Tema 12.Programación en Bases De Datos (I)

Contenido

1. Introducción

Hasta el momento, hemos trabajado con la base de datos de manera interactiva: el usuario introduce un comando y Oracle da una respuesta. Esta forma de trabajar, incluso con un lenguaje tan sencillo y potente como SQL, no resulta operativa en un entorno de producción, ya que supondría que todos los usuarios conocen y manejan SQL, y además está sujeta a frecuentes errores.

También hemos creado pequeños scripts de instrucciones SQL. No obstante, estos scripts tienen importantes limitaciones en cuanto al control de la secuencia de ejecución de instrucciones, el uso de variables, la modularidad, la gestión de posibles errores, etc.

Para superar estas limitaciones, los SGBD incorporan lenguajes de tipo procedimental que permiten manipular de forma más avanzada los datos de las bases de datos.

Oracle incorpora un gestor **PL/SQL** en el servidor de la base de datos y en las principales herramientas (Forms, Reports, Graphics, etc.), es una extensión procedimental del lenguaje SQL, es decir, se trata de un lenguaje creado para dar a SQL nuevas posibilidades que permiten utilizar condiciones y bucles al estilo de los lenguajes de tercera generación (C++, JAVA,…).

En otros SGBD existen otros lenguajes procedimentales: **SQL Server** utiliza **Transact SQL**, **Informix** usa **Informix 4GL**,…

Lo interesante de PL/SQL es que integra SQL, por lo que gran parte de su sintaxis procede de él. Este lenguaje, basado en el lenguaje ADA, incorpora todas las características propias de los lenguajes de tercera generación: manejo de variables, estructura modular (procedimientos y funciones), estructuras de control (bifurcaciones, bucles y demás estructuras), control de excepciones, así como una total integración en el entorno Oracle.

Los programas creados con PL/SQL se pueden almacenar en la base de datos como cualquier otro objeto de ésta; de este modo se facilita a todos los usuarios autorizados el acceso a estos programas. Esta forma de trabajo facilita enormemente la distribución, la instalación y el mantenimiento de software y reduce drásticamente los costes asociados a estas tareas. Además, los programas se ejecutan en el servidor, con el consiguiente ahorro de recursos en los clientes y disminución del tráfico de red ya que en vez de enviar muchas instrucciones, los usuarios realizan operaciones enviando una única instrucción, lo cual disminuye el número de solicitudes entre el cliente y el servidor.

El uso del lenguaje PL/SQL es también imprescindible para construir *disparadores* de bases de datos que permiten implementar reglas complejas de negocio y auditoría en la base de datos.

Por todo ello, el conocimiento de este lenguaje es imprescindible para poder trabajar en el entorno Oracle, tanto para administradores de la base de datos como para desarrolladores de aplicaciones.

Características

PL/SQL es un lenguaje procedimental diseñado por Oracle para trabajar con la base de datos. Está incluido en el servidor y en algunas herramientas de cliente. Soporta todos los comandos de consulta y manipulación de datos, aportando al lenguaje SQL las estructuras de control (bucles, bifurcaciones, etc.) y otros elementos propios de los lenguajes procedimentales de tercera generación. **Su unidad de trabajo es el bloque**, constituido por un conjunto de declaraciones, instrucciones y mecanismos de gestión de errores y excepciones.

1.1 Conceptos básicos

bloque PL/SQL

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

programa PL/SQL

Conjunto de bloques que realizan una determinada labor.

procedimiento

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

función

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

trigger (disparador)

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

paquete

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

1.2 Características del lenguaje

1.2.1 Escritura de instrucciones PL/SQL

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

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

Comentarios

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

1.2.2 Bloques PL/SQL

Con PL/SQL se pueden construir distintos tipos de programas: procedimientos, funciones, etcétera; todos ellos tienen en común una estructura básica característica del lenguaje denominada BLOQUE.

Un bloque tiene tres zonas claramente definidas:

- Una zona de *declaraciones* donde se declaran objetos (variables, constantes, etc. locales. Suele ir precedida por la cláusula DECLARE (o IS/AS en los procedimientos y funciones). Es opcional.
- Un conjunto de *instrucciones* precedido por la cláusula BEGIN.
- Una zona de tratamiento de *excepciones* precedido por la cláusula EXCEPTION. Esta zona, igual que la de declaraciones, es opcional.

El formato genérico del bloque es:

```
[ DECLARE
<declaraciones> ]
BEGIN
<órdenes>
[ EXCEPTION
<gestión de excepciones> ]
END;
```

/*La zona de declaraciones comenzará con **DECLARE** o con **IS**, dependiendo del tipo de bloque. Las únicas cláusulas obligatorias son **BEGIN** y **END** */

En el siguiente ejemplo se borra el departamento número 20, pero antes se crea un departamento provisional, al que se asigna los empleados del departamento 20 que de otra forma hubieran sido borrados al borrar el departamento. También informa del número de empleados afectados.

Para que se visualice el resultado, es necesario modificar la variable de entorno **SERVEROUTPUT**

```
SET SERVEROUTPUT ON
```

Ejemplo 1

```
DECLARE
v_num_empleados  NUMBER(2);
BEGIN
  INSERT INTO depart
  VALUES (99,'PROVISIONAL',NULL);
  UPDATE emple SET dept_no =99
    WHERE dept_no = 20;
  v_num_empleados := SQL%ROWCOUNT ;
  DELETE FROM depart
    WHERE dept_no = 20;
  DBMS_OUTPUT.PUT_LINE( v_num_empleados || ' empleados ubicados en PROVISIONAL') ;
EXCEPTION
  WHEN OTHERS THEN
    ROLLBACK;
  RAISE_APPLICATION_ERROR(-20000, 'Error en la aplicación');
END;
```

La utilización de bloques supone una notable mejora de rendimiento, ya que se envían los bloques completos al servidor para que sean procesados, en lugar de cada sentencia SQL. Así se ahorran muchas operaciones de E/S.

1.2.3 Definición de datos compatible con SQL

PL/SQL dispone de tipos de datos compatibles con los tipos utilizados para las columnas de las tablas: NUMBER, VARCHAR2, DATE, etc., además de otros propios, como BOOLEAN. Las declaraciones de los datos deben realizarse en la sección de declaraciones:

```
DECLARE
Importe        NUMBER(8,2);
Contador       NUMBER(2)   DEFAULT 0;
Nombre         CHAR(20)    NOT NULL := "MIGUEL";
Nuevo                  VARCHAR2(15) ;
BEGIN
……
```

PL/SQL permite declarar una variable del mismo tipo que otra variable o que una columna de una tabla mediante el atributo **%TYPE**.

Ejemplo 2

```
NombreAct    Empleados.Nombre%TYPE
```

Declara la variable NombreAct, que es del mismo tipo que la columna Nombre de la tabla Empleados.

También se puede declarar una variable para guardar una fila completa de una tabla mediante el atributo **%ROWTYPE**.

Ejemplo 3

```
Mifila Empleados%ROWTYPE.
```

Declara la variable Mifila, que es del mismo tipo que las filas de la tabla Empleados.

1.2.4 Estructuras de control

Las estructuras de control de PL/SQL son las habituales de los lenguajes de programación estructurados: IF, WHILE, FOR y LOOP.

Estructuras de control alternativas		
Alternativa simple	Alternativa doble	Alternativa múltiple
IF <condición> THEN instrucciones; …… END IF;	IF <condición> THEN instrucciones; …… ELSE instrucciones; ……. END IF;	IF <condición> THEN instrucciones; ….. ELSIF <condición2> THEN instrucciones; …….. ELSIF <condición3> THEN instrucciones; ……. ELSE instrucciones; END IF;

Estructuras de control repetitivas		
Mientras	Para	Iterar
WHILE <condición> LOOP Instrucciones; …… END LOOP;	FOR <variable> IN <mínimo>. . <máximo> LOOP instrucciones; …… END LOOP;	LOOP instrucciones; …… EXIT WHEN <condición>; …… instrucciones; …… END LOOP;

1.2.5 Soporte para órdenes de manipulación de datos

Desde PL/SQL se puede ejecutar cualquier orden de manipulación de datos.

Ejemplos:

```
DELETE FROM clientes WHERE nif = Vnif;
```

Borra de la tabla clientes la fila correspondiente al cliente cuyo NIF se especifica en la variable Vnif.

……

```
UPDATE PRODUCTOS SET STOCK := STOCK - UnidadesVendidas
WHERE CodProducto = Vcodigo;
```

….

Actualiza en la tabla Productos la columna STOCK, correspondiente al código de producto especificado en la variable Vcodigo, restándole el valor que se especifica en la variable Unidades Vendidas.

```
INSERT  INTO  clientes  VALUES (v_num,  v_nom,  v_loc, . . . );
```

Añade a la tabla CLIENTES una fila con los valores contenidos en las variables que se especifican.

1.2.6 Uso de cursores

En PL/SQL el resultado de una consulta no va directamente al terminal del usuario, sino que se guarda en un área de memoria a la que se accede mediante una estructura denominada cursor. Los cursores sirven para guardar el resultado de una consulta.

A este tipo de cursores se les denomina **cursores implícitos**, ya que no hay que declararlos. Es el tipo más sencillo, pero tiene ciertas limitaciones: la consulta deberá *devolver una única fila*, pues en caso contrario se producirá un error.

Para guardar el resultado de la siguiente consulta en la variable VnumVentas, que deberá haber sido declarada previamente, emplearemos

```
SELECT COUNT(*) INTO VnumVentas FROM Ventas;
```

El formato básico es:

```
SELECT <columna/s> INTO <variable/s> FROM <tabla>
[WHERE ..-etc] ;
```

La/s variable/s que siguen al **INTO** reciben el valor de la consulta. Por tanto, debe haber coincidencia en el tipo con las columnas especificadas en la cláusula **SELECT** y el número de variables.

Ejemplo 3

Escribir el apellido y el oficio del empleado número 7900

```
DECLARE
   V_ape       VARCHAR2(10);-- mejor          emple.apellido%TYPE
   V_oficio    VARCHAR2(10);--  mejor          emple.oficio%TYPE
BEGIN
   SELECT apellido, oficio INTO v_ape, v_oficio FROM emple
   WHERE emp_no = 7900;
   DBMS_OUTPUT.PUT_LINE  (v_ape || '*'|| v_oficio) ;
END;
```

1.2.7 Gestión de excepciones

Las excepciones sirven para tratar errores y mensajes de las diversas herramientas. Oracle tiene determinadas excepciones correspondientes a algunos de los errores más frecuentes que se producen al trabajar con la base de datos, como por ejemplo:

NO_DATA_FOUND. Una orden de tipo SELECT INTO no ha devuelto ningún valor.

TOO_MANY_ROWS. Una orden SELECT INTO ha devuelto más de una fila.

Se disparan automáticamente al producirse los errores asociados.

El ejemplo anterior con gestión de excepciones sería:

Ejemplo:

Crear la tabla TEMP con una columna col1 VARCHAR2(50)

```
DECLARE
  V_ape VARCHAR(10);
  V_oficio VARCHAR(10);
BEGIN
  SELECT apellido, oficio INTO v_ape, v_oficio
  FROM emple WHERE emp_no = 7900;
  DBMS_OUTPUT.PUT_LINE(v_ape || '*' || v_oficio) ;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    INSERT INTO  temp (col1) VALUES ('ERROR no hay datos');
  WHEN TOO_MANY_ROWS THEN
    INSERT  INTO temp (col1) VALUES ('ERROR demasiados datos');
  WHEN OTHERS THEN
    RAISE_APPLICATION_ERROR (-20000, 'Error en la aplicación');
END;
```

Si PL/SQL detecta una excepción, pasa el control a la cláusula WHEN correspondiente de la sección EXCEPTION del bloque PL, que lo tratará según lo establecido. Al finalizar este tratamiento, se devuelve el control al programa que llamó al bloque que trató la excepción.

1.2.8 Estructura modular

En una aproximación inicial podemos distinguir los siguientes tipos de programas:

- *BLOQUES ANÓNIMOS*. No tienen nombre. La zona de declaraciones comienza con la palabra DECLARE. Son las estructuras utilizadas fundamentalmente en los primeros ejercicios. Su utilización real es escasa.
- *SUBPROGRAMAS.* Son bloques PL/SQL que tienen un nombre. La zona de declaraciones comienza con la palabra *IS ó AS*. A su vez, pueden ser de dos tipos:
    - *PROCEDIMIENTOS*. Es el tipo de programas más usado en PL/SQL. Normalmente se almacenan en la base de datos. Su formato genérico es el siguiente:

```
CREATE OR REPLACE PROCEDURE <nombreprocedimiento>
 [(<lista de parámetros> )]

IS | AS
      [<declaraciones objetos locales>;]
BEGIN
      <instrucciones>;
[EXCEPTION
       <excepciones>;]
END [<nombreprocedimiento>];
```

Se pueden distinguir dos partes en el procedimiento: **la cabecera**, que es donde va el nombre del procedimiento y los parámetros, y el **cuerpo** del procedimiento (la zona cursiva), que es un bloque PL/SQL.

- *FUNCIONES*. Su formato genérico es similar al de los procedimientos, pero éstas pueden devolver un valor.

1.3 Interacción con el usuario en PL/SQL

PL/SQL no es un lenguaje creado para interactuar con el usuario, sino para trabajar con la base de datos. Prueba de ello es el hecho de que no dispone de órdenes o sentencias que capturen datos introducidos por teclado, ni tampoco para visualizar datos en la pantalla. No obstante, Oracle incorpora el paquete DBMS_OUTPUT con fines de depuración. Éste incluye, entre otros, el procedimiento PUT_LINE, que permite visualizar textos en la pantalla.

Puesto que para aprender a programar en un lenguaje se necesita probar frecuentemente los programas y visualizar sus resultados, utilizaremos el procedimiento PUT_LINE con este fin, sabiendo que en un entorno de producción se deberán emplear herramientas como ORACLE FORMS para visualizar los resultados. El formato genérico para invocar a este procedimiento es el siguiente:

```
DBMS_OUTPUT.PUT_LINE(<expresión>);
```

Para que funcione correctamente, la variable de entorno **SERVEROUTPUT** deberá estar en **ON**; en caso contrario no se visualizará nada. Para cambiar el estado de la variable introduciremos al comienzo de la sesión:

```
SET SERVEROUTPUT ON
```

Para pasar datos a un programa podemos recurrir a una de las siguientes opciones:

- Introducir datos en una tabla desde SQLDeveloper y, después, leerlos desde el programa.
- Utilizar variables de sustitución (en bloques anónimos).
- Pasar los datos como parámetros en la llamada (en procedimientos y funciones).

1.4 Arquitectura

PL/SQL no es un producto independiente, sino una tecnología integrada en el servidor Oracle y también en algunas herramientas. Se trata de un motor o gestor que es capaz de ejecutar subprogramas y bloques PL/SQL, que trabaja en coordinación con el ejecutor de órdenes SQL. Existen diferencias importantes entre el motor PL/SQL del servidor y el de las herramientas.

PL/SQL del servidor Oracle

Podemos diferenciar tres tipos de bloques PL/SQL que trabajan con el motor del servidor:

- **Bloques PL/SQL anónimos**. Se pueden generar con diversas herramientas, y se envían al servidor Oracle, donde serán compilados y ejecutados.
- **Subprogramas almacenados**. Se trata de procedimientos y funciones que se compilan y almacenan en la base de datos, donde quedarán disponibles para ser ejecutados. Por razones de organización y funcionalidad, estos procedimientos y funciones también se pueden agrupar en paquetes, los cuales se almacenan también en la base de datos. En este caso, reciben el nombre de subprogramas empaquetados.
- **Disparadores de base de datos** (**trigers**). Son subprogramas almacenados que se asocian a una tabla de la base de datos. Estos subprogramas se ejecutan automáticamente al producirse determinados cambios en la tabla (inserción, borrado o modificación). Son muy útiles para controlar los cambios que suceden en la base de datos, para implementar restricciones complejas, etc.

PL/SQL de las herramientas

Algunas herramientas de Oracle (Forms, Reports, Graphics, etc.) contienen un motor PL/SQL capaz de procesar bloques PL/SQL ejecutando las órdenes procedimentales en el cliente o el lugar donde se encuentre la herramienta, y enviando solamente las instrucciones SQL al servidor Oracle.

1.5 Uso de bloques anónimos y procedimientos

1.5.1 Bloques anónimos

Como ya se ha explicado, los bloques anónimos son bloques que no tienen nombre. **Desde el prompt de SQL*Plus** Oracle reconoce el comienzo de un bloque anónimo cuando encuentra la palabra reservada DECLARE o BEGIN. Cuando esto ocurre: limpia el buffer SQL, ignora los ; y entra en modo INPUT.

La última línea del bloque debe ser un punto (.). Esto provoca que se guarde todo el bloque en el buffer SQL. Una vez guardado, podremos ejecutarlo mediante la orden RUN. También se puede usar la barra oblicua (/) en lugar del punto. En este caso, además de almacenarse en el buffer, se ejecutará.

El bloque del buffer se puede guardar en un fichero con la orden:

SAVE nombrefichero [**REPLACE**]

La opción REPLACE se usará si el fichero ya existía.

Para cargar un bloque de un fichero en el buffer SQL se hará:

get nombre_fichero;

Una vez cargado se puede ejecutar con RUN o con /.

También se puede cargar y ejecutar con una sola orden:

start nombre_fichero

Para guardar los bloques anónimos en ficheros **`.SQL`** hay que poner la ruta del directorio donde se va a crear el fichero. También podemos crear una carpeta en la que los guardemos todos y, en las propiedades del acceso directo a SQLPlus, poner la ruta de dicha carpeta en el apartado **`Iniciar en.`**

Éstos son Algunos ejemplos de bloques PL/SQL anónimos:

El siguiente bloque no hace nada:

```
BEGIN
  NULL;
END;
/
Procedimiento PL/SQL terminado con éxito.
```

El siguiente bloque escribe 'HOLA MUNDO'.

```
BEGIN
   DBMS_OUTPUT.PUT_LINE('HOLA MUNDO') ;
END;
/
HOLA MUNDO
Procedimiento PL/SQL terminado con éxito.
```

Ejecutar el script de **tablas unidad 12.txt**.

Este bloque muestra el precio del producto especificado:

```
DECLARE
   V_precio     NUMBER;
BEGIN
      SELECT precio_uni INTO v_precio
      FROM productos
      WHERE COD_PRODUCTO = 7;
      DBMS_OUTPUT.PUT_LINE(v_precio) ;
END;
/
```

```
20000
Procedimiento PL/SQL terminado con éxito.
```

En los bloques PL/SQL se pueden utilizar variables de sustitución anteponiendo el & a la variable. Antes de ejecutar el bloque se solicitará valor para la variable:

El ejercicio anterior con variable de sustitución

```
DECLARE
    V_precio  NUMBER;
BEGIN
    SELECT precio_uni INTO v_precio
    FROM productos
    WHERE COD_PRODUCTO = &var_producto;
    DBMS_OUTPUT.PUT_LINE(v_precio) ;
END;
/
Introduzca un valor para var_producto: 4
40000

Procedimiento PL/SQL terminado con éxito.
```

Visualizar el nombre del cliente, introduciendo como variable el NIF

```
DECLARE
   v_nom       CLIENTES.NOMBRE%TYPE;
BEGIN
   SELECT nombre INTO v_nom
   FROM clientes
   WHERE nif= '&vs_nif' ;
   DBMS_OUTPUT.PUT_LINE(v_nom) ;
END;
/
Introduzca un valor para vs_nif: 333C
antiguo   6:     WHERE NIF='&vs_nif';
nuevo   6:     WHERE NIF='333C';
TERESA
```

Además, los bloques se pueden anidar para distintos propósitos.

```
DECLARE
…….
BEGIN
……
      DECLARE -- comienzo bloque anidado
      …..
      BEGIN
      ……
      END     -- fin bloque anidado
……
 EXCEPTION
……
END
```

Si trabajamos con **SQLDeveloper**, en lugar de escribir el carácter '/' a continuación de de 'END;', le daremos al botón de 'ejecutar script'.

1.5.2 Uso de procedimientos

Introduciendo las siguientes líneas desde **SQLDeveloper** dispondremos de un procedimiento PL/SQL sencillo para consultar los datos de un cliente:

```
CREATE OR REPLACE
PROCEDURE ver_cliente (p_nomcli_i    VARCHAR2)
IS
    v_nifcli     clientes.nif%TYPE;
    v_domicli    clientes.domicilio%TYPE;
BEGIN
   SELECT nif, domicilio INTO v_nifcli, v_domicli
   FROM clientes
   WHERE nombre=p_nomcli_i;
   DBMS_OUTPUT.PUT_LINE('Nombre: ' || p_nomcli_i|| ' Nif: ' ||
                v_nifcli || ' Domicilio: ' || v_domicli);
EXCEPTION
  WHEN NO_DATA_FOUND THEN
      DBMS_OUTPUT.PUT_LINE( 'No encontrado el cliente ' ||p_nomcli_i);
END ver_cliente;
/
```

- 1ª línea indica que queremos crear (CREATE) un objeto o reemplazarlo (OR REPLACE), en el caso de que exista, por uno nuevo.
- 2ª. El objeto que se va a crear es un procedimiento (PROCEDURE) y su nombre es ver_cliente. Este procedimiento recibirá el parámetro de entrada p_nomcli_i, de tipo VARCHAR2.
- 3ª. La palabra reservada IS especifica el comienzo de la zona de declaraciones donde se indicarán los objetos que intervendrán en el programa: variables, constantes, etc.
- 4ª y 5ª. Declaramos las variables v_nifcli y v_domicli de tipo y longitudes iguales que las columnas correspondientes de la tabla clientes.
- 6ª. La palabra reservada BEGIN indica el comienzo de la zona de instrucciones.
- 7ª, 8ª y 9ª. Es una cláusula SELECT con las siguientes particularidades:
    - El resultado de la SELECT se depositará en las variables que siguen al INTO; en este caso, v_nifcli y v_domicli. La asignación se realiza siguiendo un criterio posicional.
    - La condición especificada en la cláusula WHERE contiene el parámetro con el que se llamó al programa, que será el nombre cuyos datos se requieren.
- 10ª. El procedimiento DBMS_OUTPUT.PUT_LINE visualiza en pantalla el contenido de las variables y los literales. Las dos barras (||) sirven para concatenar. Para que este procedimiento funcione correctamente la variable SERVEROUTPUT debe estar en ON antes de ejecutar el programa.
- 11ª. Es continuación de la instrucción comenzada en la línea anterior. En PL/SQL se pueden utilizar todas las líneas que se requieran para una instrucción. La finalización de la orden la determina el punto y coma (;).
- 12ª. La palabra reservada EXCEPTION indica el comienzo de la zona de tratamiento de excepciones. Esta zona no se ejecutará a no ser que se produzca alguna excepción. Por ejemplo, si al ejecutarse la cláusula SELECT no se encuentra ninguna fila de datos que cumpla la condición, se levantará la excepción NO_DATA_FOUND, se bifurcará el control del programa a esta sección, se ejecutará, si existe, el tratamiento para dicha excepción, y se dará por finalizado el programa.
- 13ª. "Caza" la excepción NO_DATA_FOUND, en el caso de que se produzca, y ejecuta las instrucciones que siguen a la cláusula THEN.
- 14ª. Visualiza el mensaje de error "No encontrado el cliente X".

- 15ª. Es el fin del procedimiento. Forma parte de la sintaxis de PL/SQL.
- 16ª. Indica el final del bloque PL/SQL. Compila y guarda en la base de datos el programa introducido (si trabajamos con SQLDeveloper no hace falta ponerla).

La respuesta de Oracle será: procedimiento creado.

Si el compilador detecta errores, en lugar de este mensaje, veremos el siguiente:

```
Aviso: Procedimiento creado con errores de compilación.
```

La orden SHOW ERRORS permite ver los errores detectados. Supongamos que hemos olvidado poner el punto y coma (;) al final de la orden SELECT

```
SHOW ERRORS
Errores para PROCEDURE   VER_CLIENTE:

9/3      PLS-00103: Se ha encontrado el símbolo " DBMS_OUTPUT" cuando se esperaba uno de los siguientes:
. ( * @ % & - + ; / for mod rem an exponent (**) and or group having intersect minus order start unión where connect || El símbolo "." ha sido sustituido por " DBMS_OUTPUT" para continuar.
```

Una vez subsanado el error se ejecuta y el procedimiento se almacena en el servidor de la base de datos y puede ser invocado desde cualquier estación por cualquier usuario autorizado y con cualquier herramienta.

Por ejemplo, desde SQL*Plus o SQLDeveloper mediante la orden EXECUTE:

```
EXECUTE  ver_cliente('ANTONIO') ;
Nombre:ANTONIO Nif: 9991 Domicilio:LAS ROZAS
Procedimiento PL/SQL terminado con éxito.
```

También podemos invocar al procedimiento desde un bloque PL/SQL de esta forma:

```
BEGIN
ver_cliente('ANTONIO’);
END;
Nombre:ANTONIO Nif: 9991 Domicilio:LAS ROZAS
```

Como en cualquier otro lenguaje, podemos insertar comentarios en cualquier parte del programa (los comentarios no se pueden anidar). Estos comentarios pueden ser:

- De línea con "--". Todo lo que le sigue en esa línea será considerado comentario.
- De varias líneas con "/*" <comentarios> "*/".

Los ejemplos que siguen son una primera aproximación al lenguaje PL/SQL. Para conocer todas las posibilidades de este lenguaje iremos paso a paso.

En el siguiente procedimiento se visualiza el precio de un producto cuyo código se pasa como parámetro

```
CREATE OR REPLACE PROCEDURE ver_precio(p_cod_producto_i  NUMBER)
IS
   v_precio NUMBER;
BEGIN
    SELECT precio_uni INTO v_precio
    FROM productos
    WHERE COD_PRODUCTO = p_cod_producto_i;
    DBMS_OUTPUT.PUT_LINE(v_precio) ;
```

```
END;

Procedimiento creado.

SQL> EXECUTE ver_precio(7);
20000
Procedimiento PL/SQL terminado con éxito.
```

Procedimiento que modifique el precio de un producto pasándole el código del producto y el nuevo precio. El procedimiento comprobará que la variación de precio no supere el 20 por 100:

```
CREATE OR REPLACE PROCEDURE modificar_precio_producto
(p_codigoprod_i  NUMBER, p_nuevoprecio_i  NUMBER)
AS
precioant NUMBER(5);
BEGIN
SELECT precio_uni INTO precioant FROM productos
WHERE cod_producto = p_codigoprod_i;
IF (precioant * 0.20) > ABS(precioant - p_nuevoprecio_i)THEN
    UPDATE productos SET precio_uni = p_nuevoprecio_i
    WHERE cod_producto = p_codigoprod_i;
ELSE
DBMS_OUTPUT.PUT_LINE('Error, modificación superior al 20%');
END IF;
EXCEPTION
WHEN NO_DATA_FOUND THEN
DBMS_OUTPUT.PUT_LINE('No encontrado el producto ' || p_codigoprod_i);
END modificar_precio_producto;
```

Ejemplos de ejecución:

```
EXECUTE MODIFICAR_PRECIO_PRODUCTO(3, 8000)
Bloque anónimo terminado.
```

Compruebo la modificación

```
SELECT PRECIO_UNI FROM PRODUCTOS WHERE COD_PRODUCTO=3;
```

```
EXECUTE MODIFICAR_PRECIO_PRODUCTO(3,10000 )
Error, modificación superior al 20% Procedimiento PL/SQL terminado con éxito.
```

```
SELECT PRECIO_UNI FROM PRODUCTOS WHERE COD_PRODUCTO=3;
```

Observamos que en el segundo caso no se ha producido la modificación deseada. Aun así, el procedimiento ha terminado con éxito, ya que no se ha producido ningún error no tratado.

2. Fundamentos del lenguaje PL/SQL

2.1 Tipos de datos básicos. Los más usados

- CARÁCTER. Igual que para SQL.
  - CHAR(n): array de n caracteres, máximo 2000 bytes. Si no especificamos longitud sería 1.

- VARCHAR2 (n): para almacenar cadenas de longitud variable con un máximo de 32760 bytes.
- LONG(n): Array de caracteres con un máximo de 32760 bytes

- NUMÉRICO
  - NUMBER(P,E). PL/SQL dispone de subtipos de NUMBER que se utilizan por compatibilidad y/o para establecer restricciones. Son: DECIMAL, NUMERIC, INTEGER, REAL, SMALLINT, etc.
  - BINARY_INTEGER. Es un tipo numérico entero que se almacena en memoria en formato binario para facilitar los cálculos. Se utiliza en contadores, índices, etc. Existen los subtipos: NATURAL y POSITIVE
  - PLS_INTEGER. Similar a BINARY_INTEGER pero la representación interna es distinta. Sus ventajas son:
    - Es más rápido
    - Si se da desbordamiento en el cálculo se produce un error y se levanta la excepción correspondiente, lo cual no ocurre en BYNARY_INTEGER.

- BOOLEANO.
  - Almacena valores TRUE, FALSE y NULL.

- FECHA Y HORA
  - DATE Almacena fechas. El formato estándar es 'dd-mm-aa'. También almacena la hora.
  - TIMESTAMP. Es una extensión del tipo DATE que guarda además fracciones de segundo.

- OTROS TIPOS ESCALARES
  - RAW(n). Almacena datos binarios de longitud fija (máximo de 2000 bytes). Se utiliza para almacenar cadenas de caracteres evitando las conversiones entre conjuntos de caracteres que realiza Oracle.
  - LONG RAW. Almacena datos binarios en longitud fija (máximo de 32760 bytes) evitando conversiones entre conjuntos de caracteres
  - ROWID. Almacena identificadores de fila.

En algunos casos no existe una equivalencia exacta entre los tipos de PL/SQL y los mismos tipos de las columnas de Oracle, especialmente en lo relativo a las longitudes máximas que pueden almacenar.

Se pueden hacer conversiones implícitas de tipos (carácter-numérico y carácter-fecha), pero es preferible hacerlo de forma explícita utilizando las funciones correspondientes: **TO_CHAR, TO_NUMBER Y TO_DATE**.

PL/SQL dispone también de otros más complejos:

- Tipos compuestos: registros, tablas y arrays. Se verán más tarde
- Referencias: el equivalente a los punteros de C
- LOB (objetos de gran tamaño) (BLOB (objeto binario con capacidad de 4 GB), CLOB (objeto carácter con una capacidad de 2 GB), BFILE (puntero a un fichero del Sistema Operativo)).
- Subtipos definidos por el usuario

2.2 Variables

2.2.1 Declaración e inicialización de variables

Todas las variables PL/SQL deben declararse en la sección correspondiente antes de su uso

Formato

<nombre_de_variable> <tipo>[NOT NULL] [{:= | DEFAULT }<VALOR>]

Cada variable una declaración distinta.

La opción NOT NULL fuerza a que la variable tenga un valor. Si se usa, deberá inicializarse la variable en la declaración con DEFAULT o con :=

Si no se inicializan las variables en PL/SQL se garantiza que su valor es NULL.

2.2.2 Uso de los atributos %TYPE y %ROWTYPE

En lugar de indicar explícitamente el tipo y la longitud de una variable, existe la posibilidad de utilizar los atributos %TYPE y %ROWTYPE para declarar variables que sean del mismo tipo que otros objetos ya definidos.

- **%TYPE** declara una variable del mismo tipo que otra, o que una columna de una tabla.

Ejemplos:

```
Total  importe%TYPE
```

Declara la variable *total* del mismo tipo que la variable importe que se habrá definido previamente

```
Nombre_moroso clientes.nombre%TYPE
```

Declara la variable *nombre_moroso* del mismo tipo que la columna nombre de la tabla clientes

- **%ROWTYPE** Declara una variable de registro cuyos campos se corresponden con las columnas de una tabla o vista de la base de datos.

Ejemplos:

```
Moroso clientes%ROWTYPE;
```

Declara una variable compuesta por campos que se corresponden con los de la tabla clientes

```
Moroso. Nombre
```

Hace referencia a la columna nombre de la tabla clientes

Al declarar una variable del mismo tipo que otro objeto usando los atributos **%TYPE** y **%ROWTYPE**, se hereda el tipo y la longitud, pero no los posibles atributos NOT NULL ni los valores por defecto que tuviese el objeto original.

2.2.3 Constantes

Formato:

```
<nombre_de_constante> CONSTANT  <tipo> := <valor>
```

Siempre se deberá asignar un valor en la declaración

2.2.4 Ámbito y visibilidad de las variables

El ámbito de una variable es el bloque en el que se declara y los bloques "hijos" de dicho bloque. La variable será local para el bloque en el que ha sido declarada y global para los bloques hijos de éste. Las variables declaradas en los bloques hijos no son accesibles desde el bloque padre.

```
DECLARE ---------------------------------------- BLOQUE  PADRE
        v1 CHAR;
BEGIN
        ….
        v1 := 1;
        DECLARE   ---------------------------------BLOQUE HIJO
```

```
                V2  CHAR;
        BEGIN
                v2 := 2;
                …
                v1 := v2;
                …
        END;    ------------------------------------FIN BLOQUE HIJO
        v2:= v1;    ---------------------------------Error: v2 es desconocida en este ámbito
END;    ---------------------------------------------FIN BLOQUE PADRE
```

Podemos observar que v1 es accesible para los dos bloque (es local al bloque padre y global para el bloque hijo), mientras que v2 solamente es accesible para el bloque hijo.

Las variables se crean al comienzo del bloque y dejan de existir una vez finalizada la ejecución del bloque en el que han sido declaradas.

En el caso de que un identificador local coincida con uno global, si no se indica más, se referencia el local. No obstante, se pueden utilizar etiquetas y cualificadores para deshacer ambigüedades.

```
<<padre>>       ------------------------etiqueta que identifica al bloque padre
DECLARE
        V CHAR;
BEGIN
        ….
        DECLARE
                V CHAR;
        BEGIN
                ….
                V:= padre.V;  ------------ el primero es el identificador local
                ….
        END;
END;
```

En caso de coincidencia los identificadores de columnas tienen precedencia sobre variables y parámetros formales; éstos a su vez, tienen precedencia sobre los nombres de tablas.

2.3 Operadores

- Asignación **:=**
- Lógico: **AND**,**OR** y **NOT**

AND	TRUE	FALSE	NULL
TRUE	TRUE	FALSE	NULL
FALSE	FALSE	FALSE	FALSE
NULL	NULL	FALSE	NULL

OR	TRUE	FALSE	NULL
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	NULL
NULL	TRUE	NULL	NULL

NOT	TRUE	FALSE	NULL
	FALSE	TRUE	NULL

- Concatenación **||**
- Comparación Igual que en SQL
- Aritméticos **+**, **-**, ***, **/**, ****.** Algunos de ellos (**+**,**-**) se pueden utilizar también con fechas

Orden de precedencia en los operadores

El orden de precedencia o prioridad de los operadores determina el orden de evaluación de los operandos de una expresión.

Prioridad	**Operador**
1	**, NOT
2	*, /
3	+,-,||
4	=, !=, <, >, <=, >=, IS NULL, LIKE, BETWEEN, IN
5	AND
6	OR

Las prioridades se pueden cambiar utilizando paréntesis

2.4 Funciones

En PL/SQL se pueden utilizar todas las funciones de SQL que se han estudiado anteriormente. No obstante, algunas como las de agrupamiento (**AVG, MIN, MAX, COUNT, SUM**) solamente se pueden usar dentro de cláusulas de selección (**SELECT**).

Se deben tener en cuenta dos aspectos:

- La función no modifica el valor de las variables o expresiones que se pasan como argumento, sino que devuelve un valor a partir de dicho argumento
- Si a una función se le pasa un valor nulo en la llamada, normalmente devolverá un valor nulo

Veamos un ejemplo de funciones: escribiremos un bloque PL/SQL que muestre la fecha y la hora con minutos y segundos:

```
BEGIN
    DBMS_OUTPUT.PUT_LINE(TO_CHAR(SYSDATE, 'DAY, DD MONTH " a las "
    YYYY:HH24:MI:SS'));
END;
LUNES    , 14 MARZO      a las    2016:10:01:35
Procedimiento PL/SQL terminado con éxito.
```

2.5 Estructuras de control

Como cualquier lenguaje de tercera generación, PL/SQL dispone de estructuras para controlar el flujo de ejecución de los programas.

La mayoría de las estructuras de control requieren evaluar una condición, que en PL/SQL puede dar tres resultados: **TRUE**, **FALSE** o **NULL**. Pues bien, a efectos de estas estructuras, el valor NULL es equivalente a FALSE, es decir, se considerará que se cumple la condición sólo si el resultado es TRUE. En caso contrario (FALSE o NULL), se considerará que no se cumple.

Alternativa simple IF <condicion> THEN *instrucciones;* …. END IF;	Si la condición se cumple, se ejecutan las instrucciones que siguen a la cláusula THEN
Alternativa doble IF <condicion> THEN instrucciones1; ….; ELSE instrucciones2; …..; END IF;	Si la condición se cumple, se ejecutan las instrucciones que siguen a la cláusula THEN. En caso contrario, se ejecutarán las instrucciones que siguen a la cláusula ELSE.
Alternativa múltiple IF <condicion1> THEN instrucciones 1; ….; ELSIF <condicion2> THEN instrucciones; ….. ELSIF <condicion3> THEN instrucciones; ……; [ELSE instrucciones; ……;] END IF;	Evalúa, comenzando desde el principio, cada condición, hasta encontrar alguna condición que se cumpla, en cuyo caso ejecutará las instrucciones que siguen a la cláusula THEN correspondiente. La cláusula ELSE es opcional; en caso de que se utilice, se ejecuta cuando no se ha cumplido ninguna de las condiciones anteriores.
Alternativa múltiple con CASE de comprobación CASE expresión WHEN <valorcomprobac1> THEN Instruciones1; WHEN <valorcomprobac2> THEN Instruciones2; WHEN <valorcomprobac3> THEN Instruciones3; ……. [ELSE Instruccionesotras;] END CASE;	Calcula el resultado de la expresión que sigue la cláusula CASE. A continuación, **comprueba** si el **valor** obtenido coincide con alguno de los valores especificados detrás de las cláusulas WHEN, en cuyo caso ejecutará la instrucción o instrucciones correspondientes. La cláusula ELSE es opcional, se ejecutará en el caso de que no se encuentre un valor coincidente en las cláusulas WHEN precedentes

Alternativa múltiple con CASE de <u>búsqueda</u> CASE WHEN <condicion1> THEN Instruciones1; WHEN <condicion2> THEN Instruciones2; WHEN … ELSE Instruciones4; END CASE;	**Evalúa** comenzando desde el principio, cada **condición**, hasta encontrar alguna condición que se cumpla, en cuyo caso ejecutará las instrucciones que siguen a la cláusula THEN correspondiente. La cláusula ELSE es opcional; en caso de que se utilice se ejecuta cuando no se ha cumplido ninguna de las condiciones anteriores.
Iterar... fin iterar salir si LOOP instrucciones; EXIT WHEN <condición>; instrucciones; ….; END LOOP;	Se trata de un bucle que se repetirá hasta encontrar la cláusula EXIT. Normalmente esta orden se encontrará en el formato indicado al lado, o bien como en el ejemplo siguiente: LOOP instrucciones; EXIT WHEN <condición>; instrucciones; ….; END LOOP;
Mientras WHILE <condicion> LOOP instrucciones; …..; END LOOP;	Es un bucle que se ejecutará mientras se cumpla la condición. Se evalúa la condición y, si se cumple, se ejecutarán las instrucciones del bucle.
Para FOR <variablecontrol> IN <valorInicio>..<valorFinal> LOOP instrucciones; …; END LOOP;	Donde *<variablecontrol>* es la variable de control del bucle que se declara de manera implícita como variable local al bucle de tipo BINARY_INTEGER. Esta variable tomará, en primer lugar, el valor especificado en la expresión numérica *<valorInicio>,* incrementándose en uno para cada nueva iteración hasta alcanzar el valor especificado en la expresión numérica *<valorFinal>.* Respecto a la variable de control, hay que tener en cuenta que no hay que declararla, que es local al bucle y no es accesible desde el exterior del bucle, ni siquiera en el mismo bloque; y que se puede usar dentro del bucle en una expresión, pero no se le pueden asignar valores. El incremento siempre es una unidad, pero puede ser negativo utilizando la opción REVERSE: FOR <variable> IN REVERSE <valorFinal>..<valorlnicio> LOOP instrucciones ; …..; END LOOP; En este caso, comenzará por el valor especificado en segundo lugar e irá restando una unidad en cada iteración.

Ejemplo:

```
BEGIN
  FOR i IN REVERSE 1..3 LOOP DBMS_OUTPUT.PUT_LINE(i) ;
       END LOOP;
END;
```

```
/
3
2
1

Procedimiento PL/SQL terminado con éxito.
```

Cuando empleamos la opción REVERSE, comenzará por el segundo valor hasta tomar un valor que sea igual o menor que el especificado en primer lugar. El bloque que se muestra **a continuación tiene un error de diseño**, ya que no llega siquiera a entrar en el bucle:

```
BEGIN
FOR i  IN  REVERSE   5. . 1   LOOP
    DBMS_OUTPUT.PUT_LINE (i) ;
        END LOOP;
END;
/
Procedimiento PL/SQL terminado con éxito.
```

Podemos indicar los valores mínimo y máximo mediante expresiones:

```
DECLARE
  Num1 INTEGER;
  Num2 INTEGER;
  ….
BEGIN
  ….
  FOR x IN Num1. .Num2 LOOP
        ….
  END LOOP;
  ….
END;
```

Si definimos una variable previamente e intentamos usarla como variable de control, la estructura creará la suya propia como **local**, quedando la nuestra como **global** en el bucle:

```
«ppal»
DECLARE
i   INTEGER;
BEGIN
        ….
FOR i IN 1. .10 LOOP
        …….       /* cualquier referencia a i será entendida como a la variable local al
                  bucle. Si quisiéramos referirnos a la otra lo debemos hacer como ppal.i */
 END LOOP;
…..                /*La variable local del bucle ya no existe aquí */
 END ppal;
```

Consideraciones para el manejo de bucles

Los bucles se pueden etiquetar para conseguir mayor legibilidad:

```
« mibucle»
LOOP
   <secuencia_de_instrucciones>;
END LOOP mibucle;
```

También se pueden conseguir otras cosas con las etiquetas:

```
« mibucle»
LOOP
….
    LOOP
      ….
        EXIT  mibucle  WHEN ….          /*  sale de ambos bucles  */
    END LOOP
END LOOP mibucle;
```

2.6 Subprogramas: procedimientos y funciones

2.6.1 Procedimientos

Estructura general:

```
PROCEDURE  <nombreprocedimiento> [(lista de parámetros)]   ---- Cabecera o especificación
IS
        <declaraciones>;
BEGIN
        <declaraciones>;                    Cuerpo     /* IS o AS    */
EXCEPTION
         <declaraciones>;
END  [<nombreprocedimiento>];
```

En la lista de parámetros encuentra la declaración de cada uno de los parámetros separados por comas. El formato de cada declaración es:

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

IN es para parámetro de entrada

OUT es para parámetro de salida

IN OUT es para parámetro de entrada/salida

Al indicar los parámetros debemos especificar el tipo, pero no el tamaño. En el caso de que no tenga parámetros no se pondrán los paréntesis.

Las declaraciones se hacen después de IS, que equivale al DECLARE. En este caso, sí se deberá indicar la longitud de las variables

Para crear un procedimiento lo haremos utilizando el comando siguiente:

```
CREATE [OR REPLACE] PROCEDURE <nombreprocedimiento>
```

2.6.2 Funciones

Las funciones tienen una estructura y una funcionalidad similar a los procedimientos pero, a diferencia de aquellos, éstas devuelven un valor:

```
CREATE OR REPLACE  FUNCTION <nombredefunción>  [(<lista de parámetros>)]   /*
Cabecera*/
```

```
RETURN <tipo de valor devuelto>
IS
        <declaraciones>;
BEGIN                                   Cuerpo
        <instrucciones>;
        RETURN <expresión>;
        ….
EXCEPTION
        <excepciones>;
END [<nombredefunción>];
```

Los parámetros tienen la misma sintaxis en las funciones que en los procedimientos, y es válido todo lo indicado para ellos.

La cláusula RETURN de la cabecera especifica el tipo del valor que retorna la función, asignando el valor devuelto por la función al identificador de la misma en el punto de la llamada

Para crear una función:

```
CREATE OR REPLACE FUNCTION <nombredefunción>
```

El formato de llamada a una función consiste en utilizarla como parte de una expresión

```
<variable> := <nombredefunción>(parámetros);
```

Se pueden invocar funciones en comandos SQL, pero para hacerlo desde PL/ SQL se tiene que ejecutar:

```
BEGIN
        DBMS_OUTPUT.PUT_LINE(encontrar_num_empleado('MUÑOZ'));
END;
7934

Procedimiento PL/SQL terminado con éxito.
```

Una función puede tener varios RETURN

2.6.3 Parámetros

Los subprogramas utilizan parámetros para pasar y recibir información

Hay dos clases:

- Parámetros actuales o reales: Son las variables o expresiones indicadas en la llamada a un subprograma.
- Parámetros formales: Son variables declaradas en la especificación del subprograma.

Si es necesario PL/SQL hará la conversión automática de tipos; sin embargo, los tipos de los parámetros actuales y los correspondientes parámetros formales deben ser compatibles.

Podemos hacer el paso de parámetros utilizando las notaciones posicional, nominal o mixta

- Notación posicional: El compilador asocia los parámetros actuales a los formales basándose en su posición.
- Notación nominal: El símbolo => después del parámetro actual y antes del nombre del formal indica al compilador la correspondencia.
- Notación mixta: Consiste en usar ambas notaciones con la restricción de que la notación posicional debe preceder a la nominal.

Por ejemplo:

Dada la siguiente especificación del procedimiento ges_depart

```
PROCEDURE ges_depart (
        p_numdepart  INTEGER,
        p_localidad      VARCHAR)
IS ……
```

Distintas llamadas al procedimiento:

```
DECLARE
    num_dep        INTEGER,
    local          VARCHAR(14)
BEGIN
    ….
    Ges_depart (num_dep, local);            /*  posicional  */
    Ges_depart (p_numdepart  =>num_dep, p_localidad =>local);  /*  nominal  */
    Ges_depart (p_localidad =>local, p_numdepart => num_dep);  /*  nominal  */
    Ges_depart (num_dep, p_localidad =>local);
    ….
END;
```

Valores por defecto en el paso de parámetros (modo IN):

Los parámetros en modo IN (todos los que hemos estudiado hasta este momento) se pueden inicializar con valores por omisión, es decir, indicando al subprograma que en el caso de que no se pase el parámetro correspondiente asuma un valor por defecto. Esto se hace con la opción

```
DEFAULT <valor>
```

o bien

```
n:=<valor>
```

Tipos de parámetros

- **IN**. Permite pasar valores a un subprograma
  - Dentro del subprograma, el parámetro actúa como una constante, es decir, no se le puede asignar ningún valor.
  - El parámetro actual puede ser una variable, constante, literal o expresión.
- **OUT**. Permite devolver valores al bloque que llamó al subprograma
  - Dentro del subprograma, el parámetro actúa como una variable no inicializada.
  - No puede intervenir en ninguna expresión, salvo para tomar un valor.
  - El parámetro actual debe ser una variable.
- **IN OUT.** Permite pasar un valor inicial y devolver un valor actualizado
  - Dentro del subprograma actúa como una variable inicializada.
  - Puede intervenir en otras expresiones y puede tomar nuevos valores.
  - El parámetro actual debe ser una variable.

Los parámetros IN se sitúan siempre a la derecha del operador de asignación, los parámetros OUT siempre a la izquierda, y los IN OUT pueden situarse tanto a la derecha como a la izquierda.

Cuando un parámetro se pasa en modo OUT, en el caso de que se produzca una salida del programa por una excepción no tratada, el parámetro actual correspondiente queda sin ningún valor.

Antes de ejecutar un subprograma almacenado, Oracle marca un punto de salvaguarda implícito, de forma que si el subprograma falla durante la ejecución, se desharán todos los cambios realizados por él

Para llamar a un subprograma almacenado haremos lo siguiente:

- Desde otro subprograma:

```
nombresubprograma(lista_de_parámetros);
```

- Desde otro programa escrito en otro lenguaje (en este caso, los parámetros deben ser variables host):

```
EXEC SQL EXECUTE
  BEGIN
    Nombresubprograma(:parámetro1, :parámetro2,…);
  END;
END-EXEC;
```

- Desde SQL*PLUS, SQLDeveloper y otras herramientas ORACLE

```
EXECUTE nombresubprograma(lista_de_parámetros);
-- o también
BEGIN
   nombresubprograma(lista_de_parámetros);
END;
```

Para borrar un subprograma, igual que para eliminar otros objetos, se usa la orden DROP seguida del tipo de subprograma (PROCEDURE o FUNCTION)

```
DROP {PROCEDURE | FUNCTION} nombresubprograma;
```

2.6.4 Variables de enlace

Una variable de enlace es una variable que se declara en un entorno host, se utilizan para transferir valores de ejecución (numéricos o de carácter) a programas PL/SQL

1º Creación de la variable de enlace.

```
VARIABLE  g_salario  NUMBER
```

2º Utilización en PL/SQL. Se emplea ":" delante de la variable

```
BEGIN
   :g_salario:=20000;
END;
/
```

3º Desde SQL se puede conocer el valor de dicha variable

```
PRINT g_salario;
```

4º También se puede conocer su valor desde el PL/SQL

```
 BEGIN
:g_salario:=200;
 DBMS_OUTPUT.PUT_LINE ( :g_salario);
END;
/
```

```
200
```

O también

```
SELECT  :g_salario  FROM  DUAL;
```

Utilización de las variables de enlace

Los procedimientos con parámetros **IN** se pueden probar con **EXECUTE**

Si un procedimiento tiene parámetros **OUT**, por ejemplo:

```
CREATE OR REPLACE PROCEDURE param_out
(p_id IN emple.emp_no%type,
        p_apellido_o OUT emple.apellido%type)
IS
BEGIN
        SELECT apellido INTO p_apellido_o
        FROM  emple
        WHERE emp_no = p_id;
END  param_out;
/
SQL> START param_out
Procedimiento creado.
```

```
SQL> VARIABLE g_apellido varchar2(15);
SQL> EXECUTE param_out (7654,:g_apellido);
Procedimiento PL/SQL terminado correctamente.
SQL> PRINT g_apellido;
G_APELLIDO
--------------------------------
MARTIN
```

Si un procedimiento tiene parámetros IN OUT, por ejemplo en **SQLPlusw**:

```
CREATE OR REPLACE PROCEDURE  telefono
 (p_telefono_io IN OUT  VARCHAR2)
IS
BEGIN
  p_telefono_io := '(' || SUBSTR(p_telefono_io,1,3) ||
              ')' || SUBSTR(p_telefono_io,4,3) ||
              '-' || SUBSTR(p_telefono_io,7);
END telefono;
/
SAVE telefono;
Creado fichero telefono
```

Si trabajamos con **SQLDeveloper** no pondremos:
/
SAVE teléfono;

Pasaremos directamente a :
EXECUTE telefono (:g_telefono);

```
START telefono;
Procedimiento creado.  (Coincide fichero de órdenes con el nombre del procedimiento)
```

```
VARIABLE g_telefono VARCHAR2(12)
BEGIN :g_telefono := '976345678';  END;
/
```

```
Procedimiento PL/SQL terminado correctamente.
```

```
EXECUTE telefono (:g_telefono);
Procedimiento PL/SQL terminado correctamente.
```

```
PRINT g_telefono
G_TELEFONO
--------------------------------
(976)345-678
```

Utilización de variables Host con funciones

```
CREATE OR REPLACE FUNCTION  tan_ciento
   (p_valor_i  IN NUMBER)
   RETURN  NUMBER
 IS
 BEGIN
   RETURN (p_valor_i * 0.10);
 END tan_ciento;
 /
Función creada.
```

```
VARIABLE g_valor  number
EXECUTE :g_valor := tan_ciento(2000);
Procedimiento PL/SQL terminado correctamente.
```

```
PRINT g_valor
  G_VALOR
----------
      200
```

Se puede invocar una función en expresiones SQL

- Como columna de una SELECT

```
SELECT emp_no,apellido,salario,tan_ciento(salario)
 from emple
 WHERE emp_no = 7369
```

- Condiciones en cláusulas WHERE y HAVING
- Cláusulas CONNECT BY, START WITH, ORDER BY y GROUP BY
- Cláusulas VALUES de un comando INSERT
- Cláusulas SET de un comando UPDATE

Restricciones a las llamadas de funciones en expresiones SQL

- Una función de usuario tiene que ser una función almacenada.
- Acepta sólo parámetros de tipo IN con tipos de datos compatibles con SQL:
    - Los tipos de datos tienen que ser CHAR, VARCHAR2, DATE, NUMBER …
    - Los Tipos de datos no pueden ser PL/SQL como BOOLEAN, RECORD o TABLE
- El tipo devuelto debe ser compatible con SQL

- No pueden contener instrucciones DML
- Si una instrucción DML modifica una determinada tabla, en dicha instrucción no se puede invocar a una función que realice consultas sobre la misma tabla
- No pueden utilizar instrucciones de transacciones (COMMIT, ROLLBACK,...)
- La función no puede invocar a otra función que se salte alguna de las reglas anteriores.

Subprogramas almacenados

Los subprogramas (procedimientos y funciones) que hemos visto hasta ahora se pueden compilar independientemente y almacenar en la base de datos Oracle.

Cuando creamos procedimientos o funciones almacenados utilizando los comandos CREATE PROCEDURE O CREATE FUNCTION, Oracle automáticamente compila el código fuente, genera el código objeto y los guarda en el diccionario de datos. De este modo quedan disponibles para su utilización.

Los programas almacenados tienen dos estados: disponible (valid) y no disponible (invalid). Si alguno de los objetos referenciados por el programa ha sido borrado o alterado desde la última compilación del programa quedará en situación de "no disponible" y se compilará de nuevo automáticamente en la próxima llamada. Al compilar de nuevo, Oracle determina si hay que compilar algún otro subprograma referido por el actual, y se puede producir una cascada de compilaciones.

Estos estados se pueden comprobar en la vista **USER_OBJECTS**:

```
DESCRIBE SYS.USER_OBJECTS;

Name                          Null?    Type
 ------------------------------           -------- ----
 OBJECT_NAME                              VARCHAR2(128)
 SUBOBJECT_NAME                      VARCHAR2(30)
 OBJECT_ID                     NUMBER
 DATA_OBJECT_ID                           NUMBER
 OBJECT_TYPE                              VARCHAR2(15)
 CREATED                       DATE
 LAST_DDL_TIME                            DATE
 TIMESTAMP                                VARCHAR2(19)
 STATUS                        VARCHAR2(7)
 TEMPORARY                                VARCHAR2(1)
 GENERATED                                VARCHAR2(1)
```

```
SELECT OBJECT_NAME, OBJECT_TYPE, STATUS
  FROM USER_OBJECTS
  WHERE OBJECT_TYPE = 'PROCEDURE';
```

```
OBJECT_NAME                              OBJECT_TYPE    STATUS
 ----------------------------------------    --------------- -------
 CAMBIAR_DIVISAS                          PROCEDURE      VALID
 CAMBIAR_OFICIO                           PROCEDURE      VALID
 CAM_OFI_APE                              PROCEDURE      VALID
 PROBAR_CAMBIO_DIVISAS                    PROCEDURE      VALID
```

También se puede encontrar el código fuente en la vista **USER_SOURCE**

```
desc sys.user_source;
```

```
 Nombre Nulo Tipo
------        ---- --------------
NAME              VARCHAR2(30)
TYPE             VARCHAR2(12)
LINE             NUMBER
TEXT              VARCHAR2(4000)
```

```
SELECT LINE, SUBSTR(TEXT,1,60) FROM  USER_SOURCE
  WHERE NAME = 'CAMBIAR_OFICIO';
```

```
  LINE      SUBSTR(TEXT,1,60)
--------- ------------------------------------------------------------
      1 PROCEDURE  cambiar_oficio(
      2                    num_empleado    NUMBER,
      3                    nuevo_oficio    VARCHAR2)
      4 AS
      5                    anterior_oficio emple.oficio%TYPE;
      6 BEGIN
      7                    SELECT oficio INTO anterior_oficio FROM emple
      8                    WHERE emp_no = num_empleado;
      9                    UPDATE emple SET oficio = nuevo_oficio
     10                    WHERE emp_no = num_empleado;
     11                    DBMS_OUTPUT.PUT_LINE(num_empleado || '* oficio anterior: '
     12                    || anterior_oficio || '* oficio nuevo: ' || nuevo_oficio);
     13          END cambiar_oficio;
13 filas seleccionadas.
```

Para volver a compilar un subprograma almacenado en la base de datos se emplea la orden ALTER, indicado PROCEDURE o FUNCTION, según el tipo de subprograma

```
ALTER {PROCEDURE | FUNCTION } nombresubprograma  COMPILE;
```

2.6.5 Subprogramas locales

```
CREATE OR REPLACE  PROCEDURE pr_ejem1  /* programa que contiene el subprograma local
*/
        ….
AS
        ….        /* lista de declaraciones: variables, etc.  */
            PROCEDURE/FUNCTION  sprloc1      /* comienza el subprograma local */
                    …..                      /* lista de parámetros del subprograma local  */
            IS
                    …..                      /*  declaraciones  locales  al  subprograma  local    */
        Local
            BEGIN
                    …..                      /* instrucciones del subprograma local  */
            END;
BEGIN
        …..
        sprloc1;                             /* llamada al subprograma local */
        …..
END;
```

Estos subprogramas reciben el nombre de subprogramas locales y tienen las siguientes particularidades:

- Se declaran al final de la sección declarativa de otro subprograma o bloque.
- Se les aplica las mismas reglas de ámbito y visibilidad que a las variables declaradas en el mismo bloque.
- Se utilizará este tipo de subprogramas cuando no se contemple su reutilización por otros subprogramas (distintos a aquél en el que se declaran).
- En el caso de subprogramas locales con referencias cruzadas o de subprogramas mutuamente recursivos, hay que realizar declaraciones anticipadas, tal como se explica a continuación.

PL/SQL necesita que todos los identificadores, incluidos los subprogramas, estén declarados antes de usarlos.

```
DECLARE  /*el bloque principal es anónimo*/
    PROCEDURE subprograma2 (.....);   → declaración anticipada
    PROCEDURE subprograma1 (.....)
       IS
                ….
BEGIN
       ….        /*a continuación se llama a subprograma2*/
                subprograma2 (……); /*se desarrolla a continuación*/
                ……
END;
PROCEDURE subprograma2 (.....)  IS …
BEGIN
       ….
END;
    BEGIN
       …
    END;
```

Esta Técnica permite definir subprogramas en el orden que queramos (alfabético, etc.); incluyendo programas mutuamente recursivos

2.6.6 Recursividad

PL/SQL implementa, al igual que la mayoría de los lenguajes de programación, la posibilidad de escribir subprogramas recursivos:

No se deben usar algoritmos recursivos en conjunción con cursores FOR…LOOP, ya que se puede exceder el número máximo de cursores abiertos. Siempre se pueden sustituir las estructuras recursivas por bucles