

Cursos, excepciones y control de transacciones en PL/SQL

10

En esta unidad aprenderás a:

- 1 Utilizar cursos explícitos e implícitos para procesar la información contenida en la base de datos.
- 2 Diseñar programas robustos, capaces de recuperarse ante las condiciones de error que puedan aparecer durante la ejecución, utilizando las técnicas de tratamiento de errores y gestión de excepciones que proporciona PL/SQL.
- 3 Garantizar la integridad de la información utilizando los comandos de control de transacciones.



10.1 Introducción

En las unidades anteriores hemos estudiado los fundamentos del lenguaje, sus características básicas y aspectos lexicográficos y sintácticos. En esta unidad aprenderemos a manejar el lenguaje PL/SQL en su entorno natural: la gestión segura y eficiente de grandes volúmenes de información de la base de datos.

Para ello, PL/SQL aporta un conjunto de estructuras y comandos que podemos resumir en: la *utilización de cursores*, el *manejo de errores* y el *control de transacciones*. Todos ellos se han comentado someramente en la introducción al lenguaje. En esta unidad profundizaremos en sus características y utilización.

10.2 Cursores

Hasta el momento hemos venido utilizando *cursores implícitos*. Son muy cómodos y sencillos pero plantean diversos problemas. Lo más importante es que la subconsulta debe envolver una fila (y sólo una), de lo contrario, se produciría un error. Por ello, dado que normalmente una consulta devolverá varias filas, se suelen manejar cursores explícitos.

A. Cursores explícitos

Se utilizan para trabajar con consultas que pueden devolver más de una fila.

Hay cuatro operaciones básicas para trabajar con un cursor explícito:

1. **Declaración** del cursor en la zona de declaraciones según el siguiente formato:

```
CURSOR <nombrecursor> IS <sentencia SELECT>;
```

2. **Apertura** del cursor en la zona de instrucciones:

```
OPEN <nombrecursor>;
```

La instrucción OPEN ejecuta automáticamente la sentencia SELECT asociada y sus resultados se almacenan en las estructuras internas de memoria manejadas por el cursor.

No obstante, para acceder a la información debemos dar el paso siguiente.

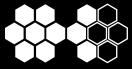
3. **Recogida de información almacenada** en el cursor. Se usa el comando FETCH con el siguiente formato:

```
FETCH <nombrecursor> INTO {<variable>|<listavariables>} ;
```

Después de INTO figurará:

Las operaciones permitidas con el cursor son: *declarar, abrir, recoger información y cerrar el cursor*. No se le pueden asignar valores ni utilizarlo en expresiones.

En realidad, un **cursor** es una estructura que apunta a una región de la PGA que contiene toda la información relativa a la consulta asociada, en especial, las filas de datos recuperadas.



10. Cursores, excepciones y control de transacciones ...

10.2 Cursores

- Una variable que recogerá la información de todas las columnas. Puede declararse de esta forma:

```
<variable> <nombrecursor>%ROWTYPE;
```

- O una lista de variables. Cada una recogerá la columna correspondiente de la cláusula SELECT; por tanto, serán del mismo tipo que las columnas.

Cada FETCH recupera una fila y el cursor avanza automáticamente a la fila siguiente en cada nueva instrucción FETCH.

4. **Cierre del cursor.** Cuando el cursor no se va a utilizar hay que cerrarlo:

```
CLOSE <nombrecursor>;
```

El siguiente ejemplo utiliza un cursor explícito para visualizar el nombre y la localidad de todos los departamentos de la tabla DEPART.

```
DECLARE
    CURSOR curl IS
        SELECT dnombre, loc FROM depart;      --1.Declarar
        v_nombre VARCHAR2(14);
        v_localidad VARCHAR2(14);
BEGIN
    OPEN curl;                            --2.Abrir
    LOOP
        FETCH curl INTO v_nombre, v_localidad; --3.Recoger
        EXIT WHEN curl%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE (v_nombre ||'*'||v_localidad);
    END LOOP;
    CLOSE curl;                          --4.Cerrar
END;
```

El resultado de la ejecución de este bloque será:

```
CONTABILIDAD*SEVILLA
INVESTIGACION*MADRID
VENTAS*BARCELONA
PRODUCCION*BILBAO
```

Observamos que:

- La sentencia SELECT en la declaración del cursor no contiene la cláusula INTO a diferencia de lo que ocurre en los cursores implícitos.
- La instrucción FETCH se usa para recuperar cada una de las filas seleccionadas por la consulta. Esta información se deposita en las variables que siguen a la cláusula INTO.
- Después de un FETCH debe comprobarse el resultado, lo que se suele hacer preguntando por el valor de alguno de los atributos del cursor que se mencionan en el siguiente apartado.



B. Atributos del cursor

Hay cuatro atributos para consultar detalles de la situación del cursor:

- **%FOUND.** Devuelve verdadero si el último FETCH ha recuperado algún valor; en caso contrario, devuelve falso. Si el cursor no estaba abierto devuelve error, y si estaba abierto pero no se había ejecutado aún ningún FETCH, devuelve NULL. Se suele utilizar como condición de continuación en bucles. En el ejemplo anterior se puede sustituir el bucle y la condición de salida por:

```
...
BEGIN
  OPEN curl;
  FETCH curl INTO v_nombre, v_localidad;
  WHILE curl%FOUND LOOP
    DBMS_OUTPUT.PUT_LINE (v_nombre ||'*'||v_localidad);
    FETCH curl INTO v_nombre, v_localidad;
  END LOOP;
  CLOSE curl;
END
```

- **%NOTFOUND.** Hace lo contrario que el atributo anterior. Se suele utilizar como condición de salida en bucles:

```
...
EXIT WHEN curl%NOTFOUND;
...
```

- **%ROWCOUNT.** Devuelve el número de filas recuperadas hasta el momento por el cursor (número de FETCH realizados satisfactoriamente).
- **%ISOPEN.** Devuelve verdadero si el cursor está abierto.

La siguiente tabla muestra los valores de retorno de los atributos del cursor en diferentes situaciones:

		%FOUND	%ISOPEN	%NOTFOUND	%ROWCOUNT
OPEN	Antes Después	Invalid_cursor NULL	F T	Invalid_cursor NULL	Invalid_cursor 0
PRIMER FETCH	Antes Después	NULL T	T T	NULL F	0 1
SIGUIENTES FETCH	Antes Después	T T	T T	F F	1 ...
ULTIMO FETCH	Antes Después	T F	T T	F T	N N
CLOSE	Antes Después	F Invalid_cursor	T F	T Invalid_cursor	N Invalid_cursor

Tabla 10.1. Valores de retorno de los atributos del cursor en diferentes situaciones.



10. Cursos, excepciones y control de transacciones ...

10.2 Cursos



Caso práctico

- 1 El siguiente ejemplo ilustra lo que hemos visto hasta ahora respecto a cursos y atributos de cursor: se trata de visualizar los apellidos de los empleados pertenecientes al departamento 20 numerándolos secuencialmente.

```
DECLARE
    CURSOR c1 IS
        SELECT apellido FROM emple WHERE dept_no=20;
        v_apellido VARCHAR2(10);
BEGIN
    OPEN c1;
    LOOP
        FETCH c1 INTO v_apellido;
        DBMS_OUTPUT.PUT_LINE(TO_CHAR(c1%ROWCOUNT, '99.') ||
                               || v_apellido);
        EXIT WHEN c1%NOTFOUND;
    END LOOP;
    CLOSE C1;
END;
```

El resultado de la ejecución será:

```
1 . SANCHEZ
2 . JIMENEZ
3 . GIL
4 . ALONSO
5 . FERNANDEZ
5 . FERNANDEZ
```

En este ejemplo observamos que el último FETCH no devuelve ningún valor, no incrementa el atributo %ROWCOUNT y no sobrescribe el valor de la variable del cursor. Es evidente que el programa está mal diseñado, ya que procesa (visualiza) la información supuestamente recuperada antes de comprobar si se ha recuperado información nueva.

En caso de que FETCH no recupere una nueva fila:

- No se incrementará el atributo %ROWCOUNT.
- No se sobrescribe el valor de la variable del cursor.



Actividades propuestas

- 1 Escribe el ejercicio anterior subsanando el error de diseño para que no aparezca el último empleado duplicado. Hacerlo primero manteniendo el bucle LOOP...EXIT WHEN, y posteriormente probar con un bucle WHILE. Observa las diferencias en ambos casos.



C. Variables de acoplamiento en el manejo de cursos

La cláusula SELECT del cursor deberá seleccionar frecuentemente las filas de acuerdo con una condición. Cuando se trabaja con SQL interactivo se introducen los términos exactos de la condición. Veamos un ejemplo:

```
SQL> SELECT apellido FROM emple
      WHERE dept_no = 20;
```

Pero en un programa PL/SQL los términos exactos de esta condición solamente se conocen en tiempo de ejecución. Esta circunstancia obliga a utilizar un diseño más abierto. Las variables de acoplamiento cumplen esta función. Su forma de uso suele ser:

1. Se declara la variable como cualquier otra.
2. Se utiliza la variable en la sentencia SELECT como parte de la expresión.

```
CURSOR nombrecursor IS clausulaselectconvariableacoplam;
```

Caso práctico

- 2 En el siguiente ejemplo se visualizan los empleados de un departamento cualquiera usando variables de acoplamiento:**

```
CREATE OR REPLACE PROCEDURE ver_emple_por_dept (
  dep VARCHAR2)
AS
  v_dept NUMBER(2);
  CURSOR c1 IS
    SELECT apellido FROM emple WHERE dept_no = v_dept;
  v_apellido VARCHAR2(10);
BEGIN
  v_dept := dep;
  OPEN c1;
  FETCH c1 INTO v_apellido;
  WHILE c1%FOUND LOOP
    DBMS_OUTPUT.PUT_LINE(v_apellido);
    FETCH c1 INTO v_apellido;
  END LOOP;
  CLOSE c1;
END;
```

El programa sustituirá la variable por su valor en el momento en que se abre el cursor, y se seleccionarán las filas según dicho valor. Aunque ese valor cambie durante la recuperación de los datos con FETCH, el conjunto de filas que contiene el cursor no variará.

También podíamos haber usado directamente el parámetro formal *dep* en lugar de la variable *v_dept*. El resultado es el mismo.

(Continúa)



10. Cursos, excepciones y control de transacciones ...

10.2 Cursos

(Continuación)

Una vez creado el procedimiento, se puede ejecutar:

```
SQL> EXECUTE ver_emple_por_dept(30);  
ARROYO  
SALA  
MARTIN  
NEGRO  
TOVAR  
JIMENO
```



Actividades propuestas

- 2 Escribe un procedimiento que reciba una cadena y visualice el apellido y el número de empleado de todos los empleados cuyo apellido contenga la cadena especificada. Al finalizar, visualiza el número de empleados mostrados. El procedimiento empleará variables de acoplamiento para la selección de filas y los atributos del cursor estudiados en el epígrafe anterior.

D. Cursos FOR...LOOP

Al escribir la sentencia SELECT del cursor debemos cuidar que seleccione únicamente aquellas filas con las que va a trabajar el programa pues de lo contrario sobrecargaremos el sistema y deberemos emplear código adicional para filtrar o tratar las filas requeridas.

Como ya hemos visto, en muchas ocasiones el trabajo con un cursor consiste en:

- Declarar el cursor.
- Declarar una variable que recogerá los datos del cursor.
- Abrir el cursor.
- Recuperar con FETCH una a una las filas extraídas, introduciendo los datos en la variable, procesándolos y comprobando también si se han recuperado datos o no.
- Cerrar el cursor.

La estructura de cursos FOR...LOOP simplifica estas tareas realizando todas ellas, excepto la declaración del cursor, de manera implícita.

El formato y el uso de esta estructura es:

1. Se declara el cursor en la sección declarativa (como cualquier otro cursor).

```
CURSOR <nombrecursor> IS <sentencia SELECT>;
```



2. Se procesa el cursor utilizando el siguiente formato:

```
FOR <nombrevareg> IN <nombrecursor> LOOP
  ...
END LOOP;
```

Donde *nombrevareg* es el nombre de la variable de registro que creará el bucle para recoger los datos del cursor.

Al entrar en el bucle:

- Se abre el cursor de manera automática.
- Se declara implícitamente la variable *nombrevareg* de tipo *nombrevareg %ROWTYPE* y se ejecuta un FETCH implícito, cuyo resultado quedará en *nombrevareg*.
- A continuación, se realizarán las acciones que correspondan hasta llegar al END LOOP, que sube de nuevo al FOR...LOOP ejecutando el siguiente FETCH implícito, y depositando otra vez el resultado en *nombrevareg*, y así sucesivamente, hasta procesar la última fila de la consulta. En ese momento, se producirá la salida del bucle y se cerrará automáticamente el cursor.

En el siguiente ejemplo, se visualizan el apellido, el oficio y la comisión de los empleados cuya comisión supera 500 € utilizando CURSOR FOR...LOOP:

```
DECLARE
  CURSOR mi_cursor IS
    SELECT apellido, oficio, comision FROM emple
      WHERE comision > 500;
BEGIN
  FOR v_reg IN mi_cursor LOOP
    DBMS_OUTPUT.PUT_LINE(v_reg.apellido||'*'||v_reg.oficio||'*'||TO_CHAR(v_reg.comision));
  END LOOP;
END;
```

El resultado será:

```
SALA*VENDEDOR*650
MARTIN*VENDEDOR*1020
```

Cabe subrayar que la variable *nombrevareg* se declara implícitamente y es local al bucle; por tanto, al salir del bucle, la variable de registro no estará disponible.

Dentro del bucle se puede hacer referencia a la variable de registro y a sus campos (cuyo nombre se corresponde con las columnas de la consulta) usando la notación de punto.

El siguiente ejemplo muestra las diferencias en el uso de cursosres FOR...LOOP con otras estructuras convencionales.



10. Cursos, excepciones y control de transacciones ...

10.2 Cursos



Caso práctico

- 3 Escribiremos un bloque PL/SQL que visualice el apellido y la fecha de alta de todos los empleados ordenados por fecha de alta.

1. Mediante una estructura cursor FOR...LOOP.

```
DECLARE
    CURSOR c_emple IS
        SELECT apellido, fecha_alt FROM emple
        ORDER BY fecha_alt;
BEGIN
    FOR v_reg_emp IN c_emple LOOP
        DBMS_OUTPUT.PUT_LINE(v_reg_emp.apellido || '*' ||
                             v_reg_emp.fecha_alt);
    END LOOP;
END;
```

2. Utilizando un bucle WHILE.

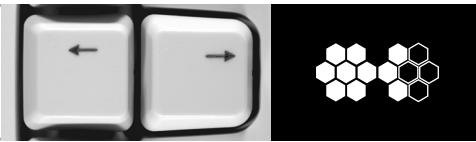
```
DECLARE
    CURSOR c_emple IS
        SELECT apellido, fecha_alt FROM emple
        ORDER BY fecha_alt;
    v_reg_emp c_emple%ROWTYPE;
BEGIN
    OPEN c_emple;
    FETCH c_emple INTO v_reg_emp;
    WHILE c_emple%FOUND LOOP
        DBMS_OUTPUT.PUT_LINE(v_reg_emp.apellido || '*' ||
                             v_reg_emp.fecha_alt);
        FETCH c_emple INTO v_reg_emp;
    END LOOP;
    CLOSE c_emple;
END;
```

Podemos observar que el bucle FOR...LOOP realiza implícitamente la mayoría de las operaciones con el cursor (abrir, comprobar, recuperar fila y cerrar).



Actividades propuestas

- 3 Escribe el procedimiento realizado anteriormente en la actividad 2 pero usando un cursor FOR...LOOP. Observa las diferencias con la estructura anterior. Debemos tener en cuenta que el cursor estará cerrado al salir del bucle y no estarán disponibles sus atributos (en concreto %ROWCOUNT).



E. Uso de alias en las columnas de selección del cursor

Ya hemos indicado que cuando utilizamos variables de registro declaradas del mismo tipo que el cursor o que la tabla, los campos tienen el mismo nombre que las columnas correspondientes. Cuando esas consultas son expresiones, se puede presentar un problema al tratar de referenciarlas:

```
CURSOR c1 IS
  SELECT dept_no, count(*), sum(salario+NVL(comision,0))
    FROM emple
   GROUP BY dept_no;
```

En estos casos, debemos indicar un alias en la columna:

```
CURSOR c1 IS
  SELECT dept_no, count(*) n_emp, sum(salario+NVL(comision,0)) suma
    FROM emple
   GROUP BY dept_no;
```

F. Cursosres con parámetros

En lugar de usar variables de acoplamiento podemos usar parámetros para determinar los términos exactos de la sentencia SELECT cada vez que se abre el cursor.

La declaración de un cursor con parámetros se realiza indicando la lista de parámetros entre paréntesis a continuación del nombre del cursor. Los parámetros declarados se usarán en la sentencia SELECT de la declaración tal como se muestra a continuación:

```
CURSOR <nombrerecursor> [ (parámetro1, parámetro2, ...) ]
IS SELECT <sentencia select con parámetros>;
```

Los parámetros formales indicados después del nombre del cursor tienen la siguiente sintaxis:

```
<Nombredeparámetro> [IN] <tipodedato> [{ := | DEFAULT }
<valor>]
```

Los parámetros formales de un cursor son parámetros de entrada. El ámbito de estos parámetros es local al cursor, por eso solamente pueden ser referenciados dentro de la consulta.

```
DECLARE
  ...
CURSOR cur1
  (v_departamento NUMBER,
  v_oficio VARCHAR2 DEFAULT 'DIRECTOR')
IS SELECT apellido, salario FROM emple
 WHERE dept_no = v_departamento AND oficio = v_oficio;
```

El uso de parámetros permite que cualquier programa pueda llamar al cursor sin necesidad de conocer y/o manipular las variables de acoplamiento.



10. Cursos, excepciones y control de transacciones ...

10.2 Cursos

La apertura del cursor pasándole parámetros se hará:

```
OPEN nombrecursor [ ( parámetro1, parámetro2, ... ) ];
```

Donde parámetro1, parámetro2, ... son expresiones que contienen los valores que se pasarán al cursor. No tienen por qué ser los mismos nombres de las variables indicadas como parámetros al declarar el cursor; es más, si lo fueran, serían consideradas como variables distintas.

Supongamos, por ejemplo, las siguientes declaraciones:

```
DECLARE
    v_dep emple.dept_no%TYPE;
    v_ofi emple.oficio%TYPE;
    CURSOR cur1
        (v_departamento NUMBER,
         v_oficio VARCHAR2 DEFAULT 'DIRECTOR')
        IS SELECT apellido, salario FROM emple
        WHERE dept_no = v_departamento AND oficio = v_oficio;
        ...
        
```

Cualquiera de los siguientes comandos abrirá el cursor:

```
BEGIN
    ...
    OPEN cur1(v_dep);
    OPEN cur1(v_dep, v_ofi);
    OPEN cur1(20, 'VENDEDOR');
    ...
    
```

Debemos recordar que:

- Los parámetros formales de un cursor son siempre IN y no devuelven ningún valor ni pueden afectar a los parámetros actuales.
- La recogida de datos se hará, igual que en otros cursos explícitos, con FETCH.
- La cláusula WHERE asociada al cursor se evalúa solamente en el momento de abrir el cursor. En ese momento es cuando se sustituyen las variables por su valor.

En el caso de los cursos FOR...LOOP, puesto que la instrucción OPEN va implícita, el paso de parámetros se hará a continuación del identificador del cursor en la instrucción FOR...LOOP, tal como se muestra en el ejemplo:

```
...
FOR reg_emple IN cur1(20, 'DIRECTOR') LOOP
    ...
    
```



E. Cursos y rupturas de secuencia

Como acabamos de estudiar, el uso de cursos facilita el procesamiento secuencial de los datos recuperados. Esto incluye la posibilidad de rupturas de secuencia según el criterio o criterios determinados por el problema, que deberán ser los criterios de ordenación. A continuación, escribiremos un ejemplo de aplicación de esta técnica.

Caso práctico

- 4 Escribe un programa que muestre, en formato similar a las rupturas de control o secuencia vistas en SQL*Plus los siguientes datos:**

- Para cada empleado: apellido y salario.
- Para cada departamento: número de empleados y suma de los salarios del departamento.
- Al final del listado: número total de empleados y suma de todos los salarios.

```

CREATE OR REPLACE PROCEDURE listar_emple
AS
CURSOR c1 IS
    SELECT apellido, salario, dept_no FROM emple
ORDER BY dept_no, apellido;
    vr_emp c1%ROWTYPE;
    dep_ant EMPLE.DEPT_NO%TYPE DEFAULT 0;
    cont_emple NUMBER(4) DEFAULT 0;
    sum_sal NUMBER(9,2) DEFAULT 0;
    tot_emple NUMBER(4) DEFAULT 0;
    tot_sal NUMBER(10,2) DEFAULT 0;
BEGIN
    OPEN c1;
    LOOP
        FETCH c1 INTO vr_emp;
        /* Si es el primer Fetch inicializamos dep_ant */
        IF c1%ROWCOUNT = 1 THEN
            dep_ant := vr_emp.dept_no;
        END IF;
        /* Comprobación nuevo departamento (o finalización) y resumen del anterior e inici-
cialización de contadores y acumuladores parciales */
        IF dep_ant <> vr_emp.dept_no OR c1%NOTFOUND THEN
            DBMS_OUTPUT.PUT_LINE('*** DEPTO: ' || dep_ant ||
                ' NUM. EMPLEADOS: ' || cont_emple ||
                ' SUM. SALARIOS: ' || sum_sal);
            dep_ant := vr_emp.dept_no;
            tot_emple := tot_emple + cont_emple;
            cont_emple := 0;
            sum_sal := 0;
        END IF;
    END LOOP;
END;

```

(Continúa)



(Continuación)

```
tot_sal := tot_sal + sum_sal;
cont_emple := 0;
sum_sal := 0;
END IF;
EXIT WHEN c1%NOTFOUND; /* Condición de salida del bucle */

/* Escribir Líneas de detalle incrementar y acumular */
DBMS_OUTPUT.PUT_LINE(RPAD(vr_emp.apellido,10) || ' * '
|| LPAD(TO_CHAR(vr_emp.salario,'999,999'),12));
cont_emple := cont_emple + 1;
sum_sal := sum_sal + vr_emp.salario;

END LOOP;
CLOSE c1;

/* Escribir totales informe */
DBMS_OUTPUT.PUT_LINE(' ***** NUMERO TOTAL EMPLEADOS: '
|| tot_emple || ' TOTAL SALARIOS: ' || tot_sal);

END listar_emple;
```



Actividades propuestas

- 4 Haz los cambios necesarios en el programa anterior para que realice el mismo listado usando una estructura de CURSOR FOR...LOOP (hay que tener en cuenta el ámbito de las variables de registro del cursor).

Hecho esto, podemos incluir en el programa rupturas por oficio, indicando en este caso únicamente el nombre del oficio y el número de empleados que tiene. Se entiende que se mantienen las rupturas por departamento y los subtotales; y dentro de cada departamento se harán, rupturas por oficio.

F. Atributos en cursos implícitos

Oracle abre implícitamente un cursor cuando procesa un comando SQL que no esté asociado a un cursor explícito. El cursor implícito se llama **SQL** y dispone también de los cuatro atributos mencionados, que pueden facilitarnos información sobre la ejecución de los comandos SELECT INTO, INSERT, UPDATE y DELETE.

El valor de los atributos del cursor SQL se refiere, en cada momento, a la última orden SQL:

- **SQL%NOTFOUND** dará TRUE si el último INSERT, UPDATE, DELETE o SELECT INTO ha fallado (no ha afectado a ninguna fila).



- **SQL%FOUND** dará TRUE si el último INSERT, UPDATE, DELETE o SELECT INTO ha afectado a una o más filas.
- **SQL%ROWCOUNT** devuelve el número de filas afectadas por el último INSERT, UPDATE, DELETE o SELECT INTO.
- **SQL%ISOPEN** siempre devolverá FALSO, ya que ORACLE cierra automáticamente el cursor después de cada orden SQL.

Estos atributos solamente están disponibles desde PL/SQL, no en órdenes SQL.

Es preciso hacer algunas observaciones respecto al uso de atributos en cursos implícitos pues su comportamiento difiere, en algunos casos, al de los cursos explícitos. De hecho, como acabamos de ver, el cursor estará cerrado inmediatamente después de procesar la orden SELECT...INTO, que es cuando se pregunta por su atributo o atributos (lo cual no determina un error como pasaba en los cursos explícitos).

A continuación, se especifican algunas peculiaridades en el comportamiento y uso de los atributos en cursos implícitos:

- Devolverán un valor relativo a la última orden INSERT, UPDATE, DELETE o SELECT...INTO, aunque el cursor esté cerrado.
- En el caso de que se trate de un SELECT...INTO, debemos tener en cuenta que ha de devolver una fila y sólo una, pues de lo contrario se producirá un error y se levantará automáticamente una excepción:
 - NO_DATA_FOUND, si la consulta no devuelve ninguna fila.
 - TOO_MANY_ROWS, si la consulta devuelve más de una fila.

Se detendrá la ejecución normal del programa y bifurcará a la sección EXCEPTION. Por tanto, cualquier comprobación de la situación del cursor en esas circunstancias resulta inútil.

- Lo indicado en el párrafo anterior no es aplicable a las órdenes INSERT, UPDATE, DELETE, ya que en estos casos no se levantan las excepciones correspondientes.

El siguiente ejemplo ilustra estas observaciones: se trata de un bloque que pretende cambiar la localidad de un departamento cuyo nombre en este caso es MARKETING (sabiendo que no existe ese departamento).

```

DECLARE
  v_dpto depart.dnombre%TYPE := 'MARKETING'; --(NO existe)
  v_loc  depart.loc%TYPE;
BEGIN
  UPDATE depart SET loc = 'SEVILLA'
    WHERE dnombre = v_dpto; /* no actualiza ninguna fila
                                pero no levanta excepción */
  IF SQL%NOTFOUND THEN
    DBMS_OUTPUT.PUT_LINE('Error en la actualización');
  END IF;

```



10. Cursos, excepciones y control de transacciones ...

10.2 Cursos

```
DBMS_OUTPUT.PUT_LINE('Continúa el programa');

SELECT loc INTO v_loc
  FROM depart
 WHERE dnombre = v_dpto; /* fallará y levantará
                           NO_DATA_FOUND */

IF SQL%NOTFOUND THEN
  DBMS_OUTPUT.PUT_LINE('Nunca pasará por aquí');
END IF;
END;
```

El resultado de la ejecución del programa será:

```
Error en la actualización
Continúa el programa
...
ERROR at line 1:
ORA-01403: no data found
ORA-06512: at line
```

Cuando un SELECT...INTO hace referencia a una función de grupo nunca se levantará la excepción NO_DATA_FOUND, y SQL%FOUND siempre será verdadero. Esto se debe a que las funciones de grupo siempre retornan algún valor (aunque sea NULL o cero).

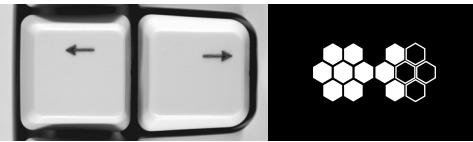
```
BEGIN
  ...
  SELECT MAX(salario) INTO v_max
    FROM emple
   WHERE dept_no = num_depart; -- > nunca levantará
                                 -- NO_DATA_FOUND
  IF SQL%NOTFOUND THEN
    ...
  END IF;
EXCEPTION
  WHEN NO_DATA_FOUND THEN      -- > nunca será cierto
    ...
  END;
```

Esta característica resulta útil para comprobar la existencia o no de una determinada fila sin que se levante NO_DATA_FOUND. Por ejemplo, para comprobar si existe un determinado departamento podemos escribir:

```
SELECT COUNT(*) INTO v_dummy
  FROM depart WHERE dept_no = v_depart;
  IF v_dummy > 0 THEN ...
```

G. Uso de cursos para actualizar filas

Los cursos se pueden usar para actualizar filas. En este caso, tenemos las siguientes opciones:



- **Cursor FOR UPDATE.** Son cursosres que permiten y facilitan la actualización (modificación o eliminación) de las filas seleccionadas por el cursor. Todas las filas seleccionadas serán bloqueadas tan pronto se abra el cursor (OPEN) y serán desbloqueadas al terminar las actualizaciones (al ejecutar COMMIT explícita o implícitamente).

Para crearlos únicamente habrá que añadirle FOR UPDATE al final de la declaración:

```
CURSOR <nombrecursor> IS <sentencia SELECT del cursor>
FOR UPDATE;
```

Los cursosres así declarados se usan exactamente igual que los cursosres explícitos estudiados (OPEN, FETCH, etcétera). Pero además de estas operaciones, permiten actualizar la última fila recuperada con FETCH mediante el comando (UPDATE o DELETE) incluyendo el especificador WHERE CURRENT OF *nombrecursor* en la cláusula para actualizar (UPDATE) o borrar (DELETE).

El formato para actualizar la fila seleccionada por un cursor FOR UPDATE es:

```
{UPDATE | DELETE} ... WHERE CURRENT OF <nombrecursor>
```

El siguiente procedimiento subirá el salario todos los empleados del departamento indicado en la llamada. La subida será el porcentaje indicado en la llamada:

```
CREATE OR REPLACE PROCEDURE subir_salario_dpto(
    vp_num_dpto NUMBER,
    vp_pct_subida NUMBER)
AS
CURSOR c_emple IS SELECT oficio, salario
    FROM emple WHERE dept_no = vp_num_dpto
    FOR UPDATE;
    vc_reg_emple c_emple%ROWTYPE;
    v_inc NUMBER(8,2);
BEGIN
    OPEN c_emple;
    FETCH c_emple INTO vc_reg_emple;
    WHILE c_emple%FOUND LOOP
        v_inc := (vc_reg_emple.salario / 100) *
            vp_pct_subida;
        UPDATE emple SET salario = salario + v_inc
            WHERE CURRENT OF c_emple; -- (al actual)
        FETCH c_emple INTO vc_reg_emple;
    END LOOP;
END subir_salario_dpto;
```

Si la consulta del cursor hace referencia a múltiples tablas, se deberá usar FOR UPDATE OF *nombrecolumna*, con lo que únicamente se bloquearán las filas correspondientes de la tabla que tenga la columna especificada.



10. Cursores, excepciones y control de transacciones ...

10.2 Cursores

```
CURSOR <nombrecursor> IS <sentencia SELECT del cursor>
FOR UPDATE OF <nombrecolumna>

DECLARE
  ...
CURSOR c_emple IS SELECT oficio, salario
  FROM emple, depart
  WHERE emple.dept_no = depart.dept_no
    FOR UPDATE OF salario;
  ...


```

La utilización de la cláusula FOR UPDATE, en ocasiones, puede ser problemática pues:

- Se bloquean todas las filas de la SELECT, no sólo la que se está actualizando en un momento dado.
- Si se ejecuta un COMMIT, después ya no se puede ejecutar FETCH. Es decir, tenemos que esperar a que estén todas las filas actualizadas para confirmar los cambios.
- **Uso de ROWID.** Consiste en usar el identificador de fila (ROWID) como condición de selección para actualizar filas. Para ello procederemos:
 - Al declarar el cursor en la cláusula SELECT, indicaremos que seleccione también el identificador de fila o ROWID:

```
CURSOR nombrecursor IS SELECT col1, col2, ... , ROWID
  FROM tabla;
```

- Al ejecutar el FETCH, se guardará el número de la fila en una variable o en un campo de la variable del cursor. Después ese número se usará en la cláusula WHERE de la actualización:

```
{UPDATE | DELETE} ... WHERE ROWID =
<variable_que_guarda_rowid>
```

En el ejemplo del epígrafe anterior:

```
CREATE OR REPLACE PROCEDURE subir_salario_dpto_b
  (vp_num_dpto NUMBER,
   vp_pct_subida NUMBER)
AS
  CURSOR c_emple IS SELECT oficio, salario, ROWID
    FROM emple WHERE dept_no = vp_num_dpto;
  vc_reg_emple c_emple%ROWTYPE;
  v_inc NUMBER(8,2);
BEGIN
  OPEN c_emple;
  FETCH c_emple INTO vc_reg_emple;
  WHILE c_emple%FOUND LOOP
    v_inc := (vc_reg_emple.salario /100) * vp_pct_subida;
    UPDATE emple SET salario = salario + v_inc
```



```

    WHERE ROWID = vc_reg_emple.ROWID;
    FETCH c_emple INTO vc_reg_emple;
END LOOP;
END subir_salario_dpto_b;

```

En caso de que tengamos que declarar explícitamente la variable que recogerá el ROWID debemos tener en cuenta que esta pseudocolumna tiene su propio tipo con el mismo nombre (ROWID), tal como estudiamos en la unidad anterior. Por tanto, declararemos la variable así:

```
<nombrededevariable> ROWID;
```

Además del uso de cursos FOR UPDATE y del ROWID, podemos seguir usando cualquier condición que permita la selección de las filas a actualizar.

Actividades propuestas



- 5 Escribe un programa que incremente el salario de los empleados de un determinado departamento que se pasará como primer parámetro. El incremento será una cantidad en euros que se pasará como segundo parámetro en la llamada. El programa deberá informar del número de filas afectadas por la actualización. Se actualizarán los salarios individualmente y usando el ROWID.

10.3 Excepciones

Las **excepciones** sirven para tratar errores en tiempo de ejecución, así como errores y situaciones definidas por el usuario.

Cuando se produce un error PL/SQL levanta una excepción y pasa el control a la sección EXCEPTION, donde buscará un manejador WHEN para la excepción o uno genérico (WHEN OTHERS) y dará por finalizada la ejecución del bloque actual.

El formato de la sección EXCEPTION es:

```

...
EXCEPTION
  WHEN <NombrededeExcepción1> THEN
    <instrucciones1>;
  WHEN <NombrededeExcepción2> THEN
    <instrucciones2>;
  ...
  [WHEN OTHERS THEN
    <instrucciones>]
END <nombre de programa>;

```

Para controlar posibles errores en otros lenguajes que no disponen de gestión de excepciones, se debe controlar después de cada orden cada una de las posibles condiciones de error.



10. Cursores, excepciones y control de transacciones ...

10.3 Excepciones

A continuación, estudiaremos los tres tipos de excepciones disponibles.

A. Excepciones internas predefinidas

Están predefinidas por Oracle. Se disparan automáticamente al producirse determinados errores. En la tabla adjunta se incluyen las excepciones más frecuentes con los códigos de error correspondientes:

Código error Oracle	Valor de SQL CODE	Excepción	Se disparan cuando...
ORA-06530	-6530	ACCESS_INTO_NULL	Se intenta acceder a los atributos de un objeto no inicializado.
ORA-06531	-6531	COLLECTION_IS_NULL	Se intenta acceder a elementos de una colección que no ha sido inicializada.
ORA-06511	-6511	CURSOR_ALREADY_OPEN	Intentamos abrir un cursor que ya se encuentra abierto.
ORA-00001	-1	DUP_VAL_ON_INDEX	Se intenta almacenar un valor que crearía duplicados en la clave primaria o en una columna con la restricción UNIQUE.
ORA-01001	-1001	INVALID_CURSOR	Se intenta realizar una operación no permitida sobre un cursor (por ejemplo, cerrar un cursor que no estaba abierto).
ORA-01722	-1722	INVALID_NUMBER	Fallo al intentar convertir una cadena a un valor numérico.
ORA-01017	-1017	LOGIN_DENIED	Se intenta conectar a ORACLE con un usuario o una clave no válidos.
ORA-01012	-1012	NOT_LOGGED_ON	Se intenta acceder a la base de datos sin estar conectado a Oracle.
ORA-01403	+100	NO_DATA_FOUND	Una sentencia SELECT ... INTO ... no devuelve ninguna fila.
ORA-06501	-6501	PROGRAM_ERROR	Hay un problema interno en la ejecución del programa.
ORA-06504	-6504	ROWTYPE_MISMATCH	La variable del cursor del HOST y la variable del cursor PL/SQL pertenecen a tipos incompatibles.
ORA-06533	-6533	SUBSCRIPT_OUTSIDE_LIMIT	Se intenta acceder a una tabla anidada o a un array con un valor de índice ilegal (por ejemplo, negativo).
ORA-06500	-6500	STORAGE_ERROR	El bloque PL/SQL se ejecuta fuera de memoria (o hay algún otro error de memoria).
ORA-00051	-51	TIMEOUT_ON_RESOURCE	Se excede el tiempo de espera para un recurso.
ORA-01422	-1422	TOO_MANY_ROWS	Una sentencia SELECT ... INTO ... devuelve más de una fila.
ORA-06502	-6502	VALUE_ERROR	Un error de tipo aritmético, de conversión, de truncamiento, etcétera.
ORA-01476	-1476	ZERO_DIVIDE	Se intenta la división entre cero.

Tabla 10.2. Excepciones más frecuentes en Oracle con los códigos de error.

No hay que declararlas en la sección DECLARE. Únicamente debemos incluir los manejadores WHEN con el tratamiento para cada excepción y/o un manejador genérico WHEN OTHERS que capturará cualquier otra excepción.

```

DECLARE
  ...
BEGIN
  ...
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    
```



```

BDMS_OUTPUT.PUT_LINE ('ERROR datos no encontrados');
WHEN TOO_MANY_ROWS THEN
  BDMS_OUTPUT.PUT_LINE ('ERROR demasiadas filas');
WHEN OTHERS THEN
  BDMS_OUTPUT.PUT_LINE ('ERROR');
END;

```

B. Excepciones definidas por el usuario

Las excepciones definidas por el usuario se usan para tratar condiciones de error definidas por el programador.

Para su utilización hay que seguir tres pasos:

1. Se declaran en la sección DECLARE de la forma siguiente:

```
<nombreexcepción> EXCEPTION;
```

2. Se disparan o levantan en la sección ejecutable del programa con la orden RAISE:

```
RAISE <nombreexcepción>;
```

3. Se tratan en la sección EXCEPTION según el formato ya conocido:

```
WHEN <nombreexcepción> THEN <tratamiento>;
```

```

DECLARE
  ...
  importe_erroneo EXCEPTION;
  ...
BEGIN
  ...
  IF precio NOT BETWEEN precio_min AND precio_maximo
THEN
  RAISE importe_erroneo;
END IF;
  ...
EXCEPTION
  ...
  WHEN importe_erroneo THEN
    DBMS_OUTPUT.PUT_LINE('Importe erróneo.
                           Venta cancelada.');
  ...
END;

```

También pueden levantarse excepciones en la sección EXCEPTION, pero no es habitual.

La instrucción RAISE se puede usar varias veces en el mismo bloque con la misma o con distintas excepciones, pero sólo puede haber un manejador WHEN para cada excepción.



10. Cursos, excepciones y control de transacciones ...

10.3 Excepciones

```
DECLARE
    ...
    venta_erronea EXCEPTION;
    ...
    importe_erroneo EXCEPTION;
    ...
BEGIN
    ...
    RAISE venta_erronea;
    ...
    RAISE importe_erroneo;
    ...
    RAISE venta_erronea;
    ...
EXCEPTION
    ...
    WHEN importe_erroneo THEN
        ...
    WHEN venta_erronea THEN
        ...
END;
```



Caso práctico

- 5 El siguiente ejemplo recibe un número de empleado y una cantidad que se incrementará al salario del empleado correspondiente. Utilizaremos dos excepciones, una definida por el usuario salario_nulo y la otra predefinida NO_DATA_FOUND.

```
CREATE OR REPLACE
PROCEDURE subir_salario(
    num_empleado INTEGER,
    incremento REAL)
IS
    salario_actual REAL;
    salario_nulo EXCEPTION;

BEGIN
    SELECT salario INTO salario_actual FROM emple
    WHERE emp_no = num_empleado;

    IF salario_actual IS NULL THEN
        RAISE salario_nulo; -- levanta salario_nulo
    END IF;

    UPDATE emple SET salario = salario + incremento
    WHERE emp_no = num_empleado;
```

(Continúa)



(Continuación)

```
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE(num_empleado||'*Err.No encontrado');
  WHEN salario_nulo THEN
    DBMS_OUTPUT.PUT_LINE(num_empleado||'*Err. Salario nulo');
END subir_salario;
```

C. Otras excepciones

Existen otros errores internos de Oracle que no tienen asignada una excepción. No obstante, generan (como cualquier otro error de Oracle) un código de error y un mensaje de error, a los que se accede mediante las funciones SQLCODE y SQLERRM.

Cuando se produce uno de estos errores también se transfiere el control a la sección EXCEPTION, donde se tratará el error en la cláusula WHEN OTHERS:

```
EXCEPTION
  ...
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('Error:' || SQLCODE || SQLERRM);
  ...
END;
```

En este ejemplo, se muestra al usuario el texto 'ERROR' con el *código de error* y el *mensaje de error* utilizando las funciones correspondientes.

Estas dos funciones, SQLCODE y SQLERRM, no son accesibles desde órdenes SQL*Plus, pero se pueden pasar a este entorno a través de soluciones como la siguiente:

```
...
WHEN OTHERS THEN
  :cod_err := SQLCODE;
  :msg_err := SQLERRM;
  ROLLBACK;
  EXIT;
END;
```

Podemos asociar una excepción de usuario a alguno de estos errores internos que no tienen excepciones predefinidas asociadas. Para ello procederemos así:

1. Definimos la excepción en la sección de declaraciones como si fuese una excepción definida por el usuario:

```
<nombreexcepción> EXCEPTION;
```



10. Cursos, excepciones y control de transacciones ...

10.3 Excepciones

En realidad, las excepciones predefinidas estudiadas anteriormente no son sino errores de Oracle asociados con excepciones (de manera similar a éstas) en el paquete STANDARD.

- Asociamos esa excepción a un determinado código de error mediante la directiva del compilador PRAGMA EXCEPTION_INIT, según el formato siguiente:

```
PRAGMA EXCEPTION_INIT(<nombre_excepción>, <número_de_error_Oracle>);
```

- Indicamos el tratamiento que recibirá la excepción en la sección EXCEPTION como si se tratase de cualquier otra excepción definida por el usuario o predefinida.

```
DECLARE
```

```
    ...
    err_externo EXCEPTION; -- Se define la excepción
    ...
    PRAGMA EXCEPTION_INIT(err_externo, -1547);/* Se asocia
                                                con el número de error de Oracle -1547 */
    ...
BEGIN
    ...
    /* no hay que levantar la excepción, pues cuando
       se produzca el error Oracle lo hará */
    ...
EXCEPTION
    ...
    WHEN err_externo THEN      -- Se trata como cualquier otra
        <tratamiento>;
END;
```



Caso práctico

- 6 El siguiente ejemplo ilustra lo estudiado hasta ahora respecto a la gestión de excepciones. Crearemos un bloque donde se define la excepción err_blancoas asociada con un error definido por el programador y la excepción no_hay_espacio asociándola con el error número -1547 de Oracle.

```
DECLARE
    cod_err number(6);
    vnif varchar2(10);
    vnom varchar2(15);
    err_blancoas EXCEPTION;
    no_hay_espacio EXCEPTION;
    PRAGMA EXCEPTION_INIT(no_hay_espacio, -1547);

BEGIN
    SELECT col1, col2 INTO vnif, vnom FROM TEMP2;
    IF SUBSTR(vnom,1,1) <= ' ' THEN
        RAISE err_blancoas;
```

(Continúa)



(Continuación)

```

END IF;
UPDATE clientes SET nombre = vnom WHERE nif = vnif;
EXCEPTION
    WHEN err_blanco THEN
        INSERT INTO temp2(col1) VALUES ('ERR blancos');
    WHEN no_hay_espacio THEN
        INSERT INTO temp2(col1) VALUES ('ERR tablespace');
    WHEN NO_DATA_FOUND THEN
        INSERT INTO temp2(col1) VALUES ('ERR no habia datos');
    WHEN TOO_MANY_ROWS THEN
        INSERT INTO temp2(col1) VALUES ('ERR demasiados datos');
    WHEN OTHERS THEN
        cod_err := SQLCODE;
        INSERT INTO temp2(col1) VALUES (cod_err);
END;

```

Cabe subrayar respecto a las excepciones que aparecen:

- *error_blanco* hay que declararla y levantarla.
- *no_hay_espacio* hay que declararla y asociarla con el error de Oracle, pero no hay que levantarla.
- *NO_DATA_FOUND* y *TOO_MANY_ROWS* no hay que declararlas ni asociarlas. Tampoco hay que levantarlas.
- El manejador *WHEN OTHERS* cazará cualquier otra excepción e insertará el código de error de Oracle en la tabla *temp2* que suponemos creada.

E. Propagación y ámbito de las excepciones

La gestión de excepciones en PL/SQL tiene unas reglas que se deben considerar en el diseño de aplicaciones:

- Cuando se levanta una excepción en la sección ejecutable, el programa bifurca a la sección EXCEPTION del bloque actual. Si no está definida en ella, la excepción se propaga a la sección EXCEPTION del bloque que llamó al actual; así, hasta encontrar tratamiento para la excepción o devolver el control al programa Host.
- Una vez tratada la excepción en un bloque, se devuelve el control al bloque que llamó al que trató la excepción, con independencia del que la disparó.
- Si, después de tratar una excepción, queremos volver a la línea siguiente a la que se produjo, no podemos hacerlo directamente, pero sí es posible diseñar el programa para que funcione así. Esto se consigue encerrando el comando o comandos que pueden levantar la excepción en un bloque junto el tratamiento para la excepción:

```

SELECT INTO ... -- > Puede levantar NO_DATA_FOUND
...

```

Si queremos que dos o más excepciones ejecuten la misma secuencia de instrucciones, podremos indicarlo en la cláusula WHEN indicando las excepciones unidas por el operador OR:

```
WHEN exc1 OR exc2 OR exc3... THEN...
```

No obstante, en la lista no podrá aparecer WHEN OTHERS.



10. Cursos, excepciones y control de transacciones ...

10.3 Excepciones

Para que el control del programa no salga del bloque actual cuando se produzca una excepción, podemos encerrar el comando que puede levantar la excepción en un bloque y tratar la excepción en ese bloque:

```
...
BEGIN
    SELECT INTO ...          -- > Puede levantar NO_DATA_FOUND
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        ...;                -- > tratamiento para la excepción
END;
...
```

- La cláusula WHEN OTHERS tratará cualquier excepción que no aparezca en las cláusulas WHEN anteriores, con independencia del tipo de excepción. De este modo, se evita que la excepción se propague a los bloques de nivel superior.

En el siguiente ejemplo, si se levanta la excepción *ex1*, se tratará en el mismo bloque, pero el control pasará después al bloque <<exterior>> en la línea siguiente a la llamada. Si se hubiese levantado NO_DATA_FOUND, al no encontrar tratamiento, la excepción se propaga al bloque <<exterior>>, donde será tratada por WHEN OTHERS (suponiendo que no exista un tratamiento específico), devolviendo el control al bloque o la herramienta que llamó a <<exterior>>:

```
<<exterior>>
DECLARE
    ...
BEGIN
    ...
<<interior>>
DECLARE
    ...
    ex1 EXCEPTION;
BEGIN
    ...
    RAISE ex1;
    ...
    SELECT coll1, col2 INTO ...; --> Levanta
NO_DATA_FOUND
    ...
EXCEPTION
    ...
    WHEN ex1 THEN   --> Tratamiento para ex1
        ROLLBACK; --> Después pasará el control
a (1)
    ...
END;

/*(1)*/... ;
...
EXCEPTION
    ...
    WHEN OTHERS THEN...
END;
```



- Si la excepción se levanta en la sección declarativa (por un fallo al inicializar una variable, por ejemplo), automáticamente se propagará al bloque que llamó al actual, sin comprobar si existe tratamiento para la excepción en el mismo bloque que se levantó.
- También se puede levantar una excepción en la sección EXCEPTION de forma voluntaria o por un error que se produzca al tratar una excepción. En este caso, también se propagará de forma automática al bloque que llamó al actual, sin comprobar si existe tratamiento para la excepción en el mismo bloque que se levantó. La excepción original (la que se estaba tratando cuando se produjo la nueva excepción) se perderá, ya que solamente puede estar activa una excepción.

EXCEPTION

```
WHEN INVALID_NUMBER THEN
    INSERT INTO ... -- podría levantar DUP_VAL_ON_INDEX
WHEN DUP_VAL_ON_INDEX THEN -- no atrapará la exception
```

- En ocasiones, puede resultar conveniente volver a levantar la misma excepción después de tratarla para que se propague al bloque de nivel superior. Esto puede hacerse indicando al final de los comandos de manejo de la excepción el comando **RAISE** sin parámetros:

```
...
WHEN TOO_MANY_ROWS THEN
    ...;      -- intrucciones de manejo del error
    RAISE;   -- levanta de nuevo la excepción y la pro-
    paga
...
```

- Las excepciones declaradas en un bloque son locales al bloque, no son conocidas en bloques de nivel superior. No se puede declarar dos veces la misma excepción en el mismo bloque, pero sí en distintos bloques. En este caso, la excepción del subbloque prevalecerá sobre la del bloque, aunque esta última se puede levantar desde el subbloque utilizando la notación de punto (**RAISE nombrebloque.nombreexcepción**).
- Las variables locales, las globales, los atributos de un cursor, y otros objetos del programa se pueden referenciar desde la sección EXCEPTION según las reglas de ámbito que rigen para los objetos del bloque. Pero si la excepción se ha disparado dentro de un bucle cursor FOR...LOOP, no se podrá acceder a los atributos del cursor, ya que Oracle cierra este cursor antes de disparar la excepción.
- En la sección EXCEPTION no se puede usar < GOTO etiqueta > para salir de esta sección o entrar en otra cláusula WHERE.
- Cuando no se encuentra tratamiento para una excepción en ninguno de los bloques o programas, Oracle da por finalizado con fallos el programa y ejecutará automáticamente un ROLLBACK deshaciendo todos los cambios pendientes de confirmación realizados por el programa.

Si la excepción se levanta en la sección declarativa o en la sección EXCEPTION, automáticamente se propagará al bloque que llamó al actual.

Algunos tipos de excepciones se han de tratar con mucha precaución, ya que se puede caer en un bucle infinito (por ejemplo, NOT_LOGGED_ON).

Si una excepción de usuario no encuentra tratamiento en el bloque en el que ha sido declarada se propagará a bloques de nivel superior; pero éstos sólo pueden cazarlas mediante WHEN OTHERS pues la excepción es desconocida fuera de su ámbito.



F. Utilización de RAISE_APPLICATION_ERROR

RAISE_APPLICATION_ERROR tiene un tercer parámetro opcional cuyos detalles de uso quedan fuera de nuestros objetivos.

En el paquete DBMS_STANDARD se incluye un procedimiento muy útil llamado RAISE_APPLICATION_ERROR que sirve para levantar errores y definir y enviar mensajes de error. Su formato es el siguiente:

RAISE_APPLICATION_ERROR(*número_de_error*, *mensaje_de_error*);

Donde *número_de_error* es un número comprendido entre -20000 y -20999, y *mensaje_de_error* es una cadena de hasta 512 bytes.

Cuando un subprograma hace esta llamada, se levanta la excepción y produce la salida del programa. Esta excepción solamente puede ser manejada con WHEN OTHERS.



Caso práctico

- 7 El siguiente ejemplo muestra el funcionamiento de RAISE_APPLICATION_ERROR en un procedimiento de funcionalidad similar al estudiado en el caso práctico 7 (*subir_sueldo*).

```
CREATE OR REPLACE
PROCEDURE subir_sueldo
  (Num_emple NUMBER, incremento NUMBER)
IS
  salario_actual NUMBER;
BEGIN
  SELECT salario INTO salario_actual FROM empleados
    WHERE emp_no = num_emple;
  IF salario_actual IS NULL THEN
    RAISE_APPLICATION_ERROR(-20010, 'Salario Nulo');
  ELSE
    UPDATE empleados SET sueldo = salario_actual +
      incremento WHERE emp_no = num_emple;
  ENDIF;
END subir_sueldo;
```



Actividades propuestas



6 Escribe un procedimiento que reciba todos los datos de un nuevo empleado y procese la transacción de alta, gestionando posibles errores. El procedimiento deberá gestionar en concreto los siguientes puntos:

- no_existe_departamento.
- no_existe_director.
- numero_empleado_duplicado.
- Salario nulo: con RAISE_APPLICATION_ERROR.
- Otros posibles errores de Oracle visualizando código de error y el mensaje de error.

10.4 Control de transacciones

Una **transacción** es un conjunto de operaciones dependientes unas de otras que se realizan en la base de datos. Para que la transacción se ejecute han de realizarse todas y cada una de las partes u operaciones que la componen; en el caso de que alguna falle se dará por fallida toda la transacción.

Por ejemplo, en el caso de una transferencia bancaria de una cuenta a otra, la transacción completa ha de contemplar el cargo en una de las cuentas y el abono en la otra. Si por alguna circunstancia una de las dos operaciones (que forman parte de la transacción) no puede llevarse a cabo tampoco debería realizarse la otra.

Una transacción puede incluir una o, frecuentemente, varias instrucciones de manipulación de datos. Será el usuario (el programador, en este caso) quien decide cuántas y cuáles serán las operaciones que compondrán la transacción para garantizar la consistencia de la información almacenada.

En el entorno Oracle la transacción:

- **Comienza** con la primera orden SQL de la sesión del usuario o con la primera orden SQL posterior a la finalización de la transacción anterior.
- **Finaliza** cuando se ejecuta un comando de control de transacciones (COMMIT o ROLLBACK), una orden de definición de datos (DDL) o cuando finaliza la sesión.

Una vez completada la transacción ya no puede deshacerse.



10. Cursos, excepciones y control de transacciones ...

10.4 Control de transacciones

Oracle garantiza la consistencia de los datos en una transacción en términos de VALE TODO o NO VALE NADA, es decir, o se ejecutan todas las operaciones que componen la transacción o no se ejecuta ninguna. Así pues, la base de datos tiene un estado antes de la transacción y un estado después de la transacción, pero no hay estados intermedios.

En el ejemplo de al lado se garantiza que la transferencia se llevará a cabo totalmente o que no se realizará ninguna operación, pero en ningún caso se quedará a medias.

```
BEGIN
    COMMIT; /* marca el final de la transacción anterior
               y el comienzo de la actual */
    ...
    UPDATE cuentas SET saldo = saldo - v_importe_tranfer
        WHERE num_cta = v_cta_origen;
    UPDATE cuentas SET saldo = saldo + v_importe_tranfer
        WHERE num_cta = v_cta_destino;
    ...
    COMMIT; /* confirma y da por finalizada la
               Transacción actual */
    ...
EXCEPTION
    WHEN OTHERS THEN
        ROLLBACK;
END;
```

A. Comandos de control de transacciones

Oracle establece un punto de salvaguarda implícito cada vez que se ejecuta una sentencia de manipulación de datos. En el caso de que la sentencia falle, Oracle restaurará automáticamente los datos a sus valores iniciales.

Oracle dispone de los siguientes comandos de control de transacciones:

- **COMMIT.** Da por concluida la transacción actual y hace definitivos los cambios efectuados, liberando las filas bloqueadas. Sólo después de que se ejecute el COMMIT los demás usuarios tendrán acceso a los datos modificados.
- **ROLLBACK.** Da por concluida la transacción actual y deshace los cambios que se pudiesen haber producido en la misma, liberando las filas bloqueadas. Se utiliza especialmente cuando no se puede concluir una transacción porque se levanta una excepción.
- **ROLLBACK implícitos.** Cuando un subprograma almacenado falla y no se controla la excepción que produjo el fallo, Oracle automáticamente ejecuta ROLLBACK sobre todo lo realizado por el subprograma, salvo que en el subprograma hubiese algún COMMIT, en cuyo caso lo confirmado no sería deshecho.
- **SAVEPOINT <punto_de_salvaguarda>.** Se utiliza en conjunción con ROLLBACK TO para poner marcas o puntos de salvaguarda al procesar transacciones. Esto permite deshacer parte de una transacción.
- **ROLLBACK TO <punto_de_salvaguarda>.** Deshace el trabajo realizado sobre la base de datos después del punto indicado, incluyendo posibles bloqueos. No obstante, tampoco se confirma el trabajo hecho hasta el *punto_de_salvaguarda*. La transacción no finaliza hasta que se ejecuta un comando de control de transacciones COMMIT o ROLLBACK, o hasta que finaliza la sesión (o se ejecuta una orden de definición de datos DDL).



```

CREATE OR REPLACE PROCEDURE prueba_savepoint (
    numfilas POSITIVE)
AS
BEGIN
    SAVEPOINT NINGUNA;
    INSERT INTO temp1 (col1) VALUES ('PRIMERA FILA');
    SAVEPOINT UNA;
    INSERT INTO temp1 (col1) VALUES ('SEGUNDA FILA');
    SAVEPOINT DOS;
    IF numfilas = 1 THEN
        ROLLBACK TO UNA;
    ELSIF numfilas = 2 THEN
        ROLLBACK TO DOS;
    ELSE
        ROLLBACK TO NINGUNA;
    END IF;
    COMMIT;
EXCEPTION
    WHEN OTHERS THEN
        ROLLBACK;
END;

```

Los nombres de las marcas son identificadores no declarados y se pueden reutilizar.

Actividades propuestas



- 7 Crea el programa anterior y la tabla de pruebas TEMP (Col1 VARCHAR2 (40)). Ejecuta el programa introduciendo diversos valores (0, 1, 2, 3,...), observa y razona su efecto en la tabla.

El ámbito de los puntos de salvaguarda es el definido por la transacción desde que comienza hasta que termina, por tanto, trasciende de las reglas de ámbito y visibilidad de otros identificadores.

Cuando se ejecuta un ROLLBACK TO <marca>, todas las marcas después del punto indicado desaparecen (la indicada no desaparece). También desaparecen todas las marcas cuando se ejecuta un COMMIT.

B. Transacciones autónomas

Son aquellas que pueden confirmarse o rechazarse con independencia de lo que pueda ocurrir con la otra transacción en curso. Y viceversa, esto es, que lo que ocurra con la transacción en curso no afecte a la transacción autónoma.

Este tipo de transacciones se usan en pequeños programas de un único bloque que realizan tareas auxiliares muy puntuales (por ejemplo, control de acceso de usuarios) que son llamados por distintos programas en diferentes contextos.

Al tratarse de una transacción autónoma, el programa puede ser llamado desde cualquier otro y realizará las acciones previstas sin afectar ni resultar afectado por las transacciones que estén en curso.



10. Cursos, excepciones y control de transacciones ...

10.4 Control de transacciones

Para crear una transacción autónoma crearemos un bloque o programa que realice las acciones previstas e incluiremos en la sección declarativa la directiva o pragma **AUTONOMOUS_TRANSACTION**. Tal como se muestra a continuación:

```
CREATE OR REPLACE
PROCEDURE control_acceso (
    usr VARCHAR2,
    obs VARCHAR2 )
IS
    PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
    INSERT INTO control_accesos
        VALUES (usr, SYSTIMESTAMP, obs);
    COMMIT;
END;
```

C. Transacciones de sólo lectura

Las transacciones de sólo lectura se usan para garantizar la consistencia de los datos recuperados entre distintas consultas frente a posibles cambios que puedan ocurrir entre ellas.

El comienzo de una transacción de sólo lectura se establece mediante la instrucción **SET TRANSACTION READ ONLY**. Todas las consultas que se ejecutan a continuación solamente verán aquellos cambios confirmados antes del comienzo de la transacción: es como si se hiciese una fotografía de la base de datos.

Antes de usar **SET TRANSACTION READ ONLY** debemos confirmar o rechazar la transacción en curso estableciendo así el comienzo de la nueva transacción. Una vez efectuadas todas las operaciones de consulta cuya consistencia queremos garantizar, introduciremos **COMMIT** para dar por finalizada la transacción de sólo lectura.

```
DECLARE
    ventas_dia REAL;
    ventas_semana REAL;
BEGIN
    COMMIT; -- Confirma transacción anterior e inicia la nueva
    SET TRANSACTION READ ONLY; /* indica que la transacción
                                   será de sólo lectura */
    SELECT count(*) INTO ventas_dia FROM ventas
        WHERE fecha = SYSDATE;
    SELECT count(*) INTO ventas_semana FROM ventas
        WHERE fecha > SYSDATE - 7;
    COMMIT; -- finaliza la transacción actual de lectura
END;
```



Conceptos básicos



- Los cursos explícitos se usan en consultas que pueden devolver más de una fila (o ninguna).

- Los formatos de las principales operaciones que se realizan con un cursor explícito son:

Declarar: CURSOR <nombrecursor> IS
SELECT <sentencia select>;

Abrir: OPEN <nombrecursor>;

Cerrar: CLOSE <nombrecursor>;

Recuperar fila: FETCH <nombrecursor> INTO
{<variable>}|<listavariables>};

- Los atributos del cursor sirven para conocer información de su estado. Los principales son: %FOUND, %NOTFOUND, %ISOPEN, %ROWCOUNT

- El cursor **FOR LOOP** incluye de manera implícita las acciones de abrir, declarar variable para recuperar los datos, recorrer y recuperar las filas y cerrar el cursor. Su formato genérico es:

```
FOR <nombrevareg> IN <nombrecursor>
LOOP <instrucciones;> END LOOP;
```

- En instrucciones con **cursos implícitos** podemos preguntar por los atributos del cursor aunque estará cerrado.

- Para actualizar una tabla en PL/SQL podemos optar por usar en la cláusula WHERE:

- CURRENT OF para indicar que la fila a actualizar es la actual en cursos declarados FOR UPDATE.
- El ROWID previamente guardado, o cualquier expresión que determine la/s fila/s a actualizar.

- Cuando se levanta una excepción el control pasa a la sección EXCEPTION donde buscará un manejador y dará por finalizada la ejecución del bloque actual. Si no encuentra tratamiento, se propagará hasta encontrarlo o

retornará al entorno que lanzó la aplicación dando la ejecución de la aplicación por fallida. Una vez tratada la excepción el control retorna a la línea siguiente a la que llamó al programa que trató la excepción.

- WHEN OTHERS se usa para tratar cualquier excepción que no aparezca en las cláusulas WHEN anteriores.
- Las excepciones definidas por el usuario deben ser declaradas y levantadas explícitamente según los formatos estudiados.

- Una transacción es un conjunto de operaciones dependientes unas de otras. Para que la transacción se complete han de realizarse todas las operaciones que la componen; en el caso de que alguna falle se dará por fallida toda la transacción.

- La transacción comienza con la primera orden SQL de la sesión o con la primera orden SQL posterior a la finalización de la transacción anterior y finaliza cuando se ejecuta un comando de control de transacciones (COMMIT o ROLLBACK), una orden de definición de datos (DDL) o cuando finaliza la sesión.
- Cuando un subprograma almacenado falla y no se controla la excepción que produjo el fallo, Oracle automáticamente ejecuta ROLLBACK sobre todo lo realizado por el subprograma, salvo que en el subprograma hubiese algún COMMIT, en cuyo caso lo confirmado no sería deshecho.
- Para garantizar la consistencia de los datos recuperados entre distintas consultas usaremos SET TRANSACTION READ ONLY antes de ejecutar las instrucciones SELECT; y una vez ejecutadas deberemos incluir COMMIT para liberar la transacción de sólo lectura.
- Para crear una transacción autónoma crearemos un bloque o programa que realice las acciones previstas e incluiremos en la sección declarativa la directiva o pragma AUTONOMOUS_TRANSACTION. Debemos confirmar o rechazar la transacción antes de salir del ámbito del programa.



Actividades complementarias

1 Desarrolla un procedimiento que visualice el apellido y la fecha de alta de todos los empleados ordenados por apellido.

2 Codifica un procedimiento que muestre el nombre de cada departamento y el número de empleados que tiene.

3 Escribe un programa que visualice el apellido y el salario de los cinco empleados que tienen el salario más alto.

4 Codifica un programa que visualice los dos empleados que ganan menos de cada oficio.

5 Desarrolla un procedimiento que permita insertar nuevos departamentos según las siguientes especificaciones:

- Se pasará al procedimiento el nombre del departamento y la localidad.
- El procedimiento insertará la fila nueva asignando como número de departamento la decena siguiente al número mayor de la tabla.
- Se incluirá la gestión de posibles errores.

6 Codifica un procedimiento que reciba como parámetros un número de departamento, un importe y un porcentaje; y que suba el salario a todos los empleados del departamento indicado en la llamada. La subida será el porcentaje o el importe que se indica en la llamada (el que sea más beneficioso para el empleado en cada caso).

7 Escribe un procedimiento que suba el sueldo de todos los empleados que ganen menos que el salario medio de su oficio. La subida será del 50 por 100 de la dife-

rencia entre el salario del empleado y la media de su oficio. Se deberá hacer que la transacción no se quede a medias, y se gestionarán los posibles errores.

8 Diseña una aplicación que simule un listado de liquidación de los empleados según las siguientes especificaciones:

- El listado tendrá el siguiente formato para cada empleado:

```
*****  
Liquidación del empleado : (1)  
Dpto : (2)  
Oficio : (3)  
Salario : (4)  
Trienios : (5)  
Comp. responsabilidad : (6)  
Comisión : (7)  
*****  
Total : (8)  
*****
```

Donde:

- 1, 2, 3 y 4 corresponden a apellido, departamento, oficio y salario del empleado.
- 5 es el importe en concepto de trienios. Un trienio son tres años completos, desde la fecha de alta hasta la de emisión, y supone 50 €.
- 6 es el complemento por responsabilidad. Será de 100 € por cada empleado que se encuentre directamente a cargo del empleado en cuestión.
- 7 es la comisión. Los valores nulos serán sustituidos por ceros.
- 8 es la suma de todos los conceptos anteriores.

El listado irá ordenado por Apellido.



- 9** Crea la tabla T_liquidacion con las columnas apellido, departamento, oficio, salario, trienios, comp_responsabilidad, comisión y total; y modifica la aplicación anterior para que, en lugar de realizar el listado directamente en pantalla, guarde los datos en la tabla. Se controlarán todas las posibles incidencias que puedan ocurrir durante el proceso.
- 10** Escribe un programa para introducir nuevos pedidos según las siguientes especificaciones:

- Recibirá como parámetros PEDIDO_NO, PRODUCTO_NO, CLIENTE_NO, UNIDADES y la FECHA_PEDIDO (opcio-

nal, por defecto la del sistema). Verificará todos estos datos así como las unidades disponibles del producto y el límite de crédito del cliente y fallará enviando un mensaje de error en caso de que alguno sea erróneo.

- Insertará el pedido y actualizará la columna DEBE de clientes incrementándola el valor del pedido (UNIDADES * PRECIO_ACTUAL). También actualizará las unidades disponibles del producto e incrementará la comisión para el empleado correspondiente al cliente en un 5% del valor total del pedido. Todas estas operaciones se realizarán como una única transacción.