ÍNDICE

Contenido

Antolín·Muñoz·Chaparro¶
Jefe·de·Proyecto·Sistemas·Informáticos·¶
División·III·de·Aplicaciones·de·Costes·de·Personal·y·Pensiones·Públicas¶
Oficina·de·Informática·Presupuestaria·(DIR3:·EA0027952)¶
Intervención·General·de·la·Administración·del·Estado¶

Antolín·Muñoz·Chaparro¶
Jefe·de·Proyecto·Sistemas·Informáticos·¶
División·III·de·Aplicaciones·de·Costes·de·Personal·y·Pensiones·Públicas¶
Oficina·de·Informática·Presupuestaria·(DIR3:·EA0027952)¶
Intervención·General·de·la·Administración·del·Estado¶

¿QUÉ ES UN CURSOR?

Para poder procesar una instrucción del lenguaje SQL, Oracle asigna un área de memoria que recibe el nombre de área de contexto.

Esta área contiene información necesaria para completar el procesamiento, incluyendo el número de filas procesadas para la instrucción, un puntero a la versión analizada de la instrucción, y en el caso de las consultas, el conjunto activo, que es el conjunto de filas resultado de la consulta.

Un cursor es un puntero al área de contexto. Mediante los cursores un programa PL/SQL puede controlar el conjunto de valores que ha devuelto una instrucción SELECT.

Existen 3 tipos de cursores:

- Cursores explícitos (CURSOR).
- Cursores implícitos. (SQL)
- Cursores dinámicos.

CURSORES EXPLÍCITOS

Un cursor explícito es aquel al que se le asigna explícitamente a una instrucción SELECT con un nombre, mediante la orden CURSOR..IS.

Todo cursor explicito necesita de 4 pasos para su procesamiento correcto:

1. Declaración del cursor (`CURSOR ... IS`).
2. Apertura del cursor (`OPEN ...`)
3. Recogida de los resultado en variables PL/SQL: (`FETCH...`)
4. Cierre del cursor. (`CLOSE....`) .

Antolín·Muñoz·Chaparro¶
Jefe·de·Proyecto·Sistemas·Informáticos·¶
División·III·de·Aplicaciones·de·Costes·de·Personal·y·Pensiones·Públicas¶
Oficina·de·Informática·Presupuestaria·(DIR3:·EA0027952)¶
Intervención·General·de·la·Administración·del·Estado¶
GOBIERNO DE ESPAÑA MINISTERIO DE HACIENDA

Declaración de un cursor

La sintaxis es la siguiente:

```
CURSOR <nombre_cursor> IS <operación de SELECT>;
```

EJEMPLO

```
DECLARE
      CURSOR c_estudiantes IS
            SELECT cod_matricula, nombre FROM estudiantes;
```

Apertura de un cursor

La sintaxis es la siguiente:

```
OPEN <nombre_cursor>;
```

La sintaxis apertura de un cursor implica las siguientes acciones:

- Se examinan los valores de las variables acopladas, en caso de haberlas.
- Se determina el conjunto activo de valores.
- Se hace apuntar el puntero del conjunto a la primera fila.

Las variables acopladas se eximan en el momento de abrir el cursor y sólo en ese momento.

EJEMPLO

```
DECLARE
      V_salaID          clase.sala_id%TYPE;
      V_edificio        salas.edificio%TYPE;
      V_departamento    clase.departamento%TYPE;
      V_curso           clase.curso%TYPE;
      CURSOR c_edificios IS
            SELECT edificio
            FROM salas, clase
            WHERE salas.sala_id = clase.sala_id
            and departamento = v_departamento
            and curso = v_curso;
BEGIN
      -- Asignar valores a las variables de acoplamiento.
      V_departamento := 'HIS';
      V_curso := 101;

      -- Abrimos el cursor.
      OPEN c_edificios;

      /* Reasignamos los valores de las variables de
acoplamiento, lo que no tiene ningún efecto, ya que el
cursor ya está abierto. */
      v_departamento := 'XXX';
      v_curso := -1;
END;
```

Extracción de los datos del cursor

Como se comentó en el capítulo anterior, la instrucción SELECT no permite devolver los valores a pantalla una vez extraídos, así que el resultado de ejecutar el cursor, tendrá que ser devuelto en algún sitio. Para eso se utiliza la instrucción FETCH que permite el envío de los datos de una fila del cursor a variables o a un registro.

Sintaxis:

FETCH <nombre del cursor> INTO <lista_variables | tipo registro>;

Por supuesto, para poder utilizar un FETCH previamente ha de estar abierto el cursor.

EJEMPLO

```
DECLARE
      V_matricula      estudiantes.cod_matricula%TYPE;
      V_nombre         estudiantes.nombre%TYPE;
      CURSOR c_estudiante IS
            SELECT matricula, nombre FROM estudiantes;
BEGIN
      OPEN c_estudiante;
      FETCH c_estudiante INTO v_matricula, v_nombre;
END;
```

Ó

```
DECLARE
      TYPE estudiante IS RECORD (
            matricula        estudiantes.cod_matricula%TYPE,
            nombre           estudiantes.nombre%TYPE
            );
      v_estudiante           estudiante;
      CURSOR c_estudiante IS
            SELECT matricula, nombre FROM estudiantes;

BEGIN
      OPEN c_estudiante;
      FETCH c_estudiante INTO v_estudiante;
END;
```

Cierre de un cursor

Sintaxis:

```
CLOSE <nombre_cursor>;
```

Atributos de los cursores

A un cursor se le pueden aplicar los siguientes atributos:

- %FOUND
- %NOTFOUND
- %ISOPEN
- %ROWCOUNT

%FOUND

Devuelve TRUE si el último FETCH ejecutado sobre un cursor recuperó una fila.

%NOTFOUND

Devuelve TRUE si el último FETCH ejecutado no recuperó ninguna fila, como por ejemplo cuando se llega al final del cursor.

%ISOPEN

Devuelve TRUE cuando el cursor asociado está abierto.

%ROWCOUNT

Devuelve el número de filas extraídas por el cursor hasta el momento.

EJEMPLO

```
DECLARE
      CURSOR c_estudiante IS
            SELECT * FROM estudiantes
            WHERE cod_matricula > 500;
      V_matricula     estudiantes%ROWTYPE;
BEGIN
      OPEN c_estudiante;
      LOOP
            FETCH c_estudiante INTO v_estudiante;
            EXIT WHEN c_estudiante%NOTFOUND;
            ……
      END LOOP;
      CLOSE c_estudiante;
END;
```

Cursores parametrizados

Se utilizan cuando la instrucción SELECT hace referencia en el WHERE a valores que se asignan mediante parámetros en el propio cursor.

EJEMPLO

```
DECLARE
      Num_matricula        estudiantes.cod_matricula%TYPE;
      CURSOR c_estudiante IS
            SELECT * FROM estudiantes
            WHERE cod_matricula > num_matricula;
      v_estudiante         c_estudiante%ROWTYPE;
BEGIN
      OPEN c_estudiante;
END;
```

Esta definición se puede sustituir por:

```
DECLARE
      CURSOR c_estudiante(num_matricula
                          estudiantes.cod_matricula%TYPE)
      IS
            SELECT * FROM estudiantes
            WHERE cod_matricula > num_matricula;
            v_estudiante          c_estudiante%ROWTYPE;
BEGIN
      OPEN c_estudiante(1000);
END;
```

CURSORES IMPLICITOS

Todo cursor explícito se realiza sobre un contexto de selección de registros previos. Un cursor implícito se puede utilizar sobre otras operaciones como por ejemplo UPDATE, DELETE o INSERT.

En estos cursores ya no se aplica el concepto de apertura, recorrido y cierre del cursor, ni el de nominación del mismo. Todos los cursores de este tipo se invocan con la palabra clave SQL.

EJEMPLO

```
BEGIN
      UPDATE estudiantes
      SET nota = 9
      WHERE
            NOMBRE = 'Juan Rodriguez';

      IF SQL%NOTFOUND THEN
            INSERT INTO estudiantes
            (cod_matricula, nombre, nota)
            VALUES (1000,'Juan Rodriguez',9);
      END IF;
      -- Si no existe en el update, una fila para
      -- actualizar ejecuta el INSERT.
END;
```

CURSORES SQL DINÁMICO

Con SQL dinámico también podemos utilizar cursores.

Para utilizar un cursor dinámico sólo debemos construir nuestra sentencia SELECT en una variable de tipo carácter y ejecutarla con EXECUTE IMMEDIATE, utilizando la palabra clave INTO, como se ha explicado en el capítulo 5.

EJEMPLO DE SQL DINÁMICO CON UNA INSTRUCCIÓN SELECT

```
DECLARE
  TYPE tipocursor     IS REF CURSOR;
  v_cursor            tipocursor;
  v_registro          EMPLEADO%ROWTYPE;
  Cadena_sql          VARCHAR2(1000);
  V_Conteo            NUMBER;
BEGIN
     Cadena_sql := 'SELECT COUNT(*) FROM EMPLEADO';
     EXECUTE IMMEDIATE Cadena_sql INTO V_conteo;
     Cadena_sql := 'SELECT * FROM EMPLEADO';
     OPEN v_cursor FOR Cadena_sql;
     LOOP
          FETCH v_cursor INTO v_registro;
          DBMS_OUTPUT.PUT_LINE(v_registro.NIF||'         -
'||v_registro.NOMBRE);
          EXIT WHEN v_cursor%NOTFOUND;
     END LOOP;
END;
```

SUBPROGRAMAS

Los subprogramas, como se explicó en el primer capítulo, son bloques nominados que se almacenan en la base de datos.

Los bloques que se habían visto hasta ahora eran bloque anónimos que se compilaban cada vez que se ejecutaban y que no se almacenaban en la base de datos.

Los bloques nominados que pueden ser de 4 tipos:

- Procedimientos
- Funciones
- Paquetes
- Disparadores

A los dos primeros: procedimientos y funciones, son los que se les conoce como subprogramas.

PROCEDIMIENTOS

Un procedimiento se compila una vez y se almacena en la base de datos.

Pueden ser invocados desde otro bloque PL/SQL tanto nomina como sin nominar.

Admite en su definición, el paso de parámetros tanto de entrada, salida, como entrada/salida.

Creación de un procedimiento.

La sintaxis es la siguiente:

```
CREATE [OR REPLACE] PROCEDURE nombre_proced
[(argumento1 [{IN | OUT | IN OUT}] tipo,
```

```
. . . . . .
argumentoX [{IN | OUT | IN OUT}] tipo)] {IS | AS}
cuerpo del procedimiento;
```

Tipos de parámetros en subprogramas

Los parámetros pueden ser de tres tipos:

- IN
- OUT
- IN OUT

La sintaxis para la definición de un parámetro dentro de un subprograma es la siguiente:

```
parametro [{IN | OUT | IN OUT}] tipo
```

Como se puede observar en la sintaxis, la indicación del tipo de parámetro (IN, OUT o IN OUT) es opcional. Si no se indica nada, se asume que el parámetro es de tipo IN (de entrada).

IN

Indica que es un parámetro de entrada.

Cuando se invoque al subprograma que lleve definido un parámetro de este tipo, habrá que pasarle el valor correspondiente. Al terminar la ejecución del subprograma, el parámetro definido con este tipo mantendrá el mismo valor que antes de su ejecución.

Por tanto un parámetro de tipo IN, es de sólo lectura.

OUT

Indica que es un parámetro de salida.

Cuando se invoque al subprograma que lleve definido un parámetro de este tipo, habrá que pasarle una variable definida en el objeto llamante. Al terminar la ejecución

del subprograma, el parámetro definido con este tipo devolverá sobre la variable de llamada, el valor actualizado.

Por tanto un parámetro de tipo OUT, es de sólo escritura.

IN OUT

Indica que es un parámetro de entrada y salida.

Cuando se invoque al subprograma que lleve definido un parámetro de este tipo, habrá que pasarle una variable definida en el objeto llamante. Al comenzar la ejecución del subprograma, utilizará como valor de comienzo el que se le ha pasado al subprograma a través de la variable de llamada, y al terminar la ejecución del subprograma, el parámetro definido con este tipo devolverá sobre la variable de llamada, el valor actualizado.

EJEMPLO

```
CREATE OR REPLACE PROCEDURE TEST
(V_lectura       IN          NUMBER,
 V_escritura     OUT         NUMBER,
 V_lectu_escri   IN OUT      NUMBER) IS

V_local NUMBER;

     BEGIN
          V_local := v_lectura;       -- Correcto
          V_Lectura := 7;        -- Error de compilación
          V_escritura := 7;           -- Correcto
          V_local := v_escritura; -- Error de compilación
          V_local := v_lectu_escri;  -- Correcto
          V_lectu_escri := 7;         -- Correcto
     END;
```

Al compilar este procedimiento se producirían dos errores en las líneas . `V_Lectura := 7` y `V_local := v_escritura` porque no se admite ni siquiera en compilación que una variable de tipo IN se le asigne un nuevo valor, ni que una variable de tipo OUT se utilice para asociarle su valor a otra.

En resumen cuando se definen parámetros en un subprograma hay que tener en cuenta las siguientes reglas:

- Un parámetro de tipo IN sólo aparecerá en la parte derecha de una asignación.
- Un parámetro de tipo OUT sólo aparecerá en la parte izquierda de una asignación.
- Un parámetro de tipo IN OUT aparecerá en la parte izquierda o derecha de una asignación.

Diferencia entre IS y AS

Como se indicó en la sintaxis de creación de un procedimiento, se puede especificar IS o AS antes de comenzar las líneas de código del bloque. Pero esta posibilidad presenta diferencias en su uso:

- Cuando se indica AS en la sintaxis de creación de un procedimiento no se pueden definir variables locales para utilizar en el procedimiento.

- Cuando se indica IS en la sintaxis de creación de un procedimiento si que se pueden definir variables locales para utilizar en el procedimiento.

En la práctica cuando se está comenzando la definición de un procedimiento del que todavía no se tiene una visión global y concreta, se crea con la cláusula IS para no ponerse impedimentos en la creación o no de variables locales.

EJEMPLO

```
CREATE OR REPLACE PROCEDURE Prueba(V_para1 IN NUMBER) IS
V_local    NUMBER;
BEGIN
     .....
END;

CREATE OR REPLACE PROCEDURE Prueba(V_para1 IN NUMBER) AS
BEGIN
     .....
END;
```

Cuerpo del procedimiento

La sintaxis del cuerpo o bloque interno al procedimiento es la que se indica a continuación:

```
CREATE [OR REPLACE] PROCEDURE nombre_proced
[(parámetros)] {IS | AS}
[/* Sección declaratica */]

BEGIN
      /* Sección ejecutable */
[EXCEPTION]
      /* Control de errores */
END [nombre_proced];
```

La sección declarativa que aparece entre el IS|AS y el BEGIN de comienzo del bloque es opcional y en ella se definen todas las variable y cursores locales al procedimiento.

La indicación de parámetros en la creación del bloque es opcional.

La sección de control de errores es opcional.

Por último, la indicación del nombre del procedimiento al final del mismo (después del END) es también opcional.

Llamada a un procedimiento dentro de un bloque

Para invocar la ejecución de un procedimiento desde un bloque PL/SQL, se utiliza la siguientes sintaxis:

```
nombre_proced [(parámetro1, …., parámetroX)]
```

EJEMPLO

```
CREATE OR REPLACE PROCEDURE Prueba(Var IN number) IS
BEGIN
      .....
END;
```

Tomando este ejemplo de procedimiento, una posible llamada que se podría hacer al mismo desde un bloque PL/SQL sería la siguiente:

```
DECLARE
      Var   NUMBER(5) := 10;
BEGIN
      Prueba(var);
END;
```

Llamada a un procedimiento desde SQL*PLUS

Para invocar la ejecución de un procedimiento directamente desde la línea de comandos de SQL*PLUS se utiliza la siguiente sintaxis

{EXECUTE | EXEC} nombre_procedimiento;

EJEMPLO

```
SQL> VARIABLE variable1;

SQL> EXECUTE simple(:variable1);
```

En este ejemplo se define variable denominada VARIABLE1 y luego se invoca la ejecución del procedimiento SIMPLE, utilizando la instrucción EXECUTE y pasándole como parámetro al procedimiento la variable VARIABLE1 creada anteriormente.

Restricciones sobre los parámetros

La declaración de parámetros de un procedimiento se realiza obligatoriamente sin indicar el tamaño de los tipos. Es decir, se indica el nombre del parámetro, el tipo de parámetro (entrada, salida o entrada/salida) de forma opcional, y el tipo sin el tamaño.

El tamaño del parámetro se asume por la variable de llamada al subprograma.

EJEMPLO

```
CREATE OR REPLACE PROCEDURE Prueba3
(Var1 IN OUT number,
 var2 IN OUT VARCHAR2) AS
BEGIN
      Var1 := 5;
      Select to_number(var2) into var1 from dual;
END;

DECLARE
      Local1     NUMBER(5);
      Local2     VARCHAR2(10);
BEGIN
      Local1 := 10918;
      Local2 := 'JA';
      Prueba3(local1,local2);
      Local2 := 'PP';
END;
```

El parámetro `Var1` asume como tamaño el de la variable `Local1` que tiene NUMBER(5). De la misma forma `Var2` asume como tamaño el de la variable `Local2` que tiene VARCHAR2(10).

Todas las reglas que se han estudiado en la definición de los tipos de datos, son de aplicación a los tipos que se definen en los parámetros de los subprogramas a excepción del tamaño, según lo que se ha explicado anteriormente.

EJEMPLO

```
CREATE OR REPLACE PROCEDURE Prueba
(Para1          estudiantes.nombre%TYPE,
 Para2          estudiantes.edad%TYPE DEFAULT 50,
 Para3          VARCHAR2 DEFAULT 'holA') AS
BEGIN
....
END;

DECLARE
....
BEGIN
      Prueba ('JOSE');
END;
```

Esta llamada es válida porque los otros dos parámetros: `Para2` y `Para3` tienen valores por defecto y se asumen estos valores en la llamada al procedimiento.

Alteración y borrado de procedimientos

Una vez creado y almacenado el procedimiento se puede borrar utilizando la siguiente sintaxis:

```
DROP PROCEDURE nombre_procedimiento;
```

Para alterar un procedimiento creado y almacenado previamente, hay que borrarlo y volverlo a crear, salvo que indiquemos CREATE OR REPLACE PROCEDURE que realiza esta operación automáticamente, reemplazando el código del procedimiento almacenado por el nuevo código.

Notación posicional y nominal de los parámetros

Lo normal es que la llamada a un procedimiento con parámetros se realice en el orden natural en el que se han definido dentro de la cabecera del procedimiento. A este método se le conoce como notación posicional y es la forma que hemos visto hasta el momento en los ejemplos anteriores.

Pero existe una segunda forma de invocar los parámetros de un subprograma que consiste en indicar el nombre del parámetro y su variable asignada. A esta metodología se la denomina como notación nominal.

EJEMPLO DE NOTACIÓN NOMINAL

```
CREATE OR REPLACE PROCEDURE Prueba
(Para1           VARCHAR2,
 Para2           NUMBER,
 Para3           DATE) AS
BEGIN
.......
END;

BEGIN
      Prueba(Para2 => var2,
             Para3 => var3,
             Para1 => var1);
END;
```

FUNCIONES

Una función se estructura de la misma manera que un procedimiento, utilizando un bloque, así como parámetros en la cabecera.

La diferencia principal con los procedimientos es que una función siempre devuelve un valor por si misma, por tanto la llamada siempre tiene que estar asignada a una variable que recogerá el valor que devuelva la función.

La sintaxis es la siguiente:

```
CREATE [OR REPLACE] FUNCTION nombre_funcion
[(argumento1 [{IN | OUT | IN OUT}] tipo,
……
Argumento [{IN | OUT | IN OUT}] tipo)]
RETURN tipo {IS | AS}
Cuerpo_función.
```

La orden RETURN

Se utiliza dentro del cuerpo de la función y se aplica para devolver el control y opcionalmente un valor a la llamada de la función.

La sintaxis es la siguiente:

```
RETURN [expresion];
```

Si no se indica un valor para la expresión que sigue a RETURN se devuelve el control a la llamada de la función sin más.

EJEMPLO

```
CREATE OR REPLACE FUNCTION Prueba
(Para1          NUMBER,
 Para2          VARCHAR2)
RETURN VARCHAR2 AS
BEGIN
      ......
      RETURN 'Hola';
END;


DECLARE
      Saludo          VARCHAR2(10);
BEGIN
      Saludo := Prueba(10,'ADIOS');
END;
```

Eliminación y alteración de funciones

Una vez creada y almacenada una función se puede borrar utilizando la siguiente sintaxis:

```
DROP FUNCTION nombre_funcion;
```

Para alterar una función creada y almacenada previamente, hay que borrarla y volverla a crear salvo que indiquemos CREATE OR REPLACE FUNTION que realiza esta operación automáticamente reemplazando el código anterior por el nuevo.

Llamada a una función dentro de un bloque PL/SQL

Como toda función devuelve un valor, la llamada a una función dentro de un bloque PL/SQL se realiza mediante una asignación a una variable con la siguiente sintaxis:

```
Variable := nombre_funcion;
```

Llamada a una función en una instrucción SELECT

También es posible realizar una llamada a una función dentro de una instrucción SELECT, dentro del SELECT…FROM como si fuese otra columna más a visualizar o incluso en la sección WHERE como otro condicionante.

EJEMPLO DE LLAMADA A UNA FUNCION EN UNA INSTRUCCIÓN SELECT COMBINADA CON COLUMNAS

```
SELECT nombre, apellidos, SYSDATE FROM empleados;
```

En este ejemplo se seleccionan todas las filas de la tabla empleados, mostrándose únicamente el nombre y apellidos de los empleados más la fecha actual del sistema invocando la llamada a la función SYSDATE. Esta función es genérica del lenguaje PL/SQL y retorna la fecha y hora del sistema donde se encuentra instalada la base de datos.

EJEMPLO DE LLAMADA A UNA FUNCION DENTRO DEL WHERE DEL SELECT

```
SELECT nombre, apellidos FROM empleados
WHERE f_baja = SYSDATE
```

En este ejemplo se seleccionan aquellas filas de la tabla empleados que cumplan que la fecha de baja del empleado sea igual a la fecha actual (SYSDATE), mostrándose únicamente el nombre y apellidos de los empleados.

EJEMPLO DE LLAMADA ÚNICA A UNA FUNCION DENTRO DEL SELECT

```
SELECT SYSDATE FROM DUAL;
```

En este ejemplo únicamente se muestra la fecha actual del sistema. Como no hay acceso a información de ninguna tabla, se utiliza la pseudotable DUAL para completar la sentencia SELECT y mostrar el valor de la función SYSDATE.

Tablas del sistema asociadas

Los procedimientos y funciones como las tablas y demás objetos se almacenan en la base de datos y como tal en el diccionario de datos.

Las vistas del diccionario que almacenan datos de los subprogramas son las siguientes:

- USER_OBJECTS
- USER_SOURCE
- USER_ERRORS

USER_OBJECTS

Esta tabla del sistema almacena el nombre y tipo de todos los objetos del usuario conectado a la base de datos.

USER_SOURCE

Esta tabla del sistema almacena el código fuente de los subprogramas del usuario conectado a la base de datos.

USER_ERRORS

Esta tabla del sistema almacena los errores (línea y error) de los subprogramas que han producido algún error en compilación.

EJEMPLO

```
SQL> CREATE OR REPLACE PROCEDURE SIMPLE IS
  2  V_COUNTER  NUMBER;
  3  BEGIN
  4     V_COUNTER := 7;
  5  END;
  6  /
```

Creamos un procedimiento denominado SIMPLE.

```
SQL> DESC USER_OBJECTS;
 Name                                      Nulo?    Type
 ----------------------------------------- -------- ----------------
OBJECT_NAME                                         VARCHAR2(128)
SUBOBJECT_NAME                                      VARCHAR2(128)
OBJECT_ID                                           NUMBER
DATA_OBJECT_ID                                      NUMBER
OBJECT_TYPE                                         VARCHAR2(23)
CREATED                                             DATE
LAST_DDL_TIME                                       DATE
TIMESTAMP                                           VARCHAR2(19)
STATUS                                              VARCHAR2(7)
TEMPORARY                                           VARCHAR2(1)
GENERATED                                           VARCHAR2(1)
SECONDARY                                           VARCHAR2(1)
NAMESPACE                                           NUMBER
EDITION_NAME                                        VARCHAR2(128)
SHARING                                             VARCHAR2(18)
EDITIONABLE                                         VARCHAR2(1)
ORACLE_MAINTAINED                                   VARCHAR2(1)
APPLICATION                                         VARCHAR2(1)
DEFAULT_COLLATION                                   VARCHAR2(100)
DUPLICATED                                          VARCHAR2(1)
SHARDED                                             VARCHAR2(1)
CREATED_APPID                                       NUMBER
CREATED_VSNID                                       NUMBER
MODIFIED_APPID                                      NUMBER
MODIFIED_VSNID                                      NUMBER
```

Invocamos la instrucción DESC (equivalente a DESCRIBE) para visualizar la lista de columnas de la tabla USER_OBJECTS.

```
SQL> SELECT OBJECT_NAME, OBJECT_TYPE, STATUS
  2  FROM USER_OBJECTS WHERE OBJECT_NAME = 'SIMPLE';

OBJECT_NAME          OBJECT_TYPE         STATUS
-------------------- ------------------- -----------
SIMPLE               PROCEDURE           VALID
```

Se seleccionan las columnas OBJECT_NAME, OBJECT_TYPE y STATUS de la tabla USER_OBJECTS para el objeto con nombre (OBJECT_NAME) 'SIMPLE'. El resultado de la consulta devuelve otra vez el nombre del objeto, el tipo de objeto (en este caso PROCEDURE) y el estado de compilación del ojbjeto: VALID.

Un STATUS VALID equivale a que el objeto no tiene errores de compilación, mientras que un STATUS INVALID indica que el objeto en cuestión tiene errores de compilación y por tanto no podrá ser ejecutado.

```
SQL> SELECT TEXT FROM USER_SOURCE
  2  WHERE NAME = 'SIMPLE' ORDER BY LINE;

TEXT
------------------------------------------------------

PROCEDURE SIMPLE IS
V_COUNTER       NUMBER;
BEGIN
        V_COUNTER := 7;
END;
```

En este caso se consulta la columna TEXT de la tabla USER_SOURCE para obtener el código fuente del objeto 'SIMPLE'.

```
SQL> CREATE OR REPLACE PROCEDURE SIMPLE2 IS
  2  V_COUNTER NUMBER
  3  BEGIN
  4  V_COUNTER = 7;
  5  END;
  6  /

LINE/COL  ERROR
--------- -------------------------------------------------
-------------
3/3         PLS-00103: Se ha encontrado el símbolo "BEGIN"
cuando se esperaba uno de los siguientes:     := . ( @ % ;
not null range default character El símbolo ":= insertado
antes de "BEGIN" para continuar.
5/3         PLS-00103: Se ha encontrado el símbolo "END"
cuando se esperaba uno de los siguientes:    begin function
```

```
pragma procedure subtype type <an identifier>   <a double-
quoted   delimited-identifier>   current   cursor   delete
exists prior
Errores: comprobar log de compilador
```

En este caso se crea un procedimiento SIMPLE2 con errores de compilación. Esto nos lo informa el sistema a través del mensaje `Errores: comprobar log de compilador`

```
SQL> SELECT LINE, POSITION, TEXT
  2  FROM USER_ERRORS
  3  WHERE NAME = 'SIMPLE2'
  4  ORDER BY SEQUENCE;

      LINE   POSITION
---------- ----------
TEXT
--------------------------------------------------------------
         3          1
PLS-00103: Se ha encontrado el símbolo "BEGIN" cuando se
esperaba uno de los siguientes:

   := . ( @ % ; not null range default character
El símbolo ":= insertado antes de "BEGIN" para continuar.

         5          1
PLS-00103: Se ha encontrado el símbolo "END" cuando se
esperaba uno de los siguientes:


      LINE   POSITION
---------- ----------
TEXT
--------------------------------------------------------------
   begin function pragma procedure subtype type <an
identifier>
   <a double-quoted delimited-identifier> current cursor
delete
   exists prior
```

En este caso consultamos las columnas LINE, POSITION y TEXT de la tabla USER_ERRORS para mostrar donde se encuentran los errores de compilación del procedimiento SIMPLE2, y cuál es el motivo del error de compilación.

Situaciones que provocan el estado INVALID

Cuando se modifica un objeto de tipo tabla para añadir o borrar columnas, e incluso para modificar el tamaño o tipo de las mismas, todos los subprogramas que hagan referencia a este objeto se se ponen a estado INVALID en la tabla USER_OBJECTS.

Cuando se recrea un procedimiento o función y se generan errores de compilación en el mismo, todos los subprogramas que invoquen a este automáticamente se ponen a estado INVALID en la tabla USER_OBJECTS.

Instrucciones que recompilan elementos descompilados

A continuación, se relacionan las instrucciones que permiten recompilar objetos descompilados:

Para recompilar un **procedimiento** INVALID:

```
ALTER PROCEDURE <nombre_procedimiento> COMPILE;
```

Para recompilar una **función** INVALID:

```
ALTER FUNCTION <nombre funcion> COMPILE;
```

Para recompilar un **trigger** INVALID:

```
ALTER TRIGGER <nombre trigger> COMPILE;
```

Para recompilar un **paquete** INVALID:

```
ALTER PACKAGE <nombre paquete> COMPILE;
ALTER PACKAGE <nombre paquete> COMPILE BODY;
```

Para recompilar una vista INVALID:

```
ALTER VIEW <nombre trigger> COMPILE;
```

Instrucciones que recompilan esquemas enteros

Cuando se quiere recompilar todos los elementos de código incluidos en un esquema de base de datos, también se puede utilizar una única instrucción:

```
EXEC DBMS_UTILITY.COMPILE_SCHEMA(schema => 'nombre_esquema');
```

Llamada a una función desde SQL*PLUS

Para invocar la ejecución de una función dentro del SQL*PLUS sin utilizar una instrucción SELECT se utiliza la siguiente sintaxis:

```
SQL> VARIABLE nombre;

SQL> :nombre := función;
```

RESTRINGIENDO PERMISOS DE USO A SUBPROGRAMAS

Como normal general, cuando se crea un nuevo subprograma (sea procedimiento, función o paquete), el mismo es accesible por los otros objetos que comparten el mismo esquema de base de datos.

Con la versión 12c de Oracle, se ha añadido una nueva funcionalidad para restringir los permisos de acceso al subprograma a ciertos objetos. Esta restricción se realiza dentro de la definición del propio subprograma con la cláusula ACCESSIBLE BY.

La sintaxis es la siguiente:

```
CREATE [OR REPLACE] SUBPROGRAMA nombre_proced
[(parámetros)] [{IS | AS}]
ACCESSIBLE BY (PROCEDURE nombre | FUNCTION nombre |
PACKAGE nombre,…)
[/* Sección declaratica */]
```

EJEMPLO RESTRINGIENDO PERMISOS DE ACCESO A UN PROCEDIMIENTO

```
CREATE OR REPLACE PROCEDURE p_demo_acc
ACCESSIBLE BY (PROCEDURE proc1, FUNCTION fun1) IS
BEGIN
dbms_output.put_line('El procedimiento P_DEMO_ACC se ha
ejecutado correctamente.');
END;
```

Mediante este ejemplo se crea un procedimiento denominado P_DEMO_ACC al que sólo se puede acceder llamándolo desde el procedimiento PROC1 o desde la función FUN1.

Si ahora ejecutamos la llamada al procedimiento P_DEMO_ACC desde un bloque sin nominar como se indica a continuación…

```
SQL> SET SERVEROUTPUT ON
     BEGIN
       P_DEMO_ACC;
     END;
```

Obtendremos el siguiente mensaje de error que nos avisa de que no existen privilegios para la ejecución del procedimiento P_DEMO_ACC.

```
Informe de error -
ORA-06550: línea 2, columna 3:
PLS-00904: insuficientes privilegios para acceder al objeto
P_DEMO_ACC
06550. 00000 -  "line %s, column %s:\n%s"
*Cause:    Usually a PL/SQL compilation error.
*Action:
```

Para comprobar que los privilegios que nos impiden la ejecución del procedimiento P_DEMO_ACC no son los del esquema, vamos a crearnos otro procedimiento, en este caso PROC1, que es uno de los procedimientos que tienen permitido la ejecución del procedimiento P_DEMO_ACC. Dentro de dicho procedimiento vamos a incluir la llamada al procedimiento P_DEMO_ACC.

```
CREATE OR REPLACE PROCEDURE proc1 IS
BEGIN
      p_demo_acc;
END;
```

Si ahora ejecutamos la llamada al procedimiento PROC1 desde un bloque sin nominar como se indica a continuación…

```
SQL> SET SERVEROUTPUT ON
      BEGIN
        PROC1;
      END;
```

Veremos que no sólo se ejecuta correctamente el procedimiento PROC1, si no que también se ejecuta el procedimiento restringido P_DEMO_ACC.

```
El procedimiento P_DEMO_ACC se ha ejecutado correctamente.
Procedimiento PL/SQL terminado correctamente.
```

EJEMPLO RESTRINGIENDO PERMISOS DE ACCESO A UNA FUNCION

```
CREATE OR REPLACE FUNCTION p_demo_acc2 RETURN VARCHAR2
ACCESSIBLE BY (PROCEDURE proc1, FUNCTION fun1) IS
BEGIN
dbms_output.put_line('Executed    from    a    white-list
program');
RETURN '0';
END;
```

Mediante este ejemplo se crea una función denominada P_DEMO_ACC2 a la que sólo se puede acceder llamándola desde el procedimiento PROC1 o desde la función FUN1.

Si ahora ejecutamos la llamada a la función P_DEMO_ACC2 desde una consulta como se indica a continuación…

```
SQL> SELECT P_DEMO_ACC2 FROM DUAL;
```

Obtendremos el siguiente mensaje de error que nos avisa de que no existen privilegios para la ejecución del procedimiento P_DEMO_ACC2.

```
ORA-06553: PLS-904: insuficientes privilegios para acceder
al objeto P_DEMO_ACC2
06553. 00000 -  "PLS-%s: %s"
*Cause:
*Action:
Error en la línea: 1, columna: 8
```

EJEMPLO RESTRINGIENDO PERMISOS DE ACCESO A UN PAQUETE

```
CREATE OR REPLACE PACKAGE p_demo_acc3
ACCESSIBLE BY (PROCEDURE proc1, FUNCTION fun1) IS

PROCEDURE prueba;

END;
```

Mediante este ejemplo se crea la especificación de un paquete denominado P_DEMO_ACC3 al que sólo se puede acceder llamándolo desde el procedimiento PROC1 o desde la función FUN1.