Conceptos básicos de programación con PL/SQL

---

1. Bloques PL/SQL

1.1 Introducción

SQL es un lenguaje de consulta para los sistemas de bases de datos relaciónales, pero que no posee la potencia de los lenguajes de programación. Cuando se desea realizar una aplicación completa para el manejo de una base de datos relacional, resulta necesario utilizar alguna herramienta que soporte la capacidad de consulta del SQL y la versatilidad de los lenguajes de programación tradicionales. PL/SQL es el lenguaje de programación que proporciona Oracle para extender el SQL estándar con otro tipo de instrucciones.

En PL/SQL los comentarios de una sola línea comienzan con '--', los de varias líneas comienzan con /* y terminan con */. Para poder visualizar valores desde dentro de un bloque PL/SQL se debe ejecutar la instrucción `dbms_output.put_line(dato)`. Previamente hay que habilitar la visualización en el entorno de SQL*Plus con la instrucción `SET SERVEROUTPUT ON`. El comando `PRINT` sólo permite visualizar variables externas, las cuales sólo se pueden utilizar en bloques PL/SQL anónimos (ni en procedimientos, ni en funciones).

1.2 Estructura de un Bloque

Los bloques PL/SQL presentan una estructura específica compuesta de tres partes bien diferenciadas:

- La sección declarativa en donde se declaran todas las constantes y variables que se van a utilizar en la ejecución del bloque.
- La sección de ejecución que incluye las instrucciones a ejecutar en el bloque PL/SQL.
- La sección de excepciones en donde se definen los manejadores de errores que soportará el bloque PL/SQL.

Cada una de las partes anteriores se delimita por una palabra reservada, de modo que un bloque PL/SQL se puede representar como sigue:

```
DECLARE
/*      Parte Declarativa       */
BEGIN
/*      Parte de Ejecución      */
EXCEPTIONS
/*      Parte de Excepciones    */
END;
```

De las anteriores, únicamente la sección de ejecución es obligatoria, que quedaría delimitada entre las cláusulas `BEGIN` y `END`. Veamos un ejemplo de bloque PL/SQL muy genérico.

```
DECLARE
   nombre_variable VARCHAR2(5);
   nombre_excepcion EXCEPTION;
BEGIN
   SELECT nombre_columna
   INTO nombre_variable
   FROM nombre_tabla;
EXCEPTIONS
   WHEN nombre_excepcion THEN ...
END;
```

1.2.1 Sección de Declaración de Variables

En esta parte se declaran todos los tipos de datos, las constantes y variables utilizadas en el bloque de ejecución. También se declaran cursores, de gran utilidad para la consulta de datos, y excepciones definidas por el usuario. Es conveniente que el nombre de una variable comience por la letra v, el de una constante por c, el de un tipo por t, el de un cursor por cur, y así sucesivamente. Las variables externas se declaran en SQL*Plus y se escriben precedidas de ':'.

CONSTANTES Y VARIABLES

```
nombre_variable [CONSTANT] tipo [NOT NULL] [:= valor_inicial];
```

- Donde `tipo` es: `tipo_escalar | identif%TYPE | identificador%ROWTYPE`
- Donde `tipo_escalar`: `NUMBER | DATE | CHAR | VARCHAR | BOOLEAN`
- La cláusula `CONSTANT` indica la definición de una constante cuyo valor no puede ser modificado. Se debe incluir la inicialización de la constante en su declaración.
- La cláusula `NOT NULL` impide que a una variable se le asigne el valor nulo, y por tanto debe inicializarse a un valor diferente de `NULL`.
- Las variables que no son inicializadas toman el valor inicial `NULL`.
- La inicialización puede incluir cualquier expresión legal de PL/SQL, que lógicamente debe corresponder con el tipo del identificador definido.
- Los tipos escalares incluyen los definidos en SQL más los tipos `VARCHAR` y `BOOLEAN`. Este último puede tomar los valores `TRUE`, `FALSE` y `NULL`, y se suele utilizar para almacenar el resultado de alguna operación lógica. Por su parte, `VARCHAR` es un sinónimo de `CHAR`, siendo más conveniente la utilización del tipo `CHAR`.
- También es posible definir el tipo de una variable o constante, dependiendo del tipo de otro identificador, mediante la utilización de las cláusulas `%TYPE` y `%ROWTYPE`. Mediante la primera opción se define una variable o constante escalar, y con la segunda se define una variable fila, donde `identificador` puede ser otra variable fila o una tabla.

Ejemplos:

```
DECLARE
   v_location VARCHAR2(15) := 'Granada';
   c_comm CONSTANT NUMBER(3) := 160;
   v_nombre tabla_empleados.nombre%TYPE;
```

CURSORES

El resultado de una consulta puede almacenarse en variables, en las que se almacenan cada una de las tuplas del resultado, o bien en una variable de tupla que sea compatible con el resultado de la consulta. Si aparece más de una fila como resultado de una consulta, resulta conveniente la utilización de cursores que permiten recorrer todas sus filas.

```
CURSOR nombre_cursor [parámetros] IS consulta_SQL;
```

Ejemplos:

```
DECLARE
   CURSOR emp_cursor IS SELECT empno,ename FROM empleados;
```

Los parámetros de un cursor se pueden utilizar para definir variables con valores de entrada que determinen el resultado de cada ejecución de la consulta SQL asociada. En la sección 1.4 hay un ejemplo.

EXCEPCIONES

En PL/SQL existe un conjunto de excepciones predefinidas que informan de los errores producidos en la ejecución de las sentencias SQL por parte del sistema de gestión de bases de datos. Además de éstas, el programador puede definir excepciones de uso específico, cuyo control es enteramente gestionado por él. La manera de definirlas es como sigue:

```
nombre_excepción EXCEPTION;
```

Las excepciones no son variables sino que su utilización debe realizarse mediante sentencias específicas de PL/SQL. Tampoco pueden ser utilizadas como argumentos en funciones ni procedimientos.

Ejemplos:

```
DECLARE
   demasiados_empleados EXCEPTION;
```

1.2.2 Sección de Instrucciones Ejecutables

Como todo lenguaje de programación, en PL/SQL se pueden distinguir tres clases de instrucciones:

- Instrucciones de asignación
- Instrucciones de control de flujo
- Bucles

INSTRUCCIONES DE ASIGNACION

```
variable_objetivo := expresión_PL/SQL;
```

Las expresiones PL/SQL pueden incluir literales, variables y constantes definidas en el bloque, así como funciones aplicadas sobre literales, constantes y variables. Los literales son similares a los utilizados en SQL, es decir,

- Las cadenas de caracteres se delimitan por la comilla simple.
- Los números reales pueden especificarse tanto en formato decimal como científico.

- Operadores sobre Números: +, -, *, /, ** (exponencial), MOD (resto).
- Operadores sobre cadenas: || (concatenación).
- Operadores lógicos: AND, OR, NOT.
- Operadores sobre cursores: %ROWCOUNT, %NOTFOUND, %FOUND, %ISOPEN.

Los valores lógicos aparecen como resultado de alguna comparación o verificación de valor. En PL/SQL se pueden utilizar los comparadores definidos en SQL y los anteriormente comentados sobre cursores,

- Comparadores clásicos: <, <=, =, !=, ^=, <, =
- Comparadores SQL: [NOT] LIKE, IS [NOT] NULL, [NOT] BETWEEN..AND.., [NOT] IN.

Las funciones definidas en SQL también aparecen en PL/SQL, además existen funciones propias del lenguaje:

- Funciones sobre cadenas de caracteres: ASCII, CHR, INITCAP, INSTR, LENGTH, LOWER, LPAD, LTRIM, REPLACE, RPAD, RTRIM, SOUNDEX, SUBSTR, TRANSLATE, UPPER.
- Funciones numéricas: ABS, CEIL, FLOOR, MOD, POWER, ROUND, SIGN, SQRT, TRUNC.
- Funciones sobre fechas: ADD_MONTHS, LAST_DAY, MONTHS_BETWEEN, NEW_TIME, NEXT_DAY, ROUND, SYSDATE, TRUNC.
- Funciones de conversión: TO_CHAR, TO_DATE, TO_NUMBER.
- Funciones de control de errores: SQLCODE, SQLERRM.
- Funciones varias: UID, USER, DECODE, GREATEST, LEAST, NVL, USERENV.

Por lo que respecta a las asignaciones asociadas a las sentencias SQL, existen dos alternativas, asignar el resultado a una lista de variables o a un cursor. Si se utiliza una sentencia SELECT para realizar una asignación a una lista de variables, la consulta asociada sólo debe dar como resultado una única fila, ya que en caso contrario se genera una excepción. Por esta razón, cuando no se conoce a priori el número de filas del resultado, resulta más conveniente utilizar cursores.

Para asignar el resultado de una sentencia SELECT a una lista de variables se utiliza la sintaxis:

```
SELECT lista_select INTO lista_variables FROM ... WHERE...;
```

```
DECLARE
   v_numdep NUMBER(2);
   v_local VERCHAR2(15);
BEGIN
   SELECT numdep, local INTO v_numdep, v_local
   FROM departamentos
   WHERE nombre='Informática'; --Seguro que sólo devuelve una fila
END;
```

El número de variables escalares en la lista de variables debe corresponder con el número de atributos del SELECT o cursor asociado. La lista de variables puede sustituirse por una variable fila del tipo correspondiente. Veamos como utilizar los cursores para tratar una a una las filas resultantes de un SELECT:

```
DECLARE
   CURSOR cur_emp IS SELECT num_empleado, nomb_empleado FROM empleado;
   emp_registro cur_emp%ROWTYPE;
BEGIN
   OPEN cur_emp;
   LOOP
      FETCH cur_emp INTO emp_registro;
      EXIT WHEN cur_emp%NOTFOUND;
      ...
   END LOOP;
   CLOSE cur_emp;
END;
```

SENTENCIAS DE CONTROL DE FLUJO

En PL/SQL es posible ejecutar un bloque de instrucciones u otro en función del valor de alguna expresión lógica, mediante la utilización de la sentencia `IF`:

```
IF expresión_lógica THEN intrucciones_PL/SQL;
[ELSIF expresión_lógica THEN intrucciones_PL/SQL;]
[ELSE intrucciones_PL/SQL;]
END IF;
```

Tal y como aparece en la sintaxis de esta sentencia, se pueden presentar diferentes alternativas, aunque al menos deben presentarse las cláusulas `IF ... THEN ... END IF`.

BUCLES

Los bucles permiten repetir un número de veces un conjunto de instrucciones PL/SQL. En PL/SQL los bucles se identifican con la cláusula `LOOP`, pudiéndose presentar cuatro sintaxis diferentes,

```
LOOP intrucciones_PL/SQL END LOOP;
WHILE expresión_lógica LOOP intrucciones_PL/SQL END LOOP;
FOR control_numérico LOOP intrucciones_PL/SQL END LOOP;
FOR control_cursor LOOP intrucciones_PL/SQL END LOOP;
```

- donde `control_numérico` corresponde a:

```
índice IN [REVERSE] expr_entera .. expr_entera
```

- donde `control_cursor` corresponde a:

```
variable_fila IN cursor | variable_fila IN sentencia_SQL
```

La finalización del bucle básico (`LOOP`) debe forzarse mediante la inclusión de la sentencia `EXIT`, definida como:

```
EXIT [WHEN expresión_lógica];
```

Dicha instrucción también puede ser utilizada para finalizar los otros bucles, aunque su uso no resulta recomendable, ya que no se ajusta al estilo de programación estructurada.

La utilización de los bucles numéricos y sobre cursores presenta efectos laterales de gran interés:

- El índice asociado a un bucle numérico no necesita declararse, sino que se declara de modo implícito. En el caso de declararse otra variable con el mismo nombre, ambas variables serían diferentes. Además, al final del bucle aparece un incremento, o decremento, implícito del índice.

- En un bucle sobre cursores también se declara de modo implícito la variable fila

asociada. Asimismo, se realiza un `OPEN` del cursor al entrar en el bucle y un `CLOSE` al salir, aunque la salida se produjera mediante la utilización de la sentencia `EXIT`, y se produce un `FETCH` implícito en cada iteración del bucle.

```
DECLARE
   CURSOR cur_emp IS
     SELECT num_empleado, nomb_empleado FROM empleado;
BEGIN
   FOR emp_registro IN cur_emp LOOP
      -- Apertura y declaración del cursor ímplitamente
      IF emp_registro.edad>30 THEN ...
   END LOOP;
   -- Cierre ímplicito del cursor
END;
```

1.2.3 Sección de Excepciones

El manejo de excepciones es muy importante en el diálogo con los sistemas de gestión de bases de datos, ya que permite responder ante cualquier problema que pueda ocurrir en la ejecución de cualquier operación. Por defecto, la ejecución de la aplicación finaliza al presentarse algún error grave, pero con la definición de manejadores específicos es posible realizar una serie de acciones y continuar la ejecución de la aplicación.

```
WHEN excepciones THEN instrucciones_PL/SQL;
```

- Donde `excepciones`: `excep_múltiples | OTHERS`
- Donde `excep_múltiples`: `excepción | excepción OR excep_múltiples`

La activación de una excepción la realiza el sistema, cuando se produce un error interno, o el programa mediante utilización de la sentencia `RAISE`:

```
RAISE excepción_válida;
```

donde se entiende por excepción válida a cualquier excepción predefinida o a una excepción definida en la parte declarativa del bloque PL/SQL. Cuando una excepción se activa, la ejecución continúa en la parte de manejadores de excepciones si la hubiera, en caso contrario se finaliza la ejecución del bloque y se mantiene la excepción por si tuviera que ser tratada en otro nivel. En la parte de manejadores, la ejecución se realiza del modo siguiente:

- Se busca, por orden secuencial, un manejador que corresponda con la excepción activada, si no se encuentra ningún manejador se finaliza el bloque, manteniendo activa la excepción.
- La cláusula `OTHERS` corresponde a todas las excepciones, por lo que resulta conveniente ponerla si se desea gestionar todas las excepciones, y poner su manejador el último para que los anteriores se puedan ejecutar.
- Una vez encontrado el manejador, se desactiva la excepción y se ejecutan las instrucciones asociadas, finalizándose a continuación el bloque.
- Si se deseara mantener activa la excepción, o se desea activar cualquier otra, es posible incluir la sentencia `RAISE` que finalizaría el bloque y activaría la excepción asociada.

Existe un conjunto de excepciones predefinidas por Oracle cuyos nombres aparecen a continuación agrupados por su funcionalidad.

- NO_DATA_FOUND, TOO_MANY_ROWS. Ocurren cuando un select no selecciona nada o selecciona varias filas cuando sólo se esperaba una.

- INVALID_NUMBER, VALUE_ERROR, ZERO_DIVIDE, DUP_VAL_ON_INDEX. Las tres primeras situaciones se producen por operaciones invalidas de tratamiento de números, y la ultima cuando se intenta insertar una clave primaria duplicada.

- CURSOR_ALREADY_OPEN, INVALID_CURSOR. La primera situación ocurre al intentar abrir un cursor ya abierto, y la segunda al intentar hacer una operación invalida sobre un cursor

- PROGRAM_ERROR, STORAGE_ERROR, TIMEOUT_ON_RESOURCE. Detectan errores de almacenamiento o de ejecución.

Estas excepciones pueden aparecer incluso en la parte de declaración, si se intenta inicializar una variable con un valor no permitido.

```
DECLARE
      e_hay_emp EXCEPTION;
      v_depnum departamento.depnum%TYPE := 777;
BEGIN
      IF (SELECT COUNT(*) FROM empleados WHERE depnum = v_depnum)=0
            THEN DELETE FROM departamento WHERE depnum = v_depnum;
            ELSE RAISE e_hay_emp;
      END IF;
      COMMIT;
EXCEPTION
      WHEN e_hay_emp THEN
            RAISE_APPLICATION_ERROR(-20001, 'No se puede borrar el
            departamento ' || TO_CHAR(v_depnum)||' ya que tiene empleados.');
      WHEN OTHERS THEN
            DBMS_OUTPUT.PUT_LINE('abortado por error desconocido');
END;
```

1.3 Anidamiento de Bloques

En algunos casos puede resultar interesante realizar un anidamiento de bloques PL/SQL, es decir, definir un bloque PL/SQL dentro de la parte de instrucciones de otro bloque PL/SQL. En estos casos, se debe estudiar el ámbito de las declaraciones definidas:

- Todas las constantes, variables y excepciones definidas en el bloque externo son visibles en el bloque interno, pero no al revés.

- Si existiera conflicto de nombres entre las declaraciones del bloque externo e interno, en cada bloque se toma la declaración local.

El bloque interno se considera como una única instrucción del bloque externo, que finaliza como cualquier otro bloque PL/SQL, es decir, por la finalización de su parte de instrucciones o por la activación y gestión de una excepción, si fuera el caso. Si el bloque interno finaliza manteniendo activa una excepción, se ejecuta el manejador de excepciones del bloque externo para su gestión, como si la excepción se hubiera producido en cualquier otra instrucción. De este modo, si se desea hacer frente a excepciones que pudieran aparecer en la ejecución de una sentencia SQL sin finalizar el bloque, la solución consiste en incluir la sentencia en un sub-bloque PL/SQL que gestione la excepción.

1.4 Ejemplo

En este ejemplo se supone que existe una tabla adicional en la base de datos de prácticas, denominada BALANCES, en la que se almacena el importe de las facturas al final de cada mes. Dicha tabla contiene dos columnas, la fecha en el que se ha realizado el balance y el importe total de las facturas del mes correspondiente.

En el siguiente ejemplo se utiliza un cursor con un parámetro, lo que indica que se puede ejecutar para diferentes valores de dicho parámetro. En este caso el parámetro indica el mes de las facturas de las que se va a calcular el balance.

```
DECLARE
  v_importe_mes NUMBER(10, 2) := 0;
  v_precio_fac  NUMBER(10, 2) := 0;
  v_iva_fac     NUMBER(4, 2) := 0;
  v_dto_fac     NUMBER(4, 2) := 0;
  v_mes_ant     CONSTANT DATE := ADD_MONTHS (SYSDATE, -1);

  CURSOR cur_fact_mes (v_mes DATE) IS
    SELECT codfac, iva, dto
    FROM facturas
    WHERE TO_CHAR (fecha, 'MMYY') = TO_CHAR (v_mes, 'MMYY');

BEGIN

FOR v_dat_fac IN cur_fact_mes (v_mes_ant) LOOP
--Declaración ímplicita y ejecución parametrizada del cursor
    v_iva_fac := 1 + v_dat_fac.iva / 100;
    v_dto_fac := 1 – v_dat_fac.dto / 100;
    SELECT SUM (cant * precio * (1 - dto/100))
      INTO v_precio_fac
      FROM lineas_fac
      WHERE codfac = v_dat_fac.codfac;
    v_importe_mes:=v_importe_mes+v_precio_fac*v_iva_fac*v_dto_fac;
END LOOP;
INSERT INTO balances VALUES (LAST_DAY (v_mes_ant), v_importe_mes);
COMMIT;
END;
```

2. Tipos de Datos Estructurados

Entre los tipos de datos estructurados que proporciona PL/SQL están los registros (`RECORD`) y los vectores (`TABLE` y `VARRAY`). Todos ellos se declaran en la sección `DECLARE`.

Para declarar un tipo registro se emplea la siguiente sintaxis:

```
TYPE nombre_tipo IS RECORD (campo [, campo] ...);
```

```
TYPE tipo_empleado_reg IS RECORD
  (nombre VARCHAR2(10),
   puesto VARCHAR2(8),
   sueldo NUMBER(6));

v_empleado_reg tipo_empleado_reg
```

Alternativamente se puede indicar que el tipo de la variable sea el de los registros de una tabla existente:

```
v_empleado_reg empleado%ROWTYPE;
```

Para declarar los vectores se emplea la siguiente sintaxis:

```
TYPE nombre_tipo IS VARRAY (tamaño_maximo) OF tipo_datos [NOT NULL];
TYPE nombre_tipo IS TABLE OF tipo_datos [NOT NULL];
```

En ambos casos los índices se empiezan a contar a partir de 1, y para poder utilizar los vectores, deben ser previamente creados vacíos o con elementos. A partir de entonces para insertar elementos adicionales se tienen que extender, como muestra el siguiente ejemplo:

```
DECLARE
   TYPE t_varray IS VARRAY (50) OF empleado.nombre%TYPE;
   v_varray1 t_varray;
   v_varray2 t_varray;

BEGIN
  ...
  v_varray1 := t_varray('Ana', 'Lola');-- se crea con dos elementos
  v_varray1.EXTEND;
  v_varray1(3) := 'Luis';
  -- v_varray1(4) := 'Juan'; Esto sería un error porque no se ha extendido
  v_varray2 := t_varray(); -- se crea vacío
  IF v_varray2 IS NULL -- cierto
       THEN v_varray2 := v_varray1; --asignación de vectores
  ....
END;
```

Una diferencia que hay entre `TABLE` y `VARRAY` es que en el primer caso el vector puede crecer ilimitadamente, pero en el segundo caso solo puede crecer hasta el tamaño máximo definido. Otra diferencia es que en los tipos `VARRAY` no se pueden borrar elementos, por lo que sus posiciones se deben ocupar consecutivamente. Sin embargo, en los tipos `TABLE` se pueden borrar elementos con la instrucción `DELETE`, pudiendo quedar huecos intermedios vacíos y sin poderse referenciar, aunque se pueden volver a llenar con una asignación. La función `EXISTS` nos permite saber si un elemento se puede referenciar o ha sido borrado. Véase el siguiente ejemplo:

```
DECLARE
   TYPE t_table IS TABLE OF empleado.nombre%TYPE;
   v_table1 t_table;
   v_table2 t_table;

BEGIN
  ...
  v_table1 := t_table('Ana', 'Lola');
  v_table1(2) := NULL; --Dejar una posición vacía no es igual que borrarla
  v_table1.DELETE(1);--Así es como se borra una posición
  v_table1.EXTEND;
  v_table1(3) := 'Luis';
  -- v_table1(4) := 'Juan'; Esto sería un error porque no se ha extendido
  v_table2 := t_table();
  IF v_table1(1).EXISTS
      THEN ...;
      ELSE v_table1(1) := 'Pepe'; --se vuelve a crear el elemento 1
  END IF;
  ...
END;
```

Algunos de los métodos predefinidos para manejar vectores son: `EXISTS`, `COUNT`, `FIRST`, `LAST`, `PRIOR`, `NEXT`, `EXTEND`, `TRIM`, `DELETE`.

3. Funciones y Procedimientos

Los bloques PL/SQL se pueden almacenar como objetos de la base de datos para ser ejecutados repetidamente. Los parámetros de entrada y salida permiten comunicarse con los procedimientos o funciones. Para ejecutarlos se invocan por su nombre con el comando `EXECUTE`, aunque previamente deben ser compilados con el comando `START`. Para depurar los errores de compilación se puede utilizar el comando `SHOW ERRORS`. Para conocerlos se pueden utilizar dos vistas del diccionario de datos `user_objects` y `user_source`, y para borrarlos se utiliza los comandos `DROP PROCEDURE nombre_procedimiento` o `DROP FUNCTION nombre_funcion`. La sintaxis de definición de funciones y procedimientos es como sigue:

```
CREATE [OR REPLACE] FUNCTION nombre_funcion
(argumento1 [modo1] tipodato1, argumento2 [modo2] tipodato2, ...)
RETURN tipodato
IS|AS
Bloque PL/SQL;
```

```
CREATE [OR REPLACE] PROCEDURE nombre_procedimiento
(argumento1 [modo1] tipodato1, argumento2 [modo2] tipodato2, ...)
IS|AS
Bloque PL/SQL;
```

El tipo de un parámetro no puede tener restricciones de tamaño y su modo por defecto es `IN`, aunque también se puede definir como `OUT` o `IN OUT`. Si no hay parámetros, se omiten los paréntesis. Las funciones devuelven siempre un valor con la instrucción `RETURN`, aunque pueden devolver más valores con parámetros de tipo `OUT`. En el bloque PL/SQL asociado a un procedimiento o función se debe omitir la palabra `DECLARE`.

Para recoger los valores de salida y/o para introducir valores en la ejecución a través de los parámetros se deben utilizar constantes y variables externas, previamente definidas en SQL*Plus con la instrucción `VARIABLE nombre_var`. De esta manera, aunque no se pueden utilizar variables externas dentro de procedimientos ni funciones, en una llamada a procedimiento se pueden utilizar poniendo su nombre precedido del carácter ‘:’. El siguiente ejemplo ilustra cómo declarar y ejecutar un procedimiento, recogiendo sus resultados en variables externas que después pueden ser visualizadas con la instrucción `PRINT`.

```
CREATE OR REPLACE PROCEDURE consulta_emp
  (v_id IN empleado.numemp%TYPE,
   v_nombre OUT empleado.nombre%TYPE,
   v_sueldo OUT empleado.sueldo%TYPE,
   v_comis OUT empleado.comision%TYPE)
IS
   BEGIN
     SELECT nombre, sueldo, comision
     INTO v_nombre, v_sueldo, v_comis
     FROM empleado
     WHERE numemp=v_id;
   END consulta_emp;
/
```

Suponiendo que el procedimiento se almacena en el fichero `consulta1.sql`, para su ejecución en SQL*Plus hay que hacer:

```
START consulta1.sql; /*compilación*/

VARIABLE ex_nombre VARCHAR2(15);
VARIABLE ex_sueldo NUMBER;
VARIABLE ex_comision NUMBER;
```

```
EXECUTE consulta_emp(6565, :ex_nombre, :ex_sueldo, :ex_comision);

PRINT ex_nombre;
PRINT ex_sueldo;
PRINT ex_comision;
```

A continuación se da un ejemplo de una función que devuelve un valor distinto dependiendo de los datos que lee de cierta tabla.

```
CREATE OR REPLACE FUNCTION ocupa_clase
  (p_departamento IN clases.dept%TYPE,
   p_curso IN clases.curso%TYPE)
RETURN VARCHAR2
IS
   v_numest NUMBER;
   v_maxest NUMBER;
   v_porcentaje NUMBER;
   BEGIN
     SELECT numero_est, num_max
       INTO v_numest, v_maxest
       FROM clases
       WHERE dept = p_departamento AND curso = p_curso;
     v_porcentaje := v_numest / v_maxest * 100;
     IF v_porcentaje = 100 THEN
       RETURN 'completa';
     ELSIF v_porcentaje > 80 THEN
       RETURN 'algo de sitio';
     ELSIF v_porcentaje > 60 THEN
       RETURN 'bastante sitio';
     ELSIF v_porcentaje > 0 THEN
       RETURN 'mucho sitio';
     ELSE
       RETURN 'vacia';
     END IF;
END ocupa_clase;
/
```

Suponiendo que la función se almacena en el fichero `ocupacion.sql`, para su ejecución en SQL*Plus hay que hacer:

```
START ocupación.sql; --compilación
SELECT ocupa_clase('Informática', 3) FROM dual;
```

Alternativamente, se podría utilizar una variable global para el resultado:

```
START ocupación.sql; --compilación
VARIABLE ex_ocupacion VARCHAR2(15);
EXECUTE :ex_ocupacion := ocupa_clase('Informática', 3);
PRINT ex_ocupacion;
```

Finalmente, después de compilar la función y dentro de un bloque PL/SQL se puede hacer:

```
declare
   v_ocupa VARCHAR2(15);
begin
   v_ocupa := ocupa_clase('Informática', 3);
   dbms_output.put_line(v_ocupa);
end;
```

4. Paquetes

Un paquete es una estructura PL/SQL que permite almacenar juntos una serie de objetos relacionados. Un paquete tiene dos partes bien diferenciadas, la especificación y el cuerpo del paquete, los cuales se almacenan por separado en el diccionario de datos. Dentro de un paquete se pueden incluir procedimientos, funciones, cursores, tipos y variables. Posteriormente, estos elementos se pueden referenciar desde otros bloques PL/SQL, con lo que los paquetes permiten disponer de variables globales en PL/SQL.

La especificación del paquete se hace en un fichero y contiene información acerca de sus contenidos. El código de ejecución se almacena en otro fichero aparte con el cuerpo del paquete. Previamente a compilar el cuerpo y ejecutar sus procedimientos y funciones, hay que compilar la cabecera. Sus sintaxis respectivas son:

```
CREATE [OR REPLACE] PACKAGE nombre_paquete IS|AS
  especificación de procedimiento o funcion
  |declaración de variable
  |declaración de tipo
  |declaración de excepcion
  |declaración de cursor
END nombre_paquete;
```

```
CREATE [OR REPLACE] PACKAGE BODY nombre_paquete IS|AS
  cuerpos de procedimiento o funcion
END nombre_paquete;
```

Después de compilar la especificación y el cuerpo de un paquete (con la instrucción `START`), quedan almacenados en el diccionario de datos, y se pueden invocar en cualquier momento a sus elementos precediéndolos del nombre del paquete y un punto. Por ejemplo:

```
CREATE OR REPLACE PACKAGE paquete_cont IS
  g_cont NUMBER := 0; -- se inicializa la variable global
  PROCEDURE reset_cont (v_nuevovalor IN NUMBER);
END paquete_cont;


CREATE OR REPLACE PACKAGE BODY paquete_cont IS
  PROCEDURE reset_cont (v_nuevovalor IN NUMBER) IS
   begin
    g_cont := v_nuevovalor;
   end reset_cont;
END paquete_cont;
```

Ahora para actualizar el valor de la variable global desde cualquier bloque PL/SQL tenemos dos opciones: acceder directamente a la variable globar definida en el paquete o ejecutar el procedimiento del paquete que tambien la actualiza.

```
EXECUTE paquete_cont.g_cont:=5;
EXECUTE dbms_output.put_line(paquete_cont.g_cont);

EXECUTE paquete_cont.reset_cont(5);
EXECUTE dbms_output.put_line(paquete_cont.g_cont);
```

5. Disparadores de Base de Datos

El siguiente diagrama resume la sintaxis para la creación de disparadores (triggers) en Oracle.

```
CREATE [OR REPLACE] TRIGGER trigger
{BEFORE | AFTER | INSTEAD OF}
      [DELETE] [OR INSERT] [OR UPDATE [OF columna [,columna]]
            ON tabla | vista
[REFERENDING [OLD [AS] old] [NEW [AS] new]]
[FOR EACH {ROW | STATEMENT}]
[WHEN condición]
Bloque PL/SQL
```

Para declarar el disparador, después de su nombre, hay que indicar si se tiene que ejecutar antes, después o en lugar de la ejecución de la sentencia SQL (`delete`, `insert`, `update`) que ha causado su disparo. Un mismo disparador puede tener varios sucesos asociados, y una instrucción SQL sólo se corresponde con un suceso, aunque afecte a varias tuplas. Para diferenciar dentro del bloque PL/SQL cuál de los posibles sucesos es el que ha activado al disparador, se pueden utilizar los predicados condicionales `INSERTING`, `UPDATING` y `DELETING`.

```
IF INSERTING THEN
  v_valor := 'I';
ELSIF UPDATING THEN
  v_valor := 'U';
ELSE
  v_valor := 'D';
END IF;
```

A continuación va la palabra `ON` y el nombre de la tabla a la que se asocia el disparador. La cláusula `REFERENCING` sirve para asignar alias a las variables externas `old` y `new`, las cuales almacenan respectivamente los valores de una tupla antes y después de ejecutar sobre ella la instrucción que lanza al disparador. Sin embargo, estas variables solamente se pueden utilizar cuando se escoge la opción de ejecutar el disparador tupla por tupla (`for each row`), es decir ejecutarlo separadamente para cada una de las tuplas afectadas por la instrucción SQL que ha activado al disparador.

```
CREATE OR REPLACE TRIGGER Sal_Total
AFTER INSERT OR UPDATE OF salario ON empleado
FOR EACH ROW
WHEN (New.dep_num IS NOT NULL)
UPDATE departamento
SET sal_total= sal_total + New.salario
WHERE dep_num = New.dep_num;
```

En el caso de que el disparador se tenga que ejecutar una sola vez para la instrucción que causa su disparo (`for each statement`), entonces las variables `old` y `new` no se pueden utilizar, ya que la instrucción puede afectar a varias tuplas. La cláusula optativa `WHEN` sirve para especificar una condición adicional que debe cumplirse para que el cuerpo del disparador sea ejecutado. El cuerpo del disparador consiste en un bloque PL/SQL.

```
CREATE OR REPLACE TRIGGER hacer_pedido
/*Disparador para vigilar que cuando la cantidad de piezas caiga por
debajo del mínimo se haga un pedido de más piezas, si no se ha hecho ya*/
AFTER UPDATE OF cantidad ON piezas
FOR EACH ROW WHEN (new.cantidad < new.cant_min)
  DECLARE
    v_pendientes NUMBER;
  BEGIN
    SELECT count(*) INTO v_pendientes FROM pedidos
    WHERE numpie = :new.numpie
    IF v_pendientes = 0 THEN
      INSERT INTO pedidos VALUES (:new.numpie, :new.cant_pedido, sysdate);
    END IF;
  END;
```

Hay que tener en cuenta que desde el cuerpo de los disparadores de nivel de fila (`for each row`) no es posible ejecutar ordenes SQL de lectura o actualización sobre la tabla asociada al disparador, ni sobre las claves de las tablas referenciadas desde la tabla asociada al disparador por medio de una clave ajena.

Para activar un disparador se tiene que compilar con éxito, y se puede desactivar con la instrucción siguiente:

```
ALTER TRIGGER nombre_disparador [DISABLE|ENABLE]
```

Para borrar un disparador se ejecuta la acción:

```
DROP TRIGGER nombre_disparador
```

Para consultar la información de los disparadores se puede ejecutar la siguiente consulta sobre el diccionario de datos:

```
SELECT trigger_type, table_name, triggering_event
FROM user_triggers
WHERE trigger_name = '...';
```