4,2):=10.2;
 V_NUM2 NUMBER(4,2):=20.1;
BEGIN
 DBMS_OUTPUT.PUT_LINE('LA SUMA ES: '||TO_CHAR(V_NUM1+V_NUM2));
END;
/
```

Casting entre tipos de datos:



Las funciones de conversión básica son: TO_NUMBER, TO_CHAR y TO_DATE.

Ejemplo:

Si deseamos convertir de número a caracter: TO_CHAR(20);

Si deseamos convertir de caracter a número: TO_NUMBER('20');

Si deseamos convertir de fecha a caracter: TO_CHAR(v_fecha, 'DD-MM-YYYY HH24:MI:SS');

Mapeando variables a tipos de datos de columnas de tablas

Para indicar que el tipo de dato de una variable será el que tiene una columna de una tabla se hace mediante el operador **%type**.

Ejemplo #3:

```
SET SERVEROUTPUT ON

DECLARE

 V_FECHA V$DATABASE.CREATED%TYPE;

BEGIN

 SELECT CREATED INTO V_FECHA FROM V$DATABASE;

 DBMS_OUTPUT.PUT_LINE('LA FECHA DE CREACION DE LA BASE DE DATOS FUE: '||TO_CHAR(V_FECHA,'DD-MM-YYYY'));

END;
```

```
/
```

Consideraciones

- La asignación de un valor sobre una variable se realiza mediante el operador := ejemplo: v_mivariable:=20;
- Toda instrucción de PLSQL finaliza con punto y coma (;).
- Si un query devuelve una sola fila y con un campo puede ser asignado a una variable, ejemplo:

  SELECT COUNT(*) INTO MI_VARIABLE FROM HR.EMPLOYEES;

- Si un query devuelve una sola fila y con 2 o más campos también puede ser asignado a varias variables, ejemplo:

  SELECT NAME, CREATED INTO MI_VAR1, MI_VAR2 FROM V$DATABASE;

Manejo de Estructuras de Control (IF/CASE/LOOP/WHILE/FOR)

IF

Sintaxis:

IF (OPERACIÓN LOGICA) THEN

ELSIF THEN

ELSE

END IF;

Ejemplo #4:

```
SET SERVEROUTPUT ON

DECLARE

 V_FECHA V$DATABASE.CREATED%TYPE;

BEGIN

 SELECT CREATED INTO V_FECHA FROM V$DATABASE;

 IF (SYSDATE - V_FECHA > 30) THEN

  DBMS_OUTPUT.PUT_LINE('LA BASE DE DATOS FUE CREADA HACE MÁS DE 30
DIAS.');

 ELSE

  DBMS_OUTPUT.PUT_LINE('LA BASE DE DATOS FUE CREADA HACE MENOS DE
30 DIAS.');

 END IF;

END;

/
```

Ejemplo #5

```
SET SERVEROUTPUT ON
```

```
DECLARE
 V_FECHA V$DATABASE.CREATED%TYPE;
BEGIN
 SELECT CREATED INTO V_FECHA FROM V$DATABASE;
 IF (SYSDATE - V_FECHA > 30) THEN
  DBMS_OUTPUT.PUT_LINE('LA BASE DE DATOS FUE CREADA HACE MÁS DE 30 DIAS.');
 ELSIF (SYSDATE - V_FECHA > 15) AND (SYSDATE - V_FECHA < 30) THEN
  DBMS_OUTPUT.PUT_LINE('LA BASE DE DATOS FUE CREADA HACE MÁS DE 15 DIAS Y MENOS DE 30 DIAS.');
 ELSE
  DBMS_OUTPUT.PUT_LINE('LA BASE DE DATOS FUE CREADA HACE MENOS DE 15 DIAS.');
 END IF;
END;
/
```

CASE

Sintaxis:

```
CASE
 WHEN CONDICION1 THEN
 WHEN CONDICION2 THEN
 ELSE
END CASE;
```

Ejemplo #6:

```
SET SERVEROUTPUT ON

DECLARE

 V_TOTAL NUMBER:=0;

BEGIN

 SELECT COUNT(*) INTO V_TOTAL FROM DBA_OBJECTS;

 CASE

  WHEN V_TOTAL<2000 THEN

   DBMS_OUTPUT.PUT_LINE('LA BASE DE DATOS TIENE MENOS DE 2000
OBJETOS.');

  WHEN (V_TOTAL<4000) AND (V_TOTAL > 2000) THEN

   DBMS_OUTPUT.PUT_LINE('LA BASE DE DATOS TIENE ENTRE 4000 A 2000
OBJETOS.');

  ELSE

   DBMS_OUTPUT.PUT_LINE('LA BASE DE DATOS TIENE MAS DE 4000
OBJETOS.');

 END CASE;

END;

/
```

LOOP

Permite generar bucles.

Sintaxis:

```
LOOP

 EXIT WHEN CONDICION_SALIDA

END LOOP
```

Ejemplo #7:

```
Imprimiendo los primeros 10 números:
SET SERVEROUTPUT ON
DECLARE
 V_NUM NUMBER:=0;
BEGIN
 LOOP
  V_NUM:=V_NUM+1;
  DBMS_OUTPUT.PUT_LINE('NUMERO: '||TO_CHAR(V_NUM));
  EXIT WHEN V_NUM >= 10;
 END LOOP;
END;
/
```

WHILE

Sintaxis:

```
WHILE CONDICION LOOP
END LOOP
```

Ejemplo #8:

```
SET SERVEROUTPUT ON
DECLARE
 V_NUM NUMBER:=1;
BEGIN
 WHILE V_NUM<11 LOOP
  DBMS_OUTPUT.PUT_LINE('NUMERO: '||TO_CHAR(V_NUM));
```

```
  V_NUM:=V_NUM+1;
 END LOOP;
END;
/
```

FOR

Sintaxis:

FOR VARIABLE IN INICIO..FINAL LOOP

END LOOP;

Ejemplo #9:

```
SET SERVEROUTPUT ON
BEGIN
 FOR V_NUM IN 1..10 LOOP
  DBMS_OUTPUT.PUT_LINE('NUMERO: '||TO_CHAR(V_NUM));
 END LOOP;
END;
/
```

Nota: La variable utilizada como contador en el FOR no requiere ser especificada en el DECLARE.

Manejo de Estructuras Complejas

(Registros/Index By/Nested Table/VArray)

Registros

Permite crear estructuras que albergan un conjunto de tipos de datos.

Por ejemplo, podemos crear el registro PERSONA con los campos código, nombre y edad, cada uno de estos campos con diferentes tipos de datos.

Sintaxis:

```
TYPE <NOMBRE_REGISTRO> IS RECORD (
 CAMPO1 TIPO_DATO,
 CAMPO2 TIPO_DATO
 …
);
```

Ejemplo #10:

```
SET SERVEROUTPUT ON
DECLARE
 TYPE TPERSONA IS RECORD (
  CODIGO NUMBER,
  NOMBRE VARCHAR(100),
  EDAD NUMBER
 );
 V_VAR1 TPERSONA;
BEGIN
 V_VAR1.CODIGO:=1;
 V_VAR1.NOMBRE:='FRANCISCO';
```

```
 V_VAR1.EDAD:=30;

 DBMS_OUTPUT.PUT_LINE('CODIGO: '||TO_CHAR(V_VAR1.CODIGO)||'
PERSONA: '||V_VAR1.NOMBRE||' EDAD: '||TO_CHAR(V_VAR1.EDAD)||'.');

END;

/
```

Adicional:

También podemos crear registros que mantengan los mismos campos y tipos de datos que una tabla.

```
DECLARE

  MIVARIABLE DBA_OBJECTS%ROWTYPE;
```

Index By

Index By permite crear arreglos en PLSQL.

Sintaxis:

TYPE <NOMBRE_TIPO_LISTA> IS TABLE OF <TIPO_DATO_NODO> INDEX BY BINARY_INTEGER|PLS_INTEGER|VARCHAR2(#)

Ejemplo #11:

```
SET SERVEROUTPUT ON

DECLARE

 TYPE T_LISTA IS TABLE OF NUMBER INDEX BY BINARY_INTEGER;

 V_LISTA T_LISTA;

BEGIN

 FOR I IN 1..10 LOOP

  V_LISTA(I):=I;

 END LOOP;

 FOR I IN 1..10 LOOP
```

```
  DBMS_OUTPUT.PUT_LINE(I);
 END LOOP;
END;
/
```

Consideraciones con los INDEX BY:

- No pueden ser un tipo de dato de una columna de una tabla.
- No requieren ser inicializados las variables de tipo INDEX BY.
- Los nodos del arreglo no requieren ser inicializados.

Nested Table

Los Nested Table también permite crear arreglos en PLSQL.

Sintaxis:

TYPE <NOMBRE_TIPO_LISTA> IS TABLE OF <TIPO_DATO_NODO>

Ejemplo #12:

```
SET SERVEROUTPUT ON
DECLARE
 TYPE T_LISTA IS TABLE OF NUMBER;
 V_LISTA T_LISTA:=T_LISTA();
BEGIN
 FOR I IN 1..10 LOOP
  V_LISTA.EXTEND;
  V_LISTA(I):=I;
 END LOOP;
 FOR I IN 1..10 LOOP
  DBMS_OUTPUT.PUT_LINE(I);
```

```
 END LOOP;

END;

/
```

Ejemplo #13:

```
CREATE TYPE T_TELEFONOS IS TABLE OF CHAR(7);

/

CREATE TABLE PERSONA (CODIGO NUMBER, NOMBRE VARCHAR(25), LISTA
T_TELEFONOS)

 NESTED TABLE LISTA STORE AS TAB_LISTA;



INSERT INTO PERSONA VALUES
(1,'FRANCISCO',T_TELEFONOS('1234567','7654321'));

COMMIT;



SELECT * FROM PERSONA;
```

Consideraciones con los NESTED TABLE:

- Puede ser un tipo de dato de una columna de una tabla, pero se guarda en otro segmento, es decir no se guarda en la misma tabla físicamente.
- Requiere ser inicializados las variables de tipo NESTED TABLE.
- Por cada nodo nuevo del arreglo debe previamente el arreglo auto extenderse.

VARRAY

Los VARRAY también permiten crear arreglos en PLSQL, pero tienen un tamaño limitado desde su especificación.

Sintaxis:

TYPE <NOMBRE_TIPO_LISTA> IS VARRAY(# NODOS) OF <TIPO_DATO_NODO>

Ejemplo #14:

```
SET SERVEROUTPUT ON

DECLARE

 TYPE T_LISTA IS VARRAY(10) OF NUMBER;

 V_LISTA T_LISTA:=T_LISTA();

BEGIN

 FOR I IN 1..10 LOOP

  V_LISTA.EXTEND;

  V_LISTA(I):=I;

 END LOOP;

 FOR I IN 1..10 LOOP

  DBMS_OUTPUT.PUT_LINE(I);

 END LOOP;

END;

/
```

Ejemplo #15:

```
CREATE TYPE T_TELEFONOS IS VARRAY(2) OF CHAR(7);

/


CREATE TABLE PERSONA (CODIGO NUMBER, NOMBRE VARCHAR(25), LISTA
T_TELEFONOS);


INSERT INTO PERSONA VALUES
(1,'FRANCISCO',T_TELEFONOS('1234567','7654321'));

COMMIT;
```

```
SELECT * FROM PERSONA;
```

Consideraciones con los VARRAY:

- Puede ser un tipo de dato de una columna de una tabla y si es guardado en la misma tabla físicamente.
- Requiere ser inicializados las variables de tipo VARRAY.
- Por cada nodo nuevo del arreglo debe previamente el arreglo auto extenderse.

Los 3 tipos de arreglos tienen métodos que podemos utilizar en nuestros códigos.

Se lista alguno de los métodos:

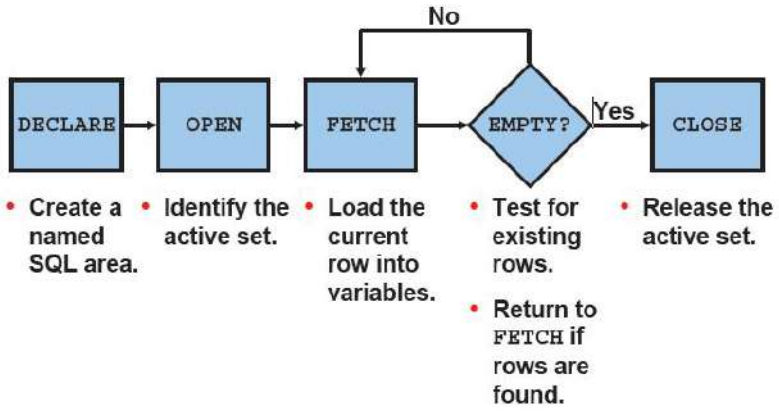
Method	Description
EXISTS(*n*)	Returns TRUE if the *n*th element in a PL/SQL table exists
COUNT	Returns the number of elements that a PL/SQL table currently contains
FIRST	• Returns the first (smallest) index number in a PL/SQL table • Returns NULL if the PL/SQL table is empty
LAST	• Returns the last (largest) index number in a PL/SQL table • Returns NULL if the PL/SQL table is empty
PRIOR(*n*)	Returns the index number that precedes index *n* in a PL/SQL table
NEXT(*n*)	Returns the index number that succeeds index *n* in a PL/SQL table
DELETE	• DELETE removes all elements from a PL/SQL table. • DELETE(n) removes the *n*th element from a PL/SQL table. • DELETE(m, n) removes all elements in the range *m* ... *n* from a PL/SQL table.

Manejo de Cursores

Cada sentencia SQL que es ejecutada en la base de datos siempre tiene asociado un cursor. Los cursores son definidos en el PGA (Memoria privada por cada server process).

Los cursores pueden ser definidios por Oracle de forma explícita o implícita. Implícita significa que Oracle Database es responsable de crear el cursor, abrirlo recorrerlo y cerrarlo. De forma explícita nosotros somos responsables de las actividades mencionadas.

El control de un cursor se define en el siguiente gráfico:



Sintaxis:

```
DECLARE
 CURSOR MICURSOR IS <QUERY>;
BEGIN
 OPEN MICURSOR;
 FETCH MICURSOR INTO VARIABLES_PLSQL
 CLOSE MICURSOR;
END;
/
```

Ejemplo #16:

```
SET SERVEROUTPUT ON

DECLARE

 CURSOR MICURSOR IS SELECT FIRST_NAME, LAST_NAME FROM
HR.EMPLOYEES;

 V_NOMBRE HR.EMPLOYEES.FIRST_NAME%TYPE;

 V_APELLIDO HR.EMPLOYEES.LAST_NAME%TYPE;

BEGIN

 OPEN MICURSOR;

 LOOP

  FETCH MICURSOR INTO V_NOMBRE, V_APELLIDO;

  EXIT WHEN MICURSOR%NOTFOUND;

  DBMS_OUTPUT.PUT_LINE('NOMBRE: '||V_NOMBRE||', APELLIDO:
'||V_APELLIDO);

 END LOOP;

 CLOSE MICURSOR;

END;

/
```

Ejemplo #17:

```
Otra forma de recorrer los cursors:

SET SERVEROUTPUT ON

DECLARE

 CURSOR MICURSOR IS SELECT FIRST_NAME, LAST_NAME FROM
HR.EMPLOYEES;

BEGIN

 FOR C IN MICURSOR LOOP

  DBMS_OUTPUT.PUT_LINE('NOMBRE: '||C.FIRST_NAME||', APELLIDO:
```

```
'||C.LAST_NAME);

 END LOOP;

END;

/
```

En este caso, el FOR automáticamente apertura el cursor y lo cierra.

Los cursores tienen atributos que pueden ayudarnos en el control del mismo.

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

Trabajando con cursores con parámetros:

Permite crear un cursor que permite manejar parámetros, de modo que el mismo cursor podemos utilizarlo en varias ocasiones reemplazando el valor del parámetro.

Sintaxis: CURSOR <NOMBRE_CURSOR>(PARAMETRO TIPO_DATO) IS <QUERY>;

Ejemplo #17:

```
SET SERVEROUTPUT ON

DECLARE

 CURSOR MICURSOR(COD_DEP NUMBER) IS SELECT FIRST_NAME, LAST_NAME
FROM HR.EMPLOYEES WHERE DEPARTMENT_ID = COD_DEP;

BEGIN

 FOR C IN MICURSOR(100) LOOP

  DBMS_OUTPUT.PUT_LINE('NOMBRE: '||C.FIRST_NAME||', APELLIDO:
```

```
'||C.LAST_NAME);

 END LOOP;

END;

/
```

WHERE CURRENT OF

El WHERE CURRENT OF permite actualizar una fila que lo tenemos actualmente apuntando en nuestro cursor. Como condición el cursor debe tener una sentencia SELECT FOR UPDATE.

Ejemplo #18:

```
SET SERVEROUTPUT ON

DECLARE

 CURSOR MICURSOR(COD_DEP NUMBER) IS SELECT FIRST_NAME, LAST_NAME
FROM HR.EMPLOYEES WHERE DEPARTMENT_ID = COD_DEP  FOR UPDATE;

BEGIN

 FOR C IN MICURSOR(100) LOOP

  DBMS_OUTPUT.PUT_LINE('NOMBRE: '||C.FIRST_NAME||', APELLIDO:
'||C.LAST_NAME);

  UPDATE HR.EMPLOYEES

  SET SALARY = 1

  WHERE CURRENT OF MICURSOR;

 END LOOP;

 COMMIT;

END;

/
```

Manejo de cursores Genéricos

Permite crear un cursor donde en tiempo de ejecución se asignará el query que el cursor debe manejar.

Sintaxis:

TYPE <NOMBRE_TIPO> IS REF CURSOR;

VARIABLE <NOMBRE_TIPO>;

Ejemplo #19:

```
SET SERVEROUTPUT ON
DECLARE
 TYPE T_CURSOR IS REF CURSOR;
 MICURSOR T_CURSOR;
 V_APELLIDO HR.EMPLOYEES.LAST_NAME%TYPE;
BEGIN
 OPEN MICURSOR FOR 'SELECT LAST_NAME FROM HR.EMPLOYEES';
 LOOP
  FETCH MICURSOR INTO V_APELLIDO;
  EXIT WHEN MICURSOR%NOTFOUND;
  DBMS_OUTPUT.PUT_LINE('APELLIDO = '||V_APELLIDO);
 END LOOP;
 CLOSE MICURSOR;
END;
/
```

Manipulación de Excepciones

Frente a errores de ejecución que no manejadas automáticamente nuestro código terminará con error. Podemos manejar los errores y hacer un trabajo respectivo frente a un problema esperado o no esperado.

Sintaxis:

```
EXCEPTION
 WHEN ERROR THEN
 <ACCION>
```

Los tipos de errores ya existen algunos pre establecidos por Oracle Database (20 aproximadamente) pero no cubren todos los escenarios. Lo mejor es definir nuestros propios errores.

Sintaxis:

```
DECLARE
 E1 EXCEPTION;
 PRAGMA EXCEPTION_INIT(E1,-#_ERROR);
BEGIN
EXCEPTION
 WHEN E1 THEN
  <ACCION>
END;
```

Nota: El PRAGMA EXCEPTION permite asociar la variable exception con un código de error de Oracle Database.

Ejemplo #20:

```
SET SERVEROUTPUT ON
DECLARE
```

```
 E1 EXCEPTION;
 PRAGMA EXCEPTION_INIT(E1,-1422);
 V_NOMBRE HR.EMPLOYEES.LAST_NAME%TYPE;
BEGIN
 SELECT LAST_NAME INTO V_NOMBRE FROM HR.EMPLOYEES;
EXCEPTION
 WHEN E1 THEN
  DBMS_OUTPUT.PUT_LINE('EL QUERY DEVUELVE MAS DE UNA FILA');
END;
/
```

Si queremos asegurarnos que siempre manejaremos un error a pesar que no lo hayamos identificado en el diseño, podemos usar la constante OTHERS.

Sintaxis:

```
EXCEPTION
 WHEN OTHERS THEN
  <ACCION>
```

Si deseamos recibir el mensaje de error que disparó la excepción se puede utilizar las funciones: SQLERRM y SQLCODE.

RAISE_APPLICATION_ERROR:

Permite disparar excepciones personalizados a las aplicaciones.

Sintaxis:

RAISE_APPLICATION_ERROR(-#_ERROR, 'MENSAJE',FALSE|TRUE);

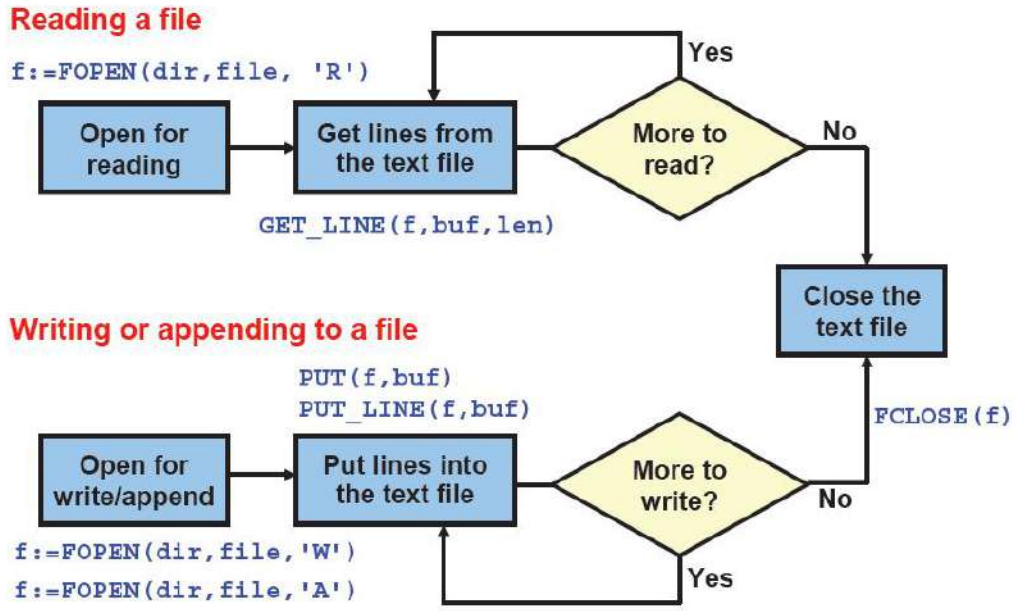
Donde Oracle Database nos reserva los códigos que se encuentran en un rango de -20000 a -20999.

El tercer parámetro que es un tipo de dato boolean, por default es FALSE el cual reemplaza cualquier error que se venga arrastrando y es reemplazado por el texto que se indica. Si en caso se coloca TRUE, se imprimirá la cola de errores arrastrados incluido el mensaje personalizado.

Manejo de Archivos

Con Oracle Database tenemos la posibilidad de crear programas que creen archivos y asimismo leerlos. Esto lo podemos realizar gracias al paquete UTL_FILE que nos proporciona Oracle.

Se muestra un cuadro general del flujo de trabajo con el manejo de archivos.



Como requisito obligatorio debemos crear un objeto directorio el cual apuntará a un directorio del sistema operativo donde se alojará nuestro archivo.

Esto lo realizamos con el comando: CREATE OR REPLACE DIRECTORY

Ejemplo #21:

En este ejercicio se guardara la lista de todos los empleados de la tabla HR.EMPLOYEES.

```
CREATE OR REPLACE DIRECTORY MIDIRECTORIO AS 'C:\';

SET SERVEROUTPUT  ON

DECLARE

 ARCHIVO UTL_FILE.FILE_TYPE;

 CURSOR MICURSOR IS SELECT LAST_NAME FROM HR.EMPLOYEES;
```

```
BEGIN
 ARCHIVO:=UTL_FILE.FOPEN('MIDIRECTORIO','MIARCHIVO.TXT','W');
 FOR C IN MICURSOR LOOP
  UTL_FILE.PUT_LINE(ARCHIVO,C.LAST_NAME);
  UTL_FILE.NEW_LINE(ARCHIVO);
 END LOOP;
 UTL_FILE.FCLOSE(ARCHIVO);
END;
/
```

Ejemplo #22:

```
En este ejercicio leeremos el archivo que hemos creado en el ejercicio anterior.
SET SERVEROUTPUT ON
DECLARE
 ARCHIVO UTL_FILE.FILE_TYPE;
 V_LINEA VARCHAR(250);
BEGIN
 ARCHIVO:=UTL_FILE.FOPEN('MIDIRECTORIO','MIARCHIVO.TXT','R');
 BEGIN
  LOOP
   UTL_FILE.GET_LINE(ARCHIVO,V_LINEA);
   IF (V_LINEA IS NOT NULL) THEN
    DBMS_OUTPUT.PUT_LINE(V_LINEA);
   END IF;
  END LOOP;
 EXCEPTION
```

```
  WHEN NO_DATA_FOUND THEN
   DBMS_OUTPUT.PUT_LINE('FIN DEL ARCHIVO');
 END;
 UTL_FILE.FCLOSE(ARCHIVO);
END;
/
```

Creación de Stored Procedures y Funciones

Un stored procedure es un procedimiento cuyo código se guarda en la base de datos y tiene como objeto realizar una acción específica.

Su sintaxis es la siguiente:

```
CREATE OR REPLACE PROCEDURE <NOMBRE_PROCEDURE> (PARAMETROS) IS
 <DEFINICION_VARIABLES>
BEGIN
 <CODIGO_PLSQL>
END;
```

Ejemplo #23:

```
SET SERVEROUTPUT ON
CREATE OR REPLACE PROCEDURE SPU_SUMAR (X IN NUMBER, Y IN NUMBER)
IS
BEGIN
 DBMS_OUTPUT.PUT_LINE('LA SUMA ES: '||TO_CHAR(X+Y));
END;
/
Probando el stored procedure:
EXECUTE SPU_SUMAR(2,3);
```

Los parámetros que puede recibir un stored procedure pueden ser de 3 tipos:

- IN (Default): Si un parámetro es IN no puede ser modificado en el transcurso del código PLSQL.
- OUT: Si un parámetro es OUT siempre llega al código del stored procedure con el valor de NULL y cuando termina el código de PLSQL el parámetro mantiene el valor de forma persistente.

- IN OUT: Es una combinación de ambos.

Ejemplo #24:

```
CREATE OR REPLACE PROCEDURE SPU_SUMAR (X IN NUMBER, Y IN NUMBER, Z
OUT NUMBER)

IS

BEGIN

 Z:=X+Y;

END;

/

Probando el stored procedure:

SET SERVEROUTPUT ON

DECLARE

 V_SUMA NUMBER:=0;

BEGIN

 SPU_SUMAR(1,3,V_SUMA);

 DBMS_OUTPUT.PUT_LINE(V_SUMA);

END;

/
```

Ejemplo #25:

```
CREATE OR REPLACE PROCEDURE SPU_SUMAR (X IN OUT NUMBER, Y IN OUT
NUMBER, Z OUT NUMBER)

IS

BEGIN

 Z:=X+Y;

END;
```

```
/

Probando el stored procedure:

SET SERVEROUTPUT ON

DECLARE

 V_SUMA NUMBER:=0;

 V_X NUMBER:=10;

 V_Y NUMBER:=20;

BEGIN

 SPU_SUMAR(V_X,V_Y,V_SUMA);

 DBMS_OUTPUT.PUT_LINE(V_SUMA);

END;

/
```

Una función al igual que los stored procedures, su código se guarda en la base de datos y tiene como función realizar un cálculo y devolver un valor.

Sintaxis:

```
CREATE OR REPLACE FUNCTION <NOMBRE_FUNCTION> (PARAMETROS)
RETURN <TIPO_DATO>

IS

 <DEFINICION_VARIABLES>

BEGIN

 <CODIGO_PLSQL>

END;
```

Ejemplo #26:

```
CREATE OR REPLACE FUNCTION GET_TOTAL_OBJ

RETURN NUMBER
```

```
IS
 V_TOTAL NUMBER:=0;
BEGIN
 SELECT COUNT(*) INTO V_TOTAL FROM DBA_OBJECTS;
 RETURN V_TOTAL;
END;
/
```

Probando la función:

```
SELECT GET_TOTAL_OBJ FROM DUAL;
```

Restricciones

- No puede ser utilizado en constraints de tipo CHECK.
- No puede ser utilizado como default de una columna.
- En sentencias SQL que llamen a funciones, estas funciones están restringidas a lo siguiente:
    - En sentencias SELECT, la función no puede ejecutar una sentencia DML.
    - En sentencias DELETE o UPDATE, la función no puede consultar o modificar la tabla que está teniendo el DELETE o el UPDATE.
    - En cualquier sentencia SELECT o DML, la función no pueden ejecutar un COMMIT o ROLLBACK, ni una operación DDL ni DCL ya que genera un COMMIT implícito.

Adicional, podemos crear subprogramas (funciones y stored procedures) que inicien y finalicen una transacción autónoma.

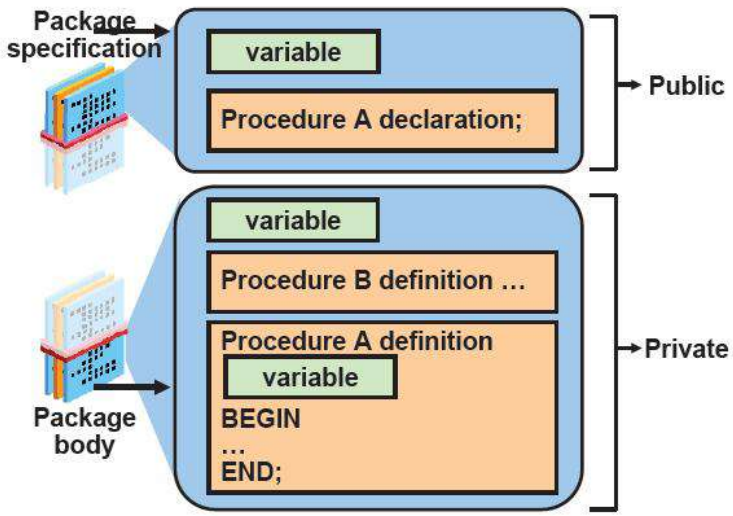
Sintaxis:

```
CREATE OR REPLACE PROCEDURE <NOMBRE_PROCEDURE>
IS
 PRAGMA AUTONOMOUS_TRANSACTION;
```

Creación Paquetes

Un paquete es una agrupación lógica de variables, funciones y stored procedures.

Se divide lógicamente en 2 partes como se detalla en el siguiente gráfico:



Especificación y Body.

En la especificación se define las variables y los subprogramas que queremos publicar a las personas que deseen utilizar nuestros programas.

En el body va la implementación de cada subprograma definido en la especificación y adicional otros subprogramas que quizás no necesitamos que se expongan.

Debemos tener mucho cuidado si definimos variables generales en la especificación o en body ya que son persistentes hasta que la sesión salga de la base de datos.

Una ventaja de utilizar paquetes es que la primera vez que un usuario llama al paquete toda la información del paquete sube en memoria y la información de los valores de las variables que utiliza cada sesión se guarda en el UGA de cada sesión.

Sintaxis:

CREATE OR REPLACE PACKAGE <NOMBRE_PAQUETE> IS <CODIGO> END;

CREATE OR REPLACE PACKAGE BODY <NOMBRE_PAQUETE> IS <CODIGO> END;

Ejemplo #27:

Especificación.

```
CREATE OR REPLACE PACKAGE MIPAQUETE
IS
 V_VAR NUMBER:=0;
 FUNCTION GET_SUMA(X NUMBER, Y NUMBER) RETURN NUMBER;
 PROCEDURE SPU_ACTUALIZA_STOCK;
END;
/
```

Body.

```
CREATE OR REPLACE PACKAGE BODY MIPAQUETE
IS

 V_VAR NUMBER:=0;

 FUNCTION GET_SUMA(X NUMBER, Y NUMBER) RETURN NUMBER
 IS
 BEGIN
  RETURN X+Y;
 END;

 PROCEDURE SPU_ACTUALIZA_STOCK
 IS
 BEGIN
  NULL;
 END;
```

```
END;
/
```

Para trabajos largos y temporales donde probablemente no requerimos tener durante todo el ciclo de vida de la sesión el valor de una variable podemos utilizar el PRAGMA SERIALLY_REUSABLE. Con el PRAGMA, Oracle muda las variables globales al SGA (Large Pool) y los mantiene ahí. Cada llamada de un subprograma que desea utilizar la variable, Oracle realiza una copia de la variable que encuentra en el SGA al UGA del usuario y mantiene el valor en la sesión del usuario hasta que se finalice el subprograma que llamó a la función.

Un PRAGMA es una directiva de compilación.

Ejemplo #28:

```
CREATE OR REPLACE PACKAGE MIPAQUETE IS
  PRAGMA SERIALLY_REUSABLE;
  V_NUM NUMBER := 0;
  PROCEDURE INICIALIZAR(PN NUMBER);
  PROCEDURE IMPRIMIR_VALOR;
END;
/

CREATE OR REPLACE PACKAGE BODY MIPAQUETE IS
  PRAGMA SERIALLY_REUSABLE;

  PROCEDURE INICIALIZAR(PN NUMBER) IS
  BEGIN
   MIPAQUETE.V_NUM:=PN;
   DBMS_OUTPUT.PUT_LINE('NUMERO: '||TO_CHAR(MIPAQUETE.V_NUM));
  END;
```

```
  PROCEDURE IMPRIMIR_VALOR IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE('NUMERO: '||TO_CHAR(MIPAQUETE.V_NUM));
  END;


END;
/
```

Probando:

```
SET SERVEROUTPUT ON
EXECUTE DBMS_OUTPUT.PUT_LINE('VALOR VARIABLE:
'||TO_CHAR(MIPAQUETE.V_NUM));
EXECUTE MIPAQUETE.INICIALIZAR(20);
EXECUTE MIPAQUETE.IMPRIMIR_VALOR;
EXECUTE DBMS_OUTPUT.PUT_LINE('VALOR VARIABLE:
'||TO_CHAR(MIPAQUETE.V_NUM));
```

Creación de Triggers

Los triggers son subprogramas que se disparan frente a eventos que ocurren en la base de datos.

Los tipos de trigger son:

- Simple DML triggers (BEFORE, AFTER y INSTEAD OF).
- Compound triggers.
- Non DML triggers (DDL & Database events).

Los triggers de tipo simple dml será explicado a continuación.

Los triggers de tipo simple dml son aquellos que se disparan cuando ocurre una operación DML y dependiendo del momento que deseemos que se dispare antes o después de la transacción o reemplazar la transacción por otro código.

Sintaxis:

```
CREATE OR REPLACE TRIGGER <NOMBRE_TRIGGER>
 INSTEAD OF|BEFORE|AFTER
 INSERT|DELETE|UPDATE OF <COLUMNAS>
 ON <TABLA>
 FOR EACH ROW
 WHEN <CONDICION>
DECLARE
BEGIN
 <CODIGO>
END;
```

Respecto a FOR EACH ROW, significa que el trigger se disparará por cada fila que esté siendo afectado por la transacción.

Se detalla una comparación entre tener y no tener habilitado el FOR EACH ROW.

Statement-Level Triggers	Row-Level Triggers
Is the default when creating a trigger	Use the FOR EACH ROW clause when creating a trigger.
Fires once for the triggering event	Fires once for each row affected by the triggering event
Fires once even if no rows are affected	Does not fire if the triggering event does not affect any rows

Ejemplo #29:

```
SET SERVEROUTPUT ON


CREATE TABLE TABLA_TG (CAMPO1 NUMBER);


CREATE OR REPLACE TRIGGER TG_TABLA_01
 BEFORE INSERT ON TABLA_TG
DECLARE
BEGIN
 DBMS_OUTPUT.PUT_LINE('CODIGO LIBERADO ANTES DEL INSERT.');
END;
/


CREATE OR REPLACE TRIGGER TG_TABLA_02
 AFTER INSERT ON TABLA_TG
DECLARE
BEGIN
 DBMS_OUTPUT.PUT_LINE('CODIGO LIBERADO DESPUES DEL INSERT.');
END;
```

```
/



INSERT INTO TABLA_TG VALUES (1);
```

Ejemplo #30:

```
SET SERVEROUTPUT ON


DELETE FROM TABLA_TG;


ALTER TRIGGER TG_TABLA_01 DISABLE;

ALTER TRIGGER TG_TABLA_02 DISABLE;


CREATE OR REPLACE TRIGGER TG_TABLA_03

 BEFORE INSERT ON TABLA_TG

 FOR EACH ROW

DECLARE

BEGIN

 DBMS_OUTPUT.PUT_LINE('VALOR INGRESADO ANTES DEL INSERT:
'||TO_CHAR(:NEW.CAMPO1));

 :NEW.CAMPO1:=10;

 DBMS_OUTPUT.PUT_LINE('VALOR MODIFICADO A 10.');

END;

/


CREATE OR REPLACE TRIGGER TG_TABLA_04

 AFTER INSERT ON TABLA_TG
```

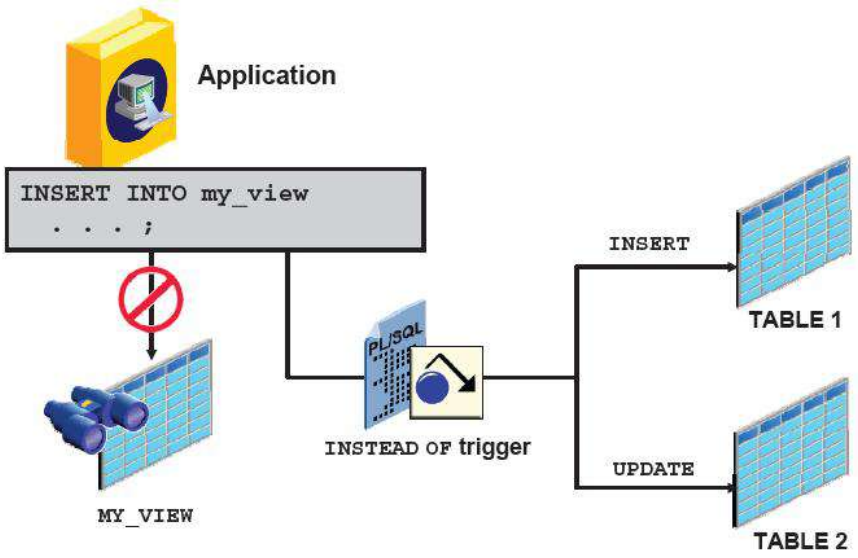
```
 FOR EACH ROW

DECLARE

BEGIN

 DBMS_OUTPUT.PUT_LINE('VALOR INGRESADO DESPUES DEL INSERT:
'||TO_CHAR(:NEW.CAMPO1));

END;

/



INSERT INTO TABLA_TG VALUES (1);

SET SERVEROUTPUT ON


DELETE FROM TABLA_TG;


ALTER TRIGGER TG_TABLA_01 DISABLE;

ALTER TRIGGER TG_TABLA_02 DISABLE;


CREATE TABLE TABLA_TG (CAMPO1 NUMBER);


CREATE OR REPLACE TRIGGER TG_TABLA_03

 BEFORE INSERT ON TABLA_TG

 FOR EACH ROW

DECLARE

BEGIN

 DBMS_OUTPUT.PUT_LINE('VALOR INGRESADO ANTES DEL INSERT:
'||TO_CHAR(:NEW.CAMPO1));

 :NEW.CAMPO1:=10;
```

```
 DBMS_OUTPUT.PUT_LINE('VALOR MODIFICADO A 10.');

END;

/


CREATE OR REPLACE TRIGGER TG_TABLA_04

 AFTER INSERT ON TABLA_TG

 FOR EACH ROW

DECLARE

BEGIN

 DBMS_OUTPUT.PUT_LINE('VALOR INGRESADO DESPUES DEL INSERT:
'||TO_CHAR(:NEW.CAMPO1));

END;

/



INSERT INTO TABLA_TG VALUES (1);
```

Se muestra a continuación un cuadro que muestra cuando tenemos disponibles las variables :NEW y :OLD.

Data Operations	Old Value	New Value
INSERT	NULL	Inserted value
UPDATE	Value before update	Value after update
DELETE	Value before delete	NULL

Los triggers de tipo: INSTEAD OF nos permite reemplazar una transacción por una acción diferente.



Ejemplo #31:

```
SET SERVEROUTPUT ON


DELETE FROM TABLA_TG;


ALTER TRIGGER TG_TABLA_01 DISABLE;
ALTER TRIGGER TG_TABLA_02 DISABLE;
ALTER TRIGGER TG_TABLA_03 DISABLE;
ALTER TRIGGER TG_TABLA_04 DISABLE;


CREATE VIEW VTABLA_TG AS SELECT COUNT(*) AS TOTAL FROM TABLA_TG;


CREATE OR REPLACE TRIGGER TG_TABLA_05
 INSTEAD OF INSERT ON VTABLA_TG
BEGIN
 DELETE FROM TABLA_TG;
```

```
END;
/

INSERT INTO VTABLA_TG VALUES (1);

COMMIT;

SELECT * FROM TABLA_TG;
```

Restricciones

- Los triggers no pueden ser creados en el esquema SYS.
- Los triggers no pueden confirmar o anular una transacción

Consideraciones en el Diseño de Código PL/SQL

Ejecución de Operaciones DDL y DCL

Con el comando EXECUTE IMMEDIATE podemos crear código dinámico que genere una trasacción DDL y DCL.

Sintaxis:

EXECUTE IMMEDIATE('<CODIGO_DDL_DCL');

Ejemplo #32:

```
SET SERVEROUTPUT ON

BEGIN

 EXECUTE IMMEDIATE('CREATE TABLE CODIGO_DDL (CAMPO1 NUMBER)');

END;

/


SELECT * FROM CODIGO_DDL;
```

Con la misma instrucción EXECUTE IMMEDIATE podemos realizar consultas y operaciones DML de forma dinámica.

Sintaxis:

EXECUTE IMMEDAITE ('<QUERY>') INTO VARIABLE USING VALOR_PARAMETRO

Ejemplo #33:

```
SET SERVEROUTPUT ON

DECLARE

 V_TOTAL NUMBER:=0;

BEGIN
```

```
 EXECUTE IMMEDIATE ('SELECT COUNT(*) AS TOTAL FROM HR.EMPLOYEES
WHERE DEPARTMENT_ID=:CODIGO') INTO V_TOTAL USING 100;

 DBMS_OUTPUT.PUT_LINE ('TOTAL DE EMPLEADOS EN EL DEPARTAMENTO 100:
'||TO_CHAR(V_TOTAL));

END;

/
```

NOCOPY

Permite al compilador pasar parámetros OUT y IN OUT como referencia y no por parámetro por valor que es el default; esto reduce el overload en el tiempo de copiado de los parámetros cuando se envían.

Ejemplo #34:

```
CREATE OR REPLACE PROCEDURE SPU_SUMAR (X IN OUT NOCOPY NUMBER, Y
IN OUT NOCOPY NUMBER, Z OUT NOCOPY NUMBER)

IS

BEGIN

 Z:=X+Y;

END;

/
```

No tiene efecto en los siguientes escenarios:

- Parámetros que son elementos de un INDEX BY.
- Sobre parámetros definidos como NOT NULL.
- Sobre parámetros que fueron declarados con %ROWTYPE o %TYPE.
- Si el parámetro requiere una conversión implícita.

BULK COLLECT

El BULK COLLECT nos permite reducir en un 50% el tiempo de acceso a la base de datos. Se basa en minimizar la cantidad de switch context que ocurren al utilizar cursores. Un switch context es un viaje de ida y vuelta entre el PLSQL Engine y el Oracle Database Server.

Ejemplo #35:

En este ejemplo se ha hecho un solo context switch, donde toda la información de la tabla empleados ha sido cargada en un arreglo y ha llegado al PLSQL Engine. Si no hubiéramos usado BULK COLLECT se hubiera hecho un context switch por cada fetch que se realiza en el cursor.

```
SET SERVEROUTPUT ON

DECLARE

 CURSOR MICURSOR IS SELECT * FROM HR.EMPLOYEES;

 TYPE TLISTA IS TABLE OF HR.EMPLOYEES%ROWTYPE;

 LISTA TLISTA;

BEGIN

 OPEN MICURSOR;

 FETCH MICURSOR BULK COLLECT INTO LISTA;

 CLOSE MICURSOR;

 FOR C IN 1..LISTA.COUNT LOOP

   DBMS_OUTPUT.PUT_LINE(LISTA(C).LAST_NAME);

 END LOOP;

END;

/
```

Ejemplo #36:

En este ejemplo se está trayendo en cada context switch 25 filas, el cual es una medida optimiza según algunos análisis realizados. Arriba de 25 no da mayor beneficio de performance.

```
SET SERVEROUTPUT ON
DECLARE
 CURSOR MICURSOR IS SELECT * FROM HR.EMPLOYEES;
 TYPE TLISTA IS TABLE OF HR.EMPLOYEES%ROWTYPE;
 LISTA TLISTA;
BEGIN
 OPEN MICURSOR;
 LOOP
  FETCH MICURSOR BULK COLLECT INTO LISTA LIMIT 25;
  FOR C IN 1..LISTA.COUNT LOOP
   DBMS_OUTPUT.PUT_LINE(LISTA(C).LAST_NAME);
  END LOOP;
  EXIT WHEN MICURSOR%NOTFOUND;
 END LOOP;
 CLOSE MICURSOR;
END;
/
```

FORALL

Consiste igual que el BULK COLLECT, pero ahora está direccionado para operaciones DML que se envían desde el PLSQL Engine hacia la base de datos en una menor cantidad de context switch.

Ejemplo #37:

```
CREATE TABLE EMPLEADO_BK AS SELECT * FROM HR.EMPLOYEES WHERE 1=2;


SET SERVEROUTPUT ON
DECLARE
 CURSOR MICURSOR IS SELECT * FROM HR.EMPLOYEES;
 TYPE TLISTA IS TABLE OF HR.EMPLOYEES%ROWTYPE;
 LISTA TLISTA;
BEGIN
 OPEN MICURSOR;
 LOOP
  FETCH MICURSOR BULK COLLECT INTO LISTA LIMIT 25;
  FORALL I IN 1..LISTA.COUNT
   INSERT INTO EMPLEADO_BK VALUES LISTA(I);
  EXIT WHEN MICURSOR%NOTFOUND;
 END LOOP;
 CLOSE MICURSOR;
END;
/


SELECT * FROM EMPLEADO_BK;
```

Ejemplo #38:

```
SET SERVEROUTPUT ON
DECLARE
 CURSOR MICURSOR IS SELECT * FROM HR.EMPLOYEES;
 TYPE TLISTA IS TABLE OF HR.EMPLOYEES%ROWTYPE;
 TYPE TINDICE IS TABLE OF NUMBER INDEX BY PLS_INTEGER;
 LISTA1 TLISTA;
 LISTA2 TINDICE;
BEGIN
 OPEN MICURSOR;
 LOOP
  FETCH MICURSOR BULK COLLECT INTO LISTA1 LIMIT 25;
  FOR I IN 1..LISTA1.COUNT LOOP
   LISTA2(I):=LISTA1(I).EMPLOYEE_ID;
  END LOOP;
  FORALL I IN 1..LISTA1.COUNT
   UPDATE HR.EMPLOYEES SET SALARY=1000 WHERE EMPLOYEE_ID =
LISTA2(I);
  EXIT WHEN MICURSOR%NOTFOUND;
 END LOOP;
 CLOSE MICURSOR;
END;
/

SELECT LAST_NAME,SALARY FROM HR.EMPLOYEES;
```

DERECHOS DE EJECUCIÓN

Cuando creamos un subprograma el dueño del subprograma en el momento de su creación y ejecución el usuario pierde sus roles y solo quedan presente sus privilegios de objeto y de sistema.

Asimismo cuando creamos un subprograma puede ser ejecutado con los derechos del creador o por el ejecutador.

Por default se ejecuta con los derechos del creador.

Para crear un subprograma que se ejecute con los permisos del ejecutador se debe tener presente la siguiente sintaxis:

CREATE OR REPLACE PROCEDURE <NOMBRE_PROCEDURE> (<PARAMETROS>)
AUTHID CURRENT_USER IS <CODIGO>

Asimismo tener presente al momento de crear un subprograma con la opción de AUTHID CURRENT_USER:

- Operaciones DML y privilegios de objeto sobre tablas son resueltas por los privilegios del invocador del subprograma.
- Otras sentencias como llamadas a funciones, procedures, paquetes son resueltos con los derechos del creador.

Programación Orientada a Objetos en PL/SQL

PLSQL permite trabajar con programación orientada a objetos, de modo que podamos incluso guardar objetos en tablas y poderlo extraer con atributos y métodos.

Sintaxis:

```
CREATE [OR REPLACE] TYPE type_name
  [AUTHID {CURRENT_USER | DEFINER}]
  { {IS | AS} OBJECT | UNDER supertype_name }
(
  attribute_name datatype[, attribute_name datatype]...
  [{MAP | ORDER} MEMBER function_spec,]
  [{FINAL| NOT FINAL} MEMBER function_spec,]
  [{INSTANTIABLE| NOT INSTANTIABLE} MEMBER function_spec,]
  [{MEMBER | STATIC} {subprogram_spec | call_spec}
  [, {MEMBER | STATIC} {subprogram_spec | call_spec}]...]
) [{FINAL| NOT FINAL}] [ {INSTANTIABLE| NOT INSTANTIABLE}];

[CREATE [OR REPLACE] TYPE BODY type_name {IS | AS}
  { {MAP | ORDER} MEMBER function_body;
   | {MEMBER | STATIC} {subprogram_body | call_spec};}
  [{MEMBER | STATIC} {subprogram_body | call_spec};]...
END;]
```

Ejemplo #39:

```
CREATE OR REPLACE TYPE PERSONA AS OBJECT
(
 CODIGO NUMBER,
 NOMBRE VARCHAR(20),
 APELLIDO VARCHAR(25),
MEMBER FUNCTION GET_NOMBRECOMPLETO RETURN VARCHAR
) NOT FINAL;
/


CREATE OR REPLACE TYPE BODY PERSONA IS
```

```
 MEMBER FUNCTION GET_NOMBRECOMPLETO RETURN VARCHAR
 IS
 BEGIN
  RETURN NOMBRE||' '||APELLIDO;
 END;

END;
/
```

Por default las clases son FINAL, lo cual indica que no permite usarse en herencias.

Asimismo podemos trabajar con herencia.

Ejemplo #40:

Estamos creando la clase EMPLEADO heredando atributos y métodos de la clase PERSONA.

```
CREATE OR REPLACE TYPE EMPLEADO UNDER PERSONA
(
 SUELDO NUMBER,
 OVERRIDING MEMBER FUNCTION GET_NOMBRECOMPLETO RETURN VARCHAR,
 MEMBER PROCEDURE SPU_ACTUALIZAR_SUELDO(PSUELDO NUMBER)
) NOT FINAL;
/


CREATE OR REPLACE TYPE BODY EMPLEADO IS


 OVERRIDING MEMBER FUNCTION GET_NOMBRECOMPLETO RETURN VARCHAR
```

```
 IS
 BEGIN
  RETURN LOWER(NOMBRE||' '||APELLIDO);
 END;

 MEMBER PROCEDURE SPU_ACTUALIZAR_SUELDO(PSUELDO NUMBER) IS
 BEGIN
  SUELDO:=PSUELDO;
 END;

END;
/
```

Overriding permite la sobre escritura de un subprograma del padre.

También está permitido polimorfismo.

En los siguientes pasos crearemos una tabla basada en nuestro objeto EMPLEADO y lo manipularemos.

Ejemplo #41:

```
CREATE TABLE TEMPLEADO OF EMPLEADO;

INSERT INTO TEMPLEADO VALUES (EMPLEADO(1,'FRANCISCO','RICCIO',8000));

SELECT * FROM TEMPLEADO;

SELECT E.GET_NOMBRECOMPLETO(),E.* FROM TEMPLEADO E;
```

```
SET SERVEROUTPUT ON

DECLARE

 V_OBJETO EMPLEADO;

BEGIN

 SELECT VALUE(E) INTO V_OBJETO FROM TEMPLEADO E;

 V_OBJETO.SPU_ACTUALIZAR_SUELDO(1000);

 DBMS_OUTPUT.PUT_LINE('EMPLEADO:
'||V_OBJETO.GET_NOMBRECOMPLETO()||' SUELDO =
'||TO_CHAR(V_OBJETO.SUELDO));

END;

/
```

Si hubiéramos creado la variable V_OBJETO como PERSONA, estaríamos trabajando con polimorfismo, de modo que la variable se declara de una clase base y se crea con clases derivadas.