PL/SQL

1.- Introducción al SQL Procedimental

Casi todos los grandes Sistemas Gestores de Datos incorporan ***utilidades*** que permiten ampliar el lenguaje SQL para producir pequeñas utilidades que añaden al SQL mejoras de la ***programación estructurada (bucles, condiciones, funciones,….)***. La razón es que hay diversas acciones en la base de datos para las que SQL no es suficiente.

Por ello todas las bases de datos incorporan algún lenguaje de tipo procedimental (de tercera generación) que permite manipular de forma más avanzada los datos de la base de datos.

PL/SQL es el lenguaje procedimental que es implementado por el pre-compilador de ***Oracle***. Es una extensión procedimental del lenguaje SQL; es decir, se trata de un lenguaje creado para dar a SQL nuevas posibilidades. Esas posibilidades permiten utilizar condiciones y bucles al estilo de los lenguajes de tercera generación (como Basic, Cobol, C++, Java, etc.).

En ***otros sistemas gestores de bases de datos existen otros lenguajes procedimentales***: ***SQL Server utiliza Transact SQL, Informix usa Informix 4GL,…***

Lo interesante del lenguaje PL/SQL es que integra SQL por lo que gran parte de su sintaxis procede de dicho lenguaje.

PL/SQL es un lenguaje pensado para la gestión de datos. La creación de aplicaciones sobre la base de datos se realiza con otras herramientas (Oracle Developer) o lenguajes externos como Visual Basic o Java.

1.1.- Funciones que pueden realizar los programas PL/SQL

Las más destacadas son:

- Facilitar la realización de tares administrativas sobre la base de datos (copia de valores antiguos, auditorías, control de usuarios,…).
- Validación y verificación avanzada de usuarios.
- Consultas muy avanzadas.
- Tareas imposibles de realizar con SQL.

1.2.- Conceptos Básicos

Bloque PL/SQL

Se trata de un trozo de código que puede ser interpretado por Oracle. Se encuentra inmerso dentro de las palabras ***BEGIN*** y ***END***.

Programa PL/SQL

Conjunto de bloques que realizan una determinada labor.

Procedimiento

Programa PL/SQL almacenado en la base de datos y que puede ser ejecutado si se desea con solo saber su nombre (y teniendo permiso para su acceso).

Función

Programa PL/SQL que a partir de unos datos de entrada obtiene un resultado (datos de salida). Una función puede ser utilizada desde cualquier otro programa PL/SQL e incluso desde una instrucción SQL.

Trigger (Disparador)

Programa PL/SQL que ***se ejecuta automáticamente*** cuando ocurre un determinado ***suceso*** a un objeto de la base de datos.

Paquete

Colección de procedimientos y funciones agrupados dentro de la misma estructura. Similar a las bibliotecas y librerías de los lenguajes convencionales.

2.- Escritura de PL/SQL

2.1.- Estructura de un Bloque PL/SQL

Ya se ha comentado antes que los programas PL/SQL se agrupan en estructuras llamadas bloques. Cuando un bloque no tiene nombre, se le llama bloque **anónimo**.

Un bloque puede constar de las siguientes secciones:

- **Declaraciones**: define e inicializa las variables, constantes, excepciones de usuario y cursores utilizados en el bloque. Va precedida de la palabra **DECLARE.**
- **Comandos ejecutables**: sentencias para manipular la base de datos y los datos del programa. Todas estas sentencias van precedidas por la palabra **BEGIN**.

- **Tratamiento de excepciones**: para indicar las acciones a realizar en caso de error. Van precedidas por la palabra **EXCEPTION.**
- **Final del bloque**: la palabra **END** da fin al bloque.

El esquema de la estructura de un **bloque sin nombre (anónimo)** sería:

```
[DECLARE
        declaraciones ]
BEGIN
        instrucciones ejecutables
[EXCEPTION
        instrucciones de manejo de errores ]
END;
```

A los bloques se les puede poner un nombre para crear **procedimientos o funciones.**

- Para ***crear un procedimiento:***

```
PROCEDURE nombre IS
bloque
```

- Para ***crear una función:***

```
FUNCTION nombre
RETURN tipoDedatos IS
bloque
```

2.2.- Escritura de Instrucciones PL/SQL

Normas Básicas

La mayor parte de las normas de escritura en PL/SQL proceden de SQL, por ejemplo:

- Las palabras clave, nombres de tabla y columna, funciones,... no distinguen entre mayúsculas y minúsculas.
- Todas las instrucciones finalizan con el signo del punto y coma (;), excepto las que encabezan un bloque.
- Los bloques comienzan con la palabra BEGIN y terminan con END.
- Las instrucciones pueden ocupar varias líneas.

Comentarios

Para incluir aclaraciones. Pueden ser de dos tipos:

- ***Comentarios de varias líneas***: Comienzan con /* y terminan con */
- ***Comentarios de línea simple***: Son los que utilizan los signos **--** (doble guión). El texto a la derecha de los guiones se considera comentario (el de la izquierda no)

Ejemplo:

```
DECLARE
        valor NUMBER := 17;
BEGIN
        /* Este es un comentario que
        ocupa varias líneas */
        valor := valor*2; -- este sólo ocupa esta línea
        DBMS_OUTPUT.PUT_LINE(valor); -- escribe 34
END;
/
```

3.- Variables y constantes

3.1.- Uso de variables y constantes

Las variables y constantes se declaran en el apartado **DECLARE** del bloque. Toda variable y constante que se desee utilizar en un bloque tiene que haber sido declarada con anterioridad.

La **sintaxis** de la declaración de variables y constantes es:

```
DECLARE
        identificador [CONSTANT] tipoDeDatos [:= valorInicial];
        [siguienteVariable...]
```

Ejemplos:

```
DECLARE
        PI CONSTANT NUMBER(9,7) := 3.1415927;
        radio NUMBER(5);
        area NUMBER(14,2) := 23.12;
```

La palabra **CONSTANT** indica que la variable no puede ser modificada (***es una constante***).

Los identificadores de Oracle deben tener como máximo 30 caracteres, empezar por letra y continuar con letras, números o guiones bajos (_) (también vale el signo de dólar ($) y la almohadilla (#). No debería coincidir con nombres de columnas de las tablas ni con palabras reservadas (como SELECT).

En PL/SQL sólo se puede declarar una variable por línea.

Operador de asignación

El **operador :=** sirve para asignar valores a las variables y constantes.

Este operador si se utiliza en la sección de la declaración, permite inicializar la variable o constante con un valor determinado.

TIPOS DE DATOS PARA LAS VARIABLES

Las variables PL/SQL pueden pertenecer a uno de los siguientes datos (sólo se listan los tipos básicos, los llamados escalares), la mayoría son los mismos del SQL de Oracle.

Tipo de datos	Descripción
CHAR(n)	***Texto*** de anchura fija.
VARCHAR2(n)	***Texto*** de ***anchura variable***.
NUMBER[(p[,s])]	***Número***. Opcionalmente puede indicar el tamaño del número (p) y el número de decimales (s).
DATE	Almacena ***fechas***.
TIMESTAMP	Almacena ***fecha y hora***.
INTERVAL YEAR TO MONTH	Almacena ***intervalos*** de años y meses.
INTERVAL DAY TO SECOND	Almacena ***intervalos*** de días, horas, minutos y segundos.
LONG	Para ***textos*** de ***más de 32767 caracteres***.
LONG RAW	Para ***datos binarios***. PL/SQL no puede mostrar estos datos directamente.
INTEGER	***Enteros*** de -32768 a 32767.
BINARY_INTEGER	***Enteros largos*** (de -2.147.483.647 a -2.147.483.648).
PLS_INTEGER	Igual que el anterior pero ocupa menos espacio.
BOOLEAN	Permite almacenar los valores ***TRUE (verdadero) y FALSE (falso)***.
BINARY_DOUBLE	Disponible desde la versión 10g, formato equivalente al double

	del lenguaje C. Representa números decimales en coma flotante.
BINARY_FLOAT	Otro tipo añadido en la versión 10g, equivalente al float del lenguaje C.

EXPRESIÓN %TYPE

Se utiliza para dar a una variable el mismo tipo de otra variable o el tipo de una columna de una tabla de la base de datos.

La sintaxis es:

```
identificador variable | tabla.columna%TYPE;
```

Ejemplo:

```
nom personas.nombre%TYPE;

precio NUMBER(9,2);

precio_iva precio%TYPE;
```

- La variable precio_iva tomará el tipo de la variable precio (es decir NUMBER(9,2))
- La variable nom tomará el tipo de datos asignado a la columna nombre de la tabla personas.

3.2.- DBMS_OUTPUT.PUT_LINE

Para poder ***mostrar datos (fechas, textos y números)***, Oracle proporciona una función llamada **put_line** en el ***paquete dbms_output***.

Ejemplo:

```
DECLARE
        num NUMBER := 17;
BEGIN
        DBMS_OUTPUT.PUT_LINE (num);
END;
/
```

Eso escribiría el número 17 en la pantalla. Pero para ello se debe ***habilitar primero el paquete*** en el entorno de trabajo que utilicemos. En el caso de iSQL*Plus hay que colocar la orden interna (no lleva punto y coma):

```
SET SERVEROUTPUT ON
```

Hay que escribirla antes de empezar a utilizar la función.

3.3.- Alcance de las variables

Ya se ha comentado que en PL/SQL puede ***haber un bloque dentro de otro bloque***. ***Un bloque puede anidarse*** dentro de:

- Un apartado **BEGIN**
- Un apartado **EXCEPTION**

Importante: Hay que tener en cuenta que las variables declaradas en un bloque concreto, son eliminadas cuando éste acaba (con su END correspondiente).

En el siguiente ejemplo se utilizará un bloque dentro del apartado BEGIN de otro bloque:

```
DECLARE
        num NUMBER := 2;
BEGIN
        num:=num*2;
        DECLARE
                valor NUMBER := 3;
        BEGIN
                valor:=num*3;
                DBMS_OUTPUT.PUT_LINE(valor); --escribe 12
                DBMS_OUTPUT.PUT_LINE(num); --escribe 4
        END;

        DBMS_OUTPUT.PUT_LINE (num*2); --escribe 8
        DBMS_OUTPUT.PUT_LINE (valor); --error (variable sin declarar)
END;
/
```

En el ejemplo anterior, se produce un error porque “valor” no es accesible desde ese punto, el bloque interior ya ha finalizado. Sin embargo desde el bloque interior sí se puede acceder a la variable “num”.

3.4.- Operadores y Funciones

Operadores

- En PL/SQL se permiten utilizar todos los operadores de SQL:
  - Los operadores aritméticos: + - * /
  - Condicionales: > < != <> >= <=
  - Lógicos: **OR AND NOT**
  - Operadores de cadena: ||

- A estos operadores, PL/SQL añade el operador de potencia **.
  - Por ejemplo 4**3 es 4^3.

Funciones

- Se pueden utilizar las **funciones de Oracle** procedentes de SQL (***TO_CHAR, SYSDATE, NVL, SUBSTR, SIN, etc.***,) excepto la función DECODE y las funciones de grupo (SUM, MAX, MIN, COUNT,...), salvo en las instrucciones SQL permitidas.

- A estas funciones se añaden diversas procedentes de paquetes de Oracle o creados por los programadores y las funciones GREATEST y LEAST

3.5.- Paquetes Estándar

Oracle incorpora una serie de paquetes para ser utilizados dentro del código PL/SQL. Es el caso del ***paquete DBMS_OUTPUT*** que sirve para utilizar funciones y procedimientos de escritura como por ejemplo:

- ***PUT_LINE (texto)***: para mostrar un texto en el buffer de datos.
- ***NEW_LINE ()***: para escribir una línea en blanco en el buffer de datos.
  - ***Nota:*** solo aparecerá la línea en blanco si se ha habilitado el paquete de la siguiente forma:

SET SERVEROUTPUT ON FORMAT WRAPPED

Otra forma de forzar un salto de línea sería utilizar la ***función chr(10):***

```
exec dbms_output.put_line ('Hola' || chr(10) || 'Adios');
```

Números Aleatorios

El ***paquete DBMS_RANDOM*** contiene diversas funciones para utilizar número aleatorios.

Quizá la más útil es la **función DBMS_RANDOM.RANDOM** que devuelve un número entero (positivo o negativo) aleatorio (y muy grande).

Si queremos obtener un número aleatorio entre 1 y 10:

```
MOD (ABS (DBMS_RANDOM.RANDOM), 10) +1
```

Ejemplo:

```
DECLARE
        numAleatorio NUMBER;
BEGIN
        numAleatorio := mod(abs(dbms_random.random),10)+1;
        dbms_output.put_line('Numero aleatorio: ' || numAleatorio);
END;
/
```

Entre 20 y 50 sería:

```
MOD (ABS (DBMS_RANDOM.RANDOM), 31) +20
```

Otra función útil seria **DBMS_RANDOM.VALUE(limInicial, limFinal)** que devuelve un número real ***mayor que el límite inicial y menor que el límite final***.

```
numAleatorio := dbms_random.value(1, 5);
        Podría devolver como valores:
                1,8037924629563241
                4,9637925621343432
```

Se podría eliminar la "parte real" mediante la función ***trunc:***

```
numAleatorio := trunc(dbms_random.value(1, 5), 0);
        Obtendríamos números comprendidos entre 1 y 4
```

Ejemplo:

```
DECLARE
        numAleatorio NUMBER;
BEGIN
        numAleatorio := trunc(dbms_random.value(1,5),0);
        dbms_output.put_line('Numero aleatorio: ' || numAleatorio);
END;
/
```

Si se quiere inicializar la ***semilla de generación de números aleatorios***, habría que utilizar la función ***seed***.

```
exec dbms_random.seed (7);
```

4.- Instrucciones SQL permitidas

4.1.- Instrucciones SELECT en PL/SQL

PL/SQL admite el uso de un SELECT que permite almacenar valores en variables. Es el llamado ***SELECT INTO***.

Su **sintaxis** es:

```
SELECT listaDeCampos
INTO listaDeVariables
FROM tabla
[JOIN ...]
[WHERE condición]
```

Importante:

- La cláusula **INTO** es obligatoria en PL/SQL.

- La expresión **SELECT** sólo puede devolver una única fila; de otro modo, ocurre un error.

Ejemplo:

```
DECLARE
        v_salario NUMBER(9,2);
        v_nombre VARCHAR2(50);
BEGIN
        SELECT salario, nombre INTO v_salario, v_nombre
                FROM empleados WHERE id_empleado=12344;
        DBMS_OUTPUT.PUT_LINE('El nuevo salario de ' || v_nombre ||
                'si se incrementa en un 20% será de ' || v_salario*1.2 || 'euros');
END;
/
```

4.2.- Instrucciones DML y de Transacción

Se pueden utilizar instrucciones DML dentro del código ejecutable. Se permiten las instrucciones **INSERT**, **UPDATE**, **DELETE** y **MERGE**; con la ventaja de que en PL/SQL se pueden utilizar variables.

Las instrucciones de transacción **ROLLBACK** y **COMMIT** también están permitidas para anular o confirmar instrucciones.

Instrucción merge:

Instrucción nueva desde la versión Oracle 9i. Esta instrucción dependiendo de una condición lógica, actualiza registros (*es decir realiza un "UPDATE"*) en caso de que la condición se cumple o da de alta registros nuevos (*es decir realiza un "INSERT"*) cuando dicha condición no se cumpla.

Ejemplo:

```
MERGE INTO clientes USING cli_cambios
ON (clientes.codCli = cli_cambios.codCli)
WHEN MATCHED THEN
        UPDATE SET
                clientes.nombre = cli_cambios.nombre,
                clientes.priApe = cli_cambios.priApe
WHEN NOT MATCHED THEN
        INSERT (clientes.codCli, clientes.nombre, clientes.priApe)
        VALUES (cli_cambios.codCli, cli_cambios.nombre, cli_cambios.priApe);
```

5.- Instrucciones de Control de Flujo

Son las instrucciones que permiten ***ejecutar un bloque de instrucciones u otro dependiendo de una condición***. También permiten ***repetir un bloque de instrucciones*** hasta cumplirse la condición (es lo que se conoce como **bucles**).

La mayoría de estructuras de control PL/SQL son las mismas que las de los lenguajes tradicionales como C o Pascal. En concreto PL/SQL se basa en el lenguaje Ada.

5.1.- Instrucción IF

Se trata de una sentencia heredada de los lenguajes estructurados. Desde esta sentencia se consigue que ciertas instrucciones se ejecuten o no dependiendo de una condición.

SENTENCIA IF SIMPLE

Sintaxis:

```
IF condicion THEN
        instrucciones
END IF;
```

Las ***instrucciones se ejecutan en el caso de que la condición sea verdadera***. La condición es cualquier expresión que devuelva verdadero o falso.

Ejemplo:

```
IF (departamento=134) THEN
        salario := salario * 13;
        departamento := 123;
END IF;
```

SENTENCIA IF-THEN-ELSE

Sintaxis:

```
IF condición THEN
        instrucciones
ELSE
        instrucciones
END IF;
```

En este caso las instrucciones bajo el ***ELSE*** se ejecutan ***si la condición es falsa***.

SENTENCIA IF-THEN-ELSIF

Cuando se utilizan sentencias de control es común desear ***anidar un IF dentro de otro***.

Ejemplo:

```
IF (saldo>90) THEN
        DBMS_OUTPUT.PUT_LINE('Saldo mayor que el esperado');
ELSE
        IF (saldo>0) THEN
                DBMS_OUTPUT.PUT_LINE('Saldo menor que el esperado');
        ELSE
                DBMS_OUTPUT.PUT_LINE('Saldo NEGATIVO');
        END IF;
END IF;
```

Otra solución es utilizar esta estructura:

```
IF condición1 THEN
        instrucciones1
ELSIF condición2 THEN
        instrucciones3
[ELSIF.... ]
[ELSE
        instruccionesElse ]
END IF;
```

En este IF (que es el más completo) es evalúa la primera condición; si es verdadera se ejecutan las primeras instrucciones y se abandona el IF; si no es así se mira la siguiente condición y si es verdadera se ejecutan las siguientes instrucciones, si es falsa se va al siguiente ELSIF a evaluar la siguiente condición, y así sucesivamente. La cláusula ELSE se ejecuta sólo si no se cumple ninguna de las anteriores condiciones.

Ejemplo (equivalente al anterior):

```
IF (saldo>90) THEN
        DBMS_OUTPUT.PUT_LINE ('Saldo mayor que el esperado');
ELSIF (saldo>0) THEN
        DBMS_OUTPUT.PUT_LINE ('Saldo menor que el esperado');
ELSE
```

```
        DBMS_OUTPUT.PUT_LINE (‘Saldo NEGATIVO’);
END IF;
```

5.2.- Sentencia CASE

La sentencia ***CASE devuelve un resultado tras evaluar una expresión***.

Sintaxis:

```
CASE selector
        WHEN expresion1 THEN resultado1
        WHEN expresion2 THEN resultado2
        ...
        [ELSE resultadoElse]
END;
```

Importante:

- La sentencia CASE sólo lleva punto y coma **(;)** al final del END!!

- Ten en cuenta que en PL/SQL la sentencia CASE sirve para devolver un valor, NO para ejecutar una instrucción.

Ejemplo:

```
texto:= CASE actitud
        WHEN 'A' THEN 'Muy buena'
        WHEN 'B' THEN 'Buena'
        WHEN 'C' THEN 'Normal'
        WHEN 'D' THEN 'Mala'
        ELSE 'Desconocida'
 END;
```

También se pueden escribir sentencias CASE más complicadas.

Ejemplo:

```
DECLARE
        nota_final number(4,2);
        nota number(4,2) := 5;
        actitud varchar2(1) : = ‘E’;
BEGIN
```

```
        nota_final:= CASE
            WHEN (actitud='A') AND (nota>=4) THEN (nota + 2)
            WHEN (actitud='B') AND (nota>=4) THEN (nota + 1)
            WHEN ((actitud='C') OR (actitud='D')) AND (nota>=5) THEN (nota - 2)
            WHEN (actitud='E') AND (nota>=5) THEN (nota - 2)
            ELSE (nota)
        END;
END;
/
```

5.3.- Bucles

BUCLE LOOP

Se trata de una instrucción que contiene instrucciones que se ***repiten indefinidamente (bucle infinito***). Se inicia con la palabra **LOOP** y finaliza con la palabra **END LOOP** y dentro de esas palabras se colocan las instrucciones que se repetirán.

Lógicamente no tiene sentido utilizar un bucle infinito, por eso existe una ***instrucción*** llamada **EXIT** que permite ***abandonar el bucle***. Cuando Oracle encuentra esa instrucción, el programa continúa desde la siguiente instrucción al END LOOP.

Lo normal es colocar ***EXIT dentro de una sentencia IF*** a fin de establecer una condición de salida del bucle, o acompañar a la palabra EXIT de la ***palabra WHEN seguida de una condición***. Si se condición es cierta, se abandona el bucle, sino continuamos dentro.

Sintaxis:

```
LOOP
        instrucciones
        ...
        EXIT [WHEN condición]
END LOOP;
```

Ejemplo (bucle que escribe los números del 1 al 10):

```
DECLARE
        cont NUMBER :=1;
BEGIN
        LOOP
                DBMS_OUTPUT.PUT_LINE(cont);
                EXIT WHEN cont=10;
                cont:=cont+1;
```

```
        END LOOP;
END;
/
```

BUCLE WHILE

Genera un bucle cuyas ***instrucciones se repiten mientras la condición que sigue a la palabra WHILE sea verdadera***.

Sintaxis:

```
WHILE condición LOOP
        instrucciones
END LOOP;
```

En este bucle es posible utilizar (aunque no es muy habitual en este tipo de bucle) la instrucción EXIT o EXIT WHEN. La diferencia con el anterior es que este es más estructurado (más familiar para los programadores de lenguajes como Pascal, C, Java,...)

Ejemplo (escribir números del 1 al 10):

```
DECLARE
        cont NUMBER :=1;
BEGIN
        WHILE (cont<=10) LOOP
                DBMS_OUTPUT.PUT_LINE(cont);
                cont:=cont+1;
        END LOOP;
END;
/
```

BUCLE FOR

Se utilizar para bucles con contador, ***bucles que se recorren un número concreto de veces***. Para ello se utiliza una ***variable (contador) que no tiene que estar declarada en el DECLARE***, esta variable es declarada automáticamente en el propio FOR y se elimina cuando éste finaliza.

Se indica el valor inicial de la variable y el valor final (el incremento irá de uno en uno). Si se utiliza la **cláusula REVERSE**, entonces el contador cuenta desde el valor alto al bajo restando 1.

Sintaxis:

```
FOR contador IN [REVERSE] valorBajo..valorAlto LOOP
        instrucciones
END LOOP;
```

Ejemplo (escribir números del 1 al 10):

```
BEGIN
        FOR contador IN 1..10 LOOP
                DBMS_OUTPUT.PUT_LINE(contador);
        END LOOP;
END;
/
```

Ejemplo (escribir números del 10 al 1):

```
BEGIN
        FOR contador IN REVERSE 1..10 LOOP
                DBMS_OUTPUT.PUT_LINE(contador);
        END LOOP;
END;
/
```

Importante:

Aunque utilices **REVERSE** en un bucle **FOR** debes indicar los límites del bucle desde el valor más bajo al más alto!!

BUCLES ANIDADOS

Se puede colocar un bucle dentro de otro sin ningún problema, puede haber un WHILE dentro de un FOR, un LOOP dentro de otro LOOP, etc.

Hay que tener en cuenta que en ese caso, la sentencia ***EXIT abandonaría el bucle en el que estamos***:

```
BEGIN
   FOR i IN 1..10 LOOP
        FOR j IN 1..30 LOOP
                EXIT WHEN j=5;
                DBMS_OUTPUT.PUT_LINE(' I: ' || i || ' J: ' || j );
```

```
        END LOOP;
    END LOOP;
END;
/
```

El bucle más interior sólo cuenta hasta que j vale 5 ya que la instrucción EXIT abandona el bucle más interior cuando j llega a ese valor.

No obstante hay una variante de la instrucción EXIT que permite salir incluso del bucle más exterior. Eso se consigue poniendo una ***etiqueta a los bucles que se deseen***. Una etiqueta es un identificador que se coloca dentro de los signos **<< *y* >>** ***delante del bucle***. Eso permite poner nombre al bucle.

Ejemplo:

```
BEGIN
    <<buclei>>
    FOR i IN 1..10 LOOP
        FOR j IN 1..30 LOOP
            EXIT buclei WHEN j=5;
            DBMS_OUTPUT.PUT_LINE(' I: ' || i || ' J: ' || j );
        END LOOP;
    END LOOP buclei;
END;
/
```

En este caso cuando j vale 5 se abandonan ambos bucles. No es obligatorio poner la etiqueta en la instrucción END LOOP (en el ejemplo en la instrucción END LOOP buclei), pero se suele hacer por dar mayor claridad al código.

6.- Cursores

6.1.- Introducción

Los cursores representan ***consultas SELECT de SQL que devuelven más de un resultado y que permiten el acceso a cada fila de dicha consulta***. Lo cual significa que el cursor siempre tiene un puntero señalando a una de las filas del SELECT que representa el cursor.

Se puede recorrer el cursor haciendo que el puntero se mueva por las filas. Los cursores son herramientas fundamentales en PL/SQL.

6.2.- Procesamiento de Cursores

1. ***Declarar el cursor.***
2. ***Abrir el cursor.*** Tras abrir el cursor, el puntero del cursor señalará a la primera fila (si la hay).
3. ***Procesar el cursor.*** La instrucción **FETCH** permite recorrer el cursor registro a registro hasta que el puntero llegue al final (se dice que hasta que el cursor esté vacío).
4. ***Cerrar el cursor.***

6.3.- Declaración de Cursores

Sintaxis:

```
CURSOR nombre IS sentencia_SELECT;
```

La sentencia SELECT indicada no puede tener apartado INTO. Lógicamente ***la declaración del curso debe aparecer en el apartado DECLARE.***

Ejemplo:

```
CURSOR cursorProvincias IS
        SELECT p.nombre as nombre, SUM(poblacion) AS poblacion
        FROM localidades l JOIN provincias p USING (n_provincia)
        GROUP BY p.nombre;
```

6.4.- Apertura de Cursores

```
OPEN cursor;
```

Esta sentencia ***abre el cursor***, lo que significa:

1. Reservar memoria suficiente para el cursor.
2. ***Ejecutar la sentencia SELECT*** a la que se refiere **el cursor**.
3. Colocar ***el puntero*** de recorrido de registros ***en la primera fila***.

Si la sentencia SELECT del cursor no devuelve registros, Oracle no devolverá una excepción. Hasta intentar leer no sabremos si hay resultados o no.

6.5.- Instrucción FETCH

La sentencia ***FETCH es la encargada de recorrer el cursor e ir procesando los valores del mismo***:

```
FETCH cursor INTO listaDeVariables;
```

Esta instrucción ***almacena el contenido de la fila a la que apunta actualmente el puntero en la lista de variables indicada***.

La lista de variables tiene tener el mismo tipo y número que las columnas representadas en el cursor (por supuesto el orden de las variables se tiene que corresponder con la lista de columnas).

Tras esta instrucción el puntero de registros avanza a la siguiente fila (si la hay).

Ejemplo:

```
FETCH cursorProvincias INTO v_nombre, v_poblacion;
```

Una instrucción FETCH lee una sola fila y su contenido lo almacena en variables. Por ello ***se usa siempre dentro de bucles a fin de poder leer todas las filas de un cursor***:

```
LOOP
        FETCH cursorProvincias INTO (v_nombre, v_poblacion);
        EXIT WHEN...    --aquí se pondría la condición de salida
        ...             --instrucciones de proceso de los datos del cursor
END LOOP;
```

6.6.- Cerrar el Cursor

```
CLOSE cursor;
```

Al cerrar el cursor se libera la memoria que ocupa y se impide su procesamiento (no se podría seguir leyendo filas). Tras cerrar el cursor se podría abrir de nuevo.

6.7.- Atributos de los Cursores

Para poder procesar adecuadamente los cursores se pueden utilizar una serie de ***atributos que devuelven verdadero o falso según la situación actual del cursor***. Estos atributos facilitan la manipulación del cursor.

Se utilizan indicando el nombre del cursor, el símbolo % e inmediatamente el nombre del atributo a valorar (por ejemplo ***cursorProvincias%ISOPEN***)

%ISOPEN

Devuelve *verdadero si el cursor ya está abierto*.

%NOTFOUND

Devuelve *verdadero si la última instrucción FETCH no devolvió ningún valor*.

Ejemplo:

```
DECLARE
        CURSOR cursorProvincias IS
                SELECT p.nombre as nombre, SUM(poblacion) AS poblacion
                FROM LOCALIDADES l
                JOIN PROVINCIAS p USING (n_provincia)
                GROUP BY p.nombre;
        v_nombre PROVINCIAS.nombre%TYPE;
        v_poblacion LOCALIDADES.poblacion%TYPE;
BEGIN
        OPEN cursorProvincias;
        LOOP
                FETCH cursorProvincias INTO v_nombre,  v_poblacion;

                EXIT WHEN cursorProvincias%NOTFOUND;

                DBMS_OUTPUT.PUT_LINE(v_nombre || ',' ||  v_poblacion);
        END LOOP;
        CLOSE cursorProvincias;
END;
```

En el ejemplo anterior ***se recorre el cursor hasta que el FETCH no devuelve ninguna fila.*** Lo que significa que el programa anterior muestra el nombre de cada provincia seguida de una coma y de la población de la misma.

%FOUND

Instrucción contraria a la anterior, devuelve *verdadero si el último FETCH devolvió una fila*.

%ROWCOUNT

Indica el *número de filas que se han recorrido en el cursor (inicialmente vale cero)*. Es decir, indica cuántos FETCH se han aplicado sobre el cursor.

6.8.- Cursores implícitos: parámetros

Los cursores anteriores realmente se llaman ***cursores explícitos***, ya que los define y los maneja el programador.

¿Qué es un cursor "implícito"? Es un cursor que se asocia de forma automática (*implícita)* a toda sentencia DML (*Data Manipulation Language)* y PL/SQL SELECT (incluyendo consultas que solo devuelven una fila).

Ejemplo:

```
BEGIN
        UPDATE personas SET nombre = 'Don ' || nombre;
END;
/
```

El bloque anterior ha modificado todos los datos de la tabla personas mediante una sentencia DML (*update)***.** Esta sentencia ha provocado que se ***declare un cursor implícito*** que ha recorrido las filas que se han actualizado (en este caso todas) y las actualice una a uno.

¿Tiene nombre este cursor implícito?* Sí**. PL/SQL nos deja referirnos a él como "**SQL**", pero sólo tenemos accesible el último cursor ya que Oracle utiliza el mismo nombre para todos los cursores implícitos. La única operación que podemos se pueden realizar con los cursores implícitos es utilizar los ***atributos del cursor para obtener información de la última operación realizada.

ATRIBUTOS DE LOS CURSORES IMPLÍCITOS

Existen cuatro atributos para los cursores implícitos que permiten evaluar qué pasó cuando se utilizó por última vez el cursor implícito "SQL".

Estos atributos se utilizan como funciones y no se pueden utilizar en sentencias SQL, aunque sí se pueden evaluar en expresiones de comparación, etc ...

SQL%ROWCOUNT

Número de filas afectadas

SQL%FOUND

Atributo booleano que da como resultado TRUE si la sentencia SQL más reciente ha afectado a una o más filas.

SQL%NOTFOUND

Atributo booleano que da como resultado TRUE si la sentencia SQL más reciente no ha afectado a ninguna fila.

SQL%ISOPEN

Siempre devuelve como resultado FALSE porque PL/SQL cierra los cursores implícitos una vez ejecutados

6.9.- Variables de Registro

Los **registros** son un ***tipo de datos que se compone de datos más simple***. Por ejemplo el registro persona se compondría de los datos simples nombre, apellidos, dirección, fecha de nacimiento, etc.

En PL/SQL su interés radica en que ***cada fila de una tabla o vista se puede interpretar como un registro***, ya que cada fila se compone de datos simples.

Gracias a esta interpretación, ***los registros facilitan la manipulación de los cursores*** ya que podemos entender que un cursor es un conjunto de registros (cada registro sería una fila del cursor).

DECLARACIÓN DEL TIPO DE DATO REGISTRO

Para utilizar registros, primero hay que definir los datos que componen al registro. Así se define ***el tipo de registro*** (por eso se utiliza la palabra **TYPE**).

Después se declarará una ***variable de registro que sea del tipo declarado*** (es decir, puede haber varias variables del mismo tipo de registro).

Sintaxis:

```
TYPE nombreTipoRegistro IS RECORD (
        campo1 tipoCampo1 [:= valorInicial],
        campo2 tipoCampo2 [:= valorInicial],
        ...
        campoN tipoCampoN [:= valorInicial]
);

nombreVariableDeRegistro nombreTipoRegistro;
```

Ejemplo:

```
DECLARE
        TYPE regPersona IS RECORD(
                nombre VARCHAR2(25),
                apellido1 VARCHAR2(25),
                apellido2 VARCHAR2(25),
                fecha_nac DATE
        );

        persona1 regPersona;
        persona2 regPersona;
```

USO DE REGISTROS

Para rellenar los valores de los registros se indica el nombre de la variable de registro seguida de un punto y el nombre del campo a rellenar:

```
persona1.nombre := 'Alvaro';
persona1.fecha_nac := TO_DATE('2/3/2004');
```

%ROWTYPE

Al declarar registros, se puede utilizar el **modificador %ROWTYPE** que sirve para asignar a un registro la estructura de una tabla.

Ejemplo:

```
DECLARE
        regPersona personas%ROWTYPE;
```

Donde:

- ***"personas"*** debe ser una tabla.
- ***"regPersona"*** será un registro que constará de los mismos campos y tipos que las columnas de la tabla personas.

6.10.- Cursores y Registros

USO DE FETCH CON REGISTROS

Una de las desventajas, con lo visto hasta ahora, de utilizar FETCH reside en que necesitamos asignar todos los valores de cada fila del cursor a una variable. Por lo que si una fila tiene 10 columnas, habrá que declarar 10 variables.

En lugar de ello ***se puede utilizar una variable de registro y asignar el resultado de FETCH a esa variable***.

Ejemplo:

```
DECLARE
        CURSOR cursorProvincias IS
                SELECT p.nombre as nombre, SUM(poblacion) AS poblacion
                FROM LOCALIDADES l JOIN PROVINCIAS p USING (n_provincia)
                GROUP BY p.nombre;

                rProvincias cursorProvincias%ROWTYPE;
BEGIN
        OPEN cursorProvincias;
        LOOP
                FETCH cursorProvincias INTO rProvincias;

                EXIT WHEN cursorProvincias%NOTFOUND;
                DBMS_OUTPUT.PUT_LINE( rProvincias.nombre || ',' ||
                                                rProvincias.poblacion);
        END LOOP;
        CLOSE cursorProvincias;
END;
```

BUCLE FOR DE RECORRIDO DE CURSORES

Es la forma más habitual de recorrer todas las filas de un cursor. Es un **bucle FOR** que se encarga de realizar tres tareas:

- ***Abre un cursor*** (realiza un **OPEN implícito sobre el cursor** antes de empezar el bucle).
- ***Recorre todas las filas de un cursor*** (cada vez que se entra en el interior del FOR se genera un **FETCH implícito**) y en cada vuelta del bucle almacena el contenido de cada fila en una variable de registro.
    - ***La variable de registro utilizada en el bucle FOR no se debe declarar en la zona DECLARE***; se crea al inicio del bucle y se elimina cuando éste finaliza.
- ***Cierra el cursor*** (cuando finaliza el FOR)

Sintaxis:

```
FOR variableRegistro IN cursor LOOP
        ..instrucciones
END LOOP;
```

Esa sintaxis es equivalente a:

```
OPEN cursor;
LOOP
    FETCH cursor INTO variableRegistro;
     EXIT WHEN cursor%NOTFOUND;
    ...instrucciones
END LOOP;

CLOSE cursor;
```

Ejemplo (equivalente al ejemplo comentado en los apartados anteriores):

```
DECLARE
        CURSOR cursorProvincias IS
                SELECT p.nombre as nombre, SUM(poblacion) AS poblacion
                FROM LOCALIDADES l
                JOIN PROVINCIAS p USING (n_provincia)
                GROUP BY p.nombre;
```

```
BEGIN
        FOR rProvincias IN cursorProvincias LOOP
                DBMS_OUTPUT.PUT_LINE (rProvincias.nombre || ',' ||
                        rProvincias.poblacion);
        END LOOP;
END;
```

Naturalmente este código es más sencillo de utilizar y más corto que los anteriores.

Importante:

Ten en cuenta que la variable de registro utilizada en el bucle for ("rProvincias" en este ejemplo) NO se tiene que declarar en la sección DECLARE.

6.11.- Cursores Avanzados

CURSORES CON PARÁMETROS

En muchas ocasiones se podría desear que el ***resultado de un cursor dependa de una variable***. Por ejemplo al presentar una lista de personal, hacer que aparezca el cursor de un determinado departamento y puesto de trabajo.

Para hacer que el cursor varíe según esos parámetros, se han de indicar los mismos en la declaración del cursor. Para ello se pone entre paréntesis su nombre y tipo tras el nombre del cursor en la declaración.

Ejemplo:

```
DECLARE
        CURSOR cur_personas(dep NUMBER, pue VARCHAR2(20)) IS
                SELECT nombre, apellidos
                FROM empleados
                WHERE departamento=dep AND puesto=pue ;
BEGIN
        OPEN cur_personas(12, 'administrativo');
        .....
        CLOSE cur_personas;

        .....

        OPEN cur_personas(10, 'gestion');
        .....
```

```
        CLOSE cur_personas;

END;
```

Al abrir el cursor es cuando se indica el valor de los parámetros, lo que significa que se puede abrir varias veces el cursor y que éste obtenga distintos resultados dependiendo del valor del parámetro.

Se pueden **indicar los parámetros también en el bucle FOR**:

```
DECLARE
        CURSOR cur_personas(dep NUMBER, pue VARCHAR2(20)) IS
                SELECT nombre, apellidos FROM empleados
                WHERE departamento=dep AND puesto=pue;
BEGIN
        FOR r IN cur_personas(12, 'administrativo') LOOP
                .....
        END LOOP;
END;
```

ACTUALIZACIONES AL RECORRER REGISTROS

En muchas ocasiones se ***realizan operaciones de actualización de registros (actualizar o suprimir) sobre el cursor que se está recorriendo***. Para evitar problemas se deben **bloquear los registros del cursor** ***para detener otros procesos que también desearan modificar los datos***.

Sintaxis:

```
CURSOR ...
        SELECT...
        FOR UPDATE [OF lista_de_campos] [NOWAIT]
```

- Esta cláusula (**FOR UPDATE** [**OF** lista_de_campos]) se coloca al final de la sentencia SELECT del cursor (iría detrás del ORDER BY).
- Se usa el texto **OF** seguido de la lista de campos que se bloquearán separadas por comas.
- Si no se especifica mediante **OF** qué columna o columnas se van a bloquear, se bloquean todos los registros de todas las tablas involucradas en la consulta.

NOWAIT: devuelve un error inmediatamente si los registros han sido bloqueados por otra sesión. Si el servidor Oracle no puede bloquear los registros que necesita debido a la clausula FOR UPDATE, espera de forma ininterrumpida, a no ser que se haya especificado NOWAIT.

Ejemplo:

```
DECLARE
        CURSOR c_emp IS
                SELECT id_emp, nombre, n_departamento, salario
                FROM empleados, departamentos
                WHERE empleados.id_dep=departamentos.id_dep
                        AND empleados.id_dep=80
                FOR UPDATE OF salario NOWAIT;
```

En este ejemplo se bloquea la columna ***salario*** de la tabla ***empleado*** de todos los empleados del departamento 80. No se bloquearían los registros relacionados de la tabla departamentos.

Ya hemos bloqueado los campos o registros para actualizar y/o borrar información del cursor que estamos recorriendo, pero

¿Cómo se puede realizar dicha actualización dentro del cursor?

Tendríamos 2 posibilidades:

- Recogiendo de la fila actual del cursor su clave primaria y ejecutar la sentencia UPDATE o DELETE de la fila identificada por la clave primaria recogida (¿qué ocurriría si una tabla no tiene clave primaria?).
- Utilizar la clausula **WHERE CURRENT OF** (*seguida del nombre del cursor)* que ***siempre va ligada a*** la clausula FOR UPDATE, que hace que se modifique sólo el registro actual del cursor.

Ejemplo:

```
DECLARE
        CURSOR c_emp IS
                SELECT id_emp, nombre, n_departamento, salario
                FROM empleados, departamentos
                WHERE empleados.id_dep=departamentos.id_dep
                        AND empleados.id_dep=80
                FOR UPDATE OF salario NOWAIT;
```

```
BEGIN
        FOR r_emp IN c_emp LOOP
                IF r_emp.salario<1500 THEN
                        UPDATE empleados SET salario = salario *1.30
                        WHERE CURRENT OF c_emp;
                END IF;
        END LOOP;
        COMMIT;
END;
```

Fíjate bien en:

1. En la declaración del cursor se ha incorporado la ***clausula FOR UPDATE***.
2. En la clausula ***WHERE CURRENT OF*** se ha especificado el nombre del cursor.
3. El ***COMMIT*** está fuera del bucle para evitar que los bloqueos se liberen antes de finalizar todas las sentencias UPDATE que se deban realizar.

7.- Excepciones

7.1.- Introducción

Se llama **excepción** a todo hecho que le sucede a un programa que causa que la ejecución del mismo finalice. Lógicamente eso causa que el ***programa termine de forma anormal***.

Las *excepciones se puede deber* a:

- Que ocurra ***un error detectado por Oracle*** (por ejemplo si un SELECT INTO, que tiene que devolver obligatoriamente un resultado, no devuelve datos ocurre el error ORA-01403 llamado NO_DATA_FOUND).
- Que el propio ***programador lance una excepción*** (comando **RAISE**).

Las excepciones se pueden **capturar** a fin de que el programa controle mejor la existencia de las mismas.

7.2.- Captura de Excepciones

La **captura** se realiza utilizando el ***bloque EXCEPTION*** que es el bloque que está justo antes del END del bloque.

Cuando una excepción ocurre, se comprueba el ***bloque EXCEPTION*** para ver si ha sido capturada, si no se captura, el error ***se propaga a Oracle*** que se encargará de indicar el error existente.

Las excepciones pueden ser de estos **tipos**:

- ***Excepciones predefinidas de Oracle:*** que tienen ya asignado un nombre de excepción.
- ***Excepciones de Oracle sin definir***: no tienen nombre asignado pero se les puede asignar.
- ***Definidas por el usuario***: las lanza el programador.

La captura de excepciones se realiza con esta **sintaxis**:

```
DECLARE
        Sección de declaraciones
BEGIN
        Instrucciones
EXCEPTION
        WHEN excepción1 [OR excepción2 ...] THEN
                Instrucciones que se ejecutan si suceden esas excepciones
        [WHEN excepción3 [OR...] THEN
                Instrucciones que se ejecutan si suceden esas excepciones]
        [WHEN OTHERS THEN
                Instrucciones que se ejecutan si suceden otras  excepciones]
END;
```

Cuando ocurre una determinada excepción, se comprueba el **primer WHEN** para comprobar si el nombre de la excepción ocurrida coincide con el que el WHEN captura; si es así se ejecutan las instrucciones, si no es así se comprueba el **siguiente WHEN** y ***así sucesivamente***.

Si existe la cláusula **WHEN OTHERS**, entonces las excepciones que no estaban reflejadas en los demás apartados WHEN ejecutan las instrucciones del WHEN OTHERS. Ésta cláusula ***debe ser la última***.

7.3.- Excepciones Predefinidas

Oracle tiene las siguientes **excepciones predefinidas**. Son errores (comunes) a los que Oracle asigna un nombre de excepción:

Nombre de excepción	Número de error	Ocurre cuando...
ACCESS_INTO_NULL	ORA-06530	Se intentan asignar valores a un objeto que no se había inicializado.
CASE_NOT_FOUND	ORA-06592	Ninguna opción WHEN dentro de la instrucción CASE captura el valor, y no hay instrucción ELSE.
COLLECTION_IS_NULL	ORA-06531	Se intenta utilizar un varray o una tabla anidada que no estaba inicializada.
CURSOR_ALREADY_OPEN	ORA-06511	Se intenta abrir un cursor que ya se había abierto.
DUP_VAL_ON_INDEX	ORA-00001	Se intentó añadir una fila que provoca que un índice único repita valores.
INVALID_CURSOR	ORA-01001	Se realizó una operación ilegal sobre un cursor.
INVALID_NUMBER	ORA-01722	Falla la conversión de carácter a número
LOGIN_DENIED	ORA-01017	Se intenta conectar con Oracle usando un nombre de usuario y contraseña inválidos.
NO_DATA_FOUND	ORA-01403	El SELECT de fila única no devolvió valores.
PROGRAM_ERROR	ORA-06501	Error interno de Oracle.
ROWTYPE_MISMATCH	ORA-06504	Hay incompatibilidad de tipos entre el cursor y las variables a las que se intentan asignar sus valores.
STORAGE_ERROR	ORA-06500	No hay memoria suficiente.
SUBSCRIPT_BEYOND_COUNT	ORA-06533	Se hace referencia a un elemento de un varray o una tabla anidada usando un índice mayor que los elementos que poseen.
SUBSCRIPT_OUTSIDE_LIMIT	ORA-06532	Se hace referencia a un elemento de un varray o una tabla anidada usando un índice cuyo valor está fuera del rango legal.
SYS_INVALID_ROWID	ORA-01410	Se convierte un texto en un número de identificación de fila (ROWID) y el texto no es válido.
TIMEOUT_ON_RESOURCE	ORA-00051	Se consumió el máximo tiempo en el que Oracle permite esperar al recurso.
TOO_MANY_ROWS	ORA-01422	El SELECT de fila única devuelve más de una fila.
VALUE_ERROR	ORA-06502	Hay un error aritmético, de conversión, de redondeo o de tamaño en una operación.
ZERO_DIVIDE	ORA-01476	Se intenta dividir entre el número cero.

Ejemplo:

```
DECLARE
        x NUMBER :=0;
        y NUMBER := 3;
        res NUMBER;
BEGIN
        res:=y/x;
         DBMS_OUTPUT.PUT_LINE(res);
EXCEPTION
        WHEN ZERO_DIVIDE THEN
                DBMS_OUTPUT.PUT_LINE('No se puede dividir por cero') ;
         WHEN OTHERS THEN
                DBMS_OUTPUT.PUT_LINE('Error inesperado') ;
END;
/
```

7.4.- Excepciones Sin Definir

Pueden ocurrir ***otras muchas excepciones*** que no están en la lista anterior. En ese caso aunque ***no tienen un nombre asignado, sí tienen un número asignado***. Ese número es el que aparece cuando Oracle muestra el mensaje de **error tras la palabra ORA**.

Por ejemplo en un error por restricción de integridad Oracle lanza un mensaje encabezado por el texto: **ORA-02292.** Por lo tanto el error de integridad referencia es el **-02292**.

Si deseamos **capturar excepciones sin definir** hay que:

1. Declarar un nombre para la excepción que capturaremos. Eso se hace en el apartado DECLARE con esta sintaxis:

```
nombreDeExcepción EXCEPTION;
```

2. Asociar ese nombre al número de error correspondiente mediante esta sintaxis en el apartado DECLARE (tras la instrucción del paso 1):

```
PRAGMA EXCEPTION_INIT (nombreDeExcepción, nºDeExcepción);
```

3. En el apartado EXCEPTION capturar el nombre de la excepción como si fuera una excepción normal.

Ejemplo:

```
DECLARE
        e_integridad EXCEPTION;
        PRAGMA EXCEPTION_INIT ( e_integridad, -02292);
BEGIN
        DELETE FROM piezas WHERE tipo='TU' AND modelo=6;
EXCEPTION
        WHEN e_integridad THEN
                DBMS_OUTPUT.PUT_LINE('No se puede borrar esa pieza');
END;
```

7.5. Funciones de uso con Excepciones

Se suelen usar dos funciones cuando se trabaja con excepciones:

- **SQLCODE**: retorna el código de error del error ocurrido.
- **SQLERRM**: devuelve el mensaje de error de Oracle asociado a ese número de error.

Ejemplo:

```
BEGIN
        DELETE FROM pais WHERE upper(nombre) = 'ITALIA'
EXCEPTION
        WHEN OTHERS THEN
                DBMS_OUTPUT.PUT_LINE('Ocurrió el error' || SQLCODE);
                DBMS_OUTPUT.PUT_LINE('Mensaje' || SQLERRM);
END;
```

Oracle mostraría por pantalla:

Ocurrió el error-2292

Mensaje: ORA-02992: restricción de integridad (HR.PIL_PAIS_FK) violada – registro secundario encontrado

7.6. Excepciones de Usuario

El ***programador*** puede ***lanzar sus propias excepciones simulando errores del programa***. Para ello hay que:

1. Declarar un nombre para la excepción en el apartado DECLARE, al igual que para las excepciones sin definir:

```
miExcepcion EXCEPTION;
```

2. En la sección ejecutable (BEGIN) utilizar la instrucción **RAISE para lanzar la excepción**:

```
RAISE miExcepcion;
```

3. En el apartado de excepciones capturar el nombre de excepción declarado:

```
EXCEPTION
        ...
        WHEN miExcepcion THEN
       ....
```

Otra forma es utilizar la **función RAISE_APPLICATION_ERROR** que ***simplifica los tres pasos anteriores***.

Sintaxis:

```
RAISE_APPLICATION_ERROR (nºDeError, mensaje, [, { TRUE | FALSE }]);
```

Esta instrucción se coloca en la sección ejecutable o en la de excepciones y sustituye a los tres pasos anteriores. Lo que hace es ***lanzar un error cuyo número debe de estar entre el -20000 y el -20999*** y hace que ***Oracle muestre el mensaje indicado***. El tercer parámetro opciones puede ser TRUE o FALSE (por defecto TRUE) e indica si el error se añade a la pila de errores existentes.

Ejemplo:

```
BEGIN
        DELETE FROM piezas WHERE tipo='ZU' AND modelo=26;
        IF SQL%NOTFOUND THEN
                RAISE_APPLICATION_ERROR(-20001, 'No existe esa pieza');
        END IF;
END;
```

En el ejemplo, si la pieza no existe, entonces SQL%NOTFOUND devuelve verdadero ya que el DELETE no elimina ninguna pieza.

Se lanza la excepción de usuario -20001 haciendo que Oracle utilice el mensaje indicado. Oracle lanzará el mensaje: ORA-20001: No existe esa pieza.

8. Procedimientos

8.1. Introducción

Un procedimiento es un ***bloque PL/SQL al que se le asigna un nombre***, que se crea para que realice una determinada **tarea de gestión**.

Los procedimientos son compilados y ***almacenados en la base de datos***. Gracias a ellos se consigue una reutilización eficiente del código, ya que se puede invocar al procedimiento las veces que haga falta desde otro código. Una vez almacenados pueden ser modificados de nuevo.

8.2. Estructura de un Procedimiento

La **sintaxis** es:

```
CREATE [OR REPLACE] PROCEDURE nombreProcedimiento
        [(parámetro1 [modelo] tipoDatos
        [,parámetro2 [modelo] tipoDatos [,...])]
{IS | AS}
        secciónDeDeclaraciones
BEGIN
        instrucciones
[EXCEPTION
        controlDeExcepciones]
END;
```

- La **opción REPLACE** hace que si ya existe un procedimiento con ese nombre, se reemplaza con el que se crea ahora.
- Los **parámetros** son la ***lista de variables*** que necesita el procedimiento para realizar su tarea:
    - Al declarar cada parámetro se indica **el tipo** de los mismos, **pero no su tamaño**; es decir sería VARCHAR2 y no VARCHAR2(50).

- El apartado opcional **modelo**, se elige si el parámetro es de tipo ***IN, OUT o IN OUT*** (se explica más adelante).

- ***No se utiliza la palabra DECLARE*** para indicar el inicio de las declaraciones. No obstante la ***sección de declaraciones figura tras las palabras IS o AS*** (es decir justo antes del BEGIN es donde debemos declarar las variables).

8.3. Desarrollo de Procedimientos

Los pasos para desarrollar procedimientos son:

1. Escribir el código en un archivo .sql desde cualquier editor.
2. ***Compilar el código*** desde un editor como iSQL*Plus o cualquier otro que realice esa tarea. El resultado es el llamado código P, el procedimiento estará creado.
3. Ejecutar el procedimiento para realizar su tarea, eso se puede hacer las veces que haga falta (en ***iSQL*Plus***, el comando que ejecuta un procedimiento es el ***comando EXECUTE***, en otros entornos se suele ***crear un bloque anónimo que incorpore una llamada al procedimiento***).

8.4. Parámetros

Los procedimientos permiten utilizar ***parámetros para realizar su tarea***.

Por ejemplo supongamos que queremos crear el procedimiento ESCRIBIR para escribir en el servidor (como hace DBMS_OUTPUT.PUT_LINE) lógicamente dicho procedimiento necesita saber lo que queremos escribir. Ese sería el parámetro, de esa forma el procedimiento sería:

```
CREATE OR REPLACE PROCEDURE Escribir(texto VARCHAR)
IS
BEGIN
        DBMS_OUTPUT.PUT_LINE (texto);
END;
/
```

Para **invocarle**:

```
BEGIN
        …
        Escribir ('Hola');
        …
END;
/
```

También se podría invocar directamente mediante el comando "execute":

EXECUTE Escribir('Hola');

O utilizar "exec" (*la forma abreviada del comando execute*):

EXEC Escribir('Hola');

Cuando se invoca a un procedimiento, si éste no tiene parámetros, se pueden omitir los paréntesis (es decir la llamada al procedimiento actualizar() se puede hacer simplemente escribiendo actualizar, sin paréntesis).

TIPOS DE PARÁMETROS: IN, OUT, IN OUT

Hay ***tres tipos de parámetros*** en PL/SQL:

1. **Parámetros IN**:
   - Son los parámetros que en otros lenguajes se denominan como ***parámetros por valor***. *El procedimiento recibe una* ***copia del valor o variable que se utiliza como parámetro*** *al llamar al procedimiento*.
   - Estos parámetros pueden ser: valores literales (18 por ejemplo), variables (v_num por ejemplo) o expresiones (como v_num+18). A estos parámetros se les puede asignar un valor por defecto.

2. **Parámetros OUT**:
   - Relacionados con el paso por variable de otros lenguajes.
   - Sólo pueden ser variables y no pueden tener un valor por defecto.
   - Se utilizan para que el procedimiento almacene en ellas algún valor. Es decir, los ***parámetros OUT son variables*** sin declarar que se envían al procedimiento ***de modo que si en el procedimiento cambian su valor, ese valor permanece en ellas cuando el procedimiento termina.***

3. **Parámetros IN OUT**:
   - Son una mezcla de los dos anteriores.
   - Se trata de variables declaradas anteriormente cuyo valor puede ser utilizado por el procedimiento que, además, puede almacenar un valor en ellas. No se las puede asignar un valor por defecto.

Se pueden especificar estas palabras en la declaración del procedimiento (es el modo del procedimiento). **Si no se indica modo alguno, se supone que se está utilizando IN** (que es el que más se usa).

Ejemplo:

```
CREATE OR REPLACE PROCEDURE  consultarEmpresa
        (v_nombre VARCHAR2, v_CIF OUT VARCHAR2, v_dir OUT VARCHAR2)
IS
BEGIN
        SELECT cif, direccion INTO v_CIF, v_dir
        FROM EMPRESAS
        WHERE nombre LIKE '%' || v_nombre || '%';
EXCEPTION
        WHEN NO_DATA_FOUND THEN
                DBMS_OUTPUT.PUT_LINE('No se encontraron datos');
        WHEN TOO_MANY_ROWS THEN
                DBMS_OUTPUT.PUT_LINE ('Hay más de una fila con esos datos');
END;
```

El procedimiento consulta las empresas cuyo nombre tenga el texto enviado en **v_nombre**, captura los posibles errores y en caso de que la consulta se ejecute correctamente ***almacena el cif y la dirección de la empresa en los parámetros v_CIF y v_dir***.

La **llamada al procedimiento** anterior podría ser:

```
DECLARE
        v_cifEmpresa VARCHAR2(50);
        v_dirEmpresa VARCHAR2(50);
BEGIN
        consultarEmpresa('Hernández', v_cifEmpresa ,v_dirEmpresa);
        DBMS_OUTPUT.PUT_LINE(v_cifEmpresa);
        DBMS_OUTPUT.PUT_LINE(v_dirEmpresa);
END;
```

Las variables v_cifEmpresa y v_dirEmpresa almacenarán (si existe una sola empresa con el texto Hernández) el CIF y la dirección de la empresa buscada.

Los procedimientos no pueden leer los valores que posean las variables OUT, sólo escribir en ellas. Si se necesitan ambas cosas es cuando hay que declararlas con IN OUT.

8.5. Borrar Procedimientos

El comando **DROP PROCEDURE** seguido del nombre del procedimiento que se elimina es el encargado de realizar esta tarea.

9. Funciones

9.1. Introducción

Las **funciones** son un ***tipo especial de procedimiento*** que sirven ***para calcular un determinado valor***.

Todo lo comentado en el apartado anterior es válido para las funciones, la diferencia estriba sólo en que éstas **devuelven un valor**.

9.2. Sintaxis

```
CREATE [OR REPLACE] FUNCTION nombreFunción
        [(parámetro1 [modelo] tipoDatos
        [,parámetro2 [modelo] tipoDatos [,...]])]
        RETURN tipoDeDatos
IS | AS
        secciónDeDeclaraciones
BEGIN
        instrucciones
[EXCEPTION
        controlDeExcepciones]
END;
```

Si comparamos con la declaración de las funciones:

- La palabra PROCEDURE se modifica por la palabra **FUNCTION** (indicando que es una función y no un procedimiento).
- Aparece la **cláusula RETURN** justo antes de la palabra IS que sirve para ***indicar el tipo de datos que poseerá el valor retornado por la función***.

9.3. Uso de Funciones

Las funciones se crean igual que los procedimientos y, al igual que éstos, ***se almacenan en la base de datos***.

Toda **función** ha de ***devolver un valor***, lo cual implica utilizar la **instrucción RETURN seguida del valor que se devuelve**.

Ejemplo:

```
CREATE OR REPLACE FUNCTION cuadrado (x NUMBER)
        RETURN NUMBER
IS
BEGIN
        RETURN x*x;
END;
/
```

La función descrita calcula el cuadrado de un número.

Una posible **llamada a la función** podría ser:

```
BEGIN
        DBMS_OUTPUT.PUT_LINE (cuadrado(9));
END;
/
```

Otra posibilidad:

```
DECLARE
        resultado NUMBER(8,0);
BEGIN
        resultado := cuadrado(9);
        DBMS_OUTPUT.PUT_LINE ('El resultado es: ' || resultado);
END;
/
```

Las funciones de PL/SQL se utilizan como las funciones de cualquier lenguaje estructurado, se pueden asignar a una variable, utilizar para escribir, etc. Además dentro de una función se puede invocar a otra función.

```
// Esta invocación no sería correcta:
execute   cuadrado(5);

// Sin embargo la siguiente, sí sería correcta:
execute DBMS_OUTPUT.PUT_LINE (cuadrado(5));
```

9.4. Utilizar funciones desde SQL

Una gran ventaja de las funciones es la ***posibilidad de utilizarlas desde una instrucción SQL***.

Ejemplo:

```
CREATE OR REPLACE FUNCTION precioMedio
        RETURN NUMBER
IS
        v_precio NUMBER(11, 4);
BEGIN
        SELECT AVG (precio_venta) INTO v_precio FROM PIEZAS;
        RETURN v_precio;
END;
```

Esta función devuelve el precio medio de la tabla de piezas. Una vez compilada y almacenada la función, se puede invocar desde una instrucción SQL cualquiera.

Por **ejemplo**:

```
SELECT * FROM PIEZAS
WHERE (precio_venta < precioMedio);
```

Esa consulta obtiene los datos de las piezas cuyo precio sea menor que el precio medio.

Hay que tener en cuenta que para que ***las funciones puedan ser invocadas desde SQL, éstas tienen que cumplir que***:

- Sólo valen funciones que se hayan almacenado.
- Sólo pueden utilizar parámetros de tipo IN.
- Sus parámetros deben ser de tipos compatibles con el lenguaje SQL (no valen tipos específicos de PL/SQL como BOOLEAN por ejemplo).
- El tipo devuelto debe ser compatible con SQL.
- No pueden contener instrucciones DML.
- Si una instrucción DML modifica una determinada tabla, en dicha instrucción no se puede invocar a una función que realice consultas sobre la misma tabla.

- No pueden utilizar instrucciones de transacciones (COMMIT, ROLLBACK,...).
- La función no puede invocar a otra función que se salte alguna de las reglas anteriores.

9.5. Eliminar Funciones

Sintaxis:

```
DROP FUNCTION nombreFunción;
```

9.6. Recursividad

En PL/SQL la ***recursividad*** (el hecho de que ***una función pueda llamarse a sí misma***) está permitida.

El siguiente código es válido:

```
CREATE FUNCTION Factorial (n NUMBER)
IS
BEGIN
        IF (n<=1) THEN
                RETURN 1;
        ELSE
                RETURN n * Factorial(n-1);
        END IF;
END;
```

9.7. Mostrar Procedimientos almacenados

La vista USER_PROCEDURES, contiene una fila por cada procedimiento o función que tenga almacenado el usuario actual.

Investiga qué columnas contiene la vista USER_PROCEDURES y para qué sirven o qué nos indican.

10. Paquetes

10.1. Introducción

Los paquetes sirven para ***agrupar bajo un mismo nombre funciones y procedimientos***. Facilitan la modularización de programas y su mantenimiento.

Los **paquetes constan de dos partes**: **especificación y cuerpo** que serán ***almacenadas por separado*** en la base de datos.

1. **Especificación**:
    - Sirve para ***declarar los elementos de los que consta el paquete***.
    - En esta especificación se indican los ***procedimientos, funciones y variables públicos del paquete*** (los que se podrán invocar desde fuera del paquete).
    - De los ***procedimientos sólo se indica su nombre y parámetros*** (sin el cuerpo).
2. **Cuerpo**:
    - Sirve para especificar el ***funcionamiento del paquete***, es decir para implementar las especificaciones (definir completamente lo que sea declarado en la especificación del paquete).
    - Consta de la definición de los procedimientos indicados en la especificación. Además se pueden declarar y definir variables y procedimientos privados (sólo visibles para el cuerpo del paquete, no se pueden invocar desde fuera del mismo).

Los paquetes se editan y se compilan al igual que los procedimientos y funciones.

10.2. Creación de Paquetes

Conviene ***almacenar la especificación y el cuerpo del paquete en dos archivos de texto (.sql) distintos*** para su posterior mantenimiento.

Sintaxis:

```
CREATE [OR REPLACE] PACKAGE nombrePaquete
{IS | AS}
        Variables, constantes, cursores y excepciones públicas
        Cabecera de procedimientos y funciones
END nombrePaquete;
```

Ejemplo:

```
CREATE OR REPLACE PACKAGE paquete1
IS
        v_cont NUMBER := 0;
        PROCEDURE reset_cont (v_nuevo_cont NUMBER);
        FUNCTION devolver_cont RETURN NUMBER;
END paquete1;
/
```

De las funciones hay que indicar sus parámetros y el tipo de datos que devuelve.

CUERPO

```
CREATE [OR REPLACE] PACKAGE BODY nombrePaquete
IS | AS
        variables, constantes, cursores y excepciones privadas
        cuerpo de los procedimientos y funciones
END nombrePaquete;
```

Ejemplo:

```
CREATE OR REPLACE PACKAGE BODY paquete1
IS
        PROCEDURE reset_cont (v_nuevo_cont NUMBER)
        IS
        BEGIN
                v_cont:=v_nuevo_cont;
        END reset_cont;

        FUNCTION devolver_cont RETURN NUMBER
        IS
        BEGIN
                RETURN v_cont;
        END devolver_cont;

END paquete1;
/
```

USO DE LOS OBJETOS DEFINIDOS EN LOS PAQUETES

Desde ***dentro del paquete***, para utilizar otra función o procedimiento o variable dentro del mismo paquete, basta con ***invocarla por su nombre***.

Si queremos utilizar un ***objeto de un paquete, fuera del mismo***, entonces se antepone el nombre del paquete a la función o procedimiento. Por ejemplo ***paquete1.reset_cont(4)*** (en el ejemplo anterior).

Para ejecutar un procedimiento de un paquete desde SQL*Plus, se usa la orden EXECUTE. Por ejemplo: ***EXECUTE paquete1.reset_cont(4)***

Si se quiere modificar una variable "global" del paquete también hay que usar la orden EXECUTE. Por ejemplo: ***EXECUTE paquete1.v_cont:=16;***

IMPORTANTE

- Crea ***paquetes flexibles***: es decir que sean lo más generales posibles y se puedan ***reutilizar*** en futuras aplicaciones.
- Define la ***especificación del paquete antes que el cuerpo***. Las especificaciones reflejan el diseño de la aplicación, por lo que hay que definirlas antes que el cuerpo.
- La ***especificación del paquete solo debe contener construcciones públicas***: sólo los subprogramas que deben ser visibles para los usuarios del paquete. De esta forma, otro usuario no podrá hacer un mal uso de nuestro paquete.
- La ***especificación del paquete debería contener el mínimo número de construcciones***, para reducir la necesidad de compilar cuando se cambia el código:
    - Los cambios en el cuerpo del paquete no requieren re-compilación de programas/paquetes dependientes.
    - Pero los cambios en la especificación sí lo requieren. Cada subprograma almacenado de forma independiente o en algún paquete que referencie al nuestro deberá ser recompilado. Oracle lo advierte mediante los mensajes de error:
        - ***ORA-04068: se ha anulado el estado existente de los paquetes.***
        - ***ORA-04061: el estado existente del paquete xxxxx ha sido invalidado.***

10.3. Eliminar paquetes

Sintaxis:

```
// Para borrar la especificación:
DROP PACKAGE nombre_del_paquete;

// Para borrar el cuerpo:
DROP PACKAGE BODY nombre_del_paquete;
```