6. Programación de bases de datos

Class	Bases de datos
Column	Xerach Casanova
Lass Edited time	@Mar 29, 2021 9:25 PM

1. Introducción

PL/SQL es un lenguaje procedimental estructurado en bloques, que amplía la funcionalidad de SQL, con él podemos usar sentencias SQL para manipular datos y sentencias de control de flujo para procesar los datos, combinando la potencia de SQL con la potencia de lenguajes procedimentales para procesar datos.

Fue creado por Oracle, pero lo utilizan todos los gestores de bases de datos.

Se pueden definir variables, constantes, funciones, procedimientos, capturar errores, anidar cualquier número de bloques, etc. También podemos usar disparadores.

El motor de PL/SQL acepta como entrada bloques PL/SQL o subprogramas, ejecuta las sentencias procedimentales y envía sentencias SQL al servidor de base de datos.

La gran ventaja es el mejor rendimiento en entornos de red cliente-servidor, ya que se envían bloques de PL/SQL desde el cliente al servidor y no se envían las sentencias SQL una a una.

2. Conceptos básicos

2.1. Unidades léxicas (I)

PL/SQL no es sensible a las mayúsculas, podemos escribir en mayúsculas y minúsculas excepto cuando hablamos de literales de tipo cadena o de tipo carácter.

Cada unidad léxica se puede separar por espacios (debe estar separada por espacios si se trata de 2 identificadores), por saltos de línea o por tabuladores, para aumentar la legibilidad.

Ejemplos equivalentes:

```
IF A=CLAVE THEN ENCONTRADO:=TRUE;ELSE ENCONTRADO:=FALSE;END IF;
```

```
if a=clave then encontrado:=true;else encontrado:=false;end if;
```

```
IF a = clave THEN
     encontrado := TRUE;
ELSE
     encontrado := FALSE;
END IF;
```

Las unidades léxicas se clasifican en:

Delimitadores

Se utilizan para representar operaciones entre tipos de datos, delimitar comentarios, etc.

Delimitadores Simples.		Delimitadores Compuestos.	
Símbolo.	Significado.	Símbolo.	Significado.
+	Suma.	**	Exponenciación.
%	Indicador de atributo.	<>	Distinto.
.	Selector.	¡=	Distinto.
/	División.	<=	Menor o igual.
(	Delimitador de lista.	>=	Mayor o igual.
)	Delimitador de lista.	..	Rango.
:	Variable host.	||	Concatenación.
,	Separador de elementos.	<<	Delimitador de etiquetas.
*	Producto.	>>	Delimitador de etiquetas.
"	Delimitador de identificador acotado.	--	Comentario de una línea.
=	Igual relacional.	/*	Comentario de varias líneas.
<	Menor.	*/	Comentario de varias líneas.
>	Mayor.	:=	Asignación.
@	Indicador de acceso remoto.	=>	Selector de nombre de parámetro.
;	Terminador de sentencias.		
-	Resta/negación.		

2.1.1. Unidades léxicas (II)

Identificadores

Se utilizan para nombrar elementos y debemos tener los siguientes aspectos:

- Es una letra seguida opcionalmente de letras, números, $, _, #.
- No se puede utilizar como identificador una palabra reservada.
- Podemos definir los identificadores acotados, en los que se puede utilizar cualquier carácter, con una longitud máxima de 30 y deben estar delimitados por ". Ejemplo: "X*Y".
- Existen algunos identificadores predefinidos con un significado especial para dar sentido sintáctico a nuestros programas. Son palabras reservadas: IF, THEN, ELSE...
- Algunas palabras reservadas para PL/SQL no lo son para SQL, por lo que podemos tener una tabla con una columna llamada 'type' que nos dará error de compilación, para solucionarlo se acota: "type".

Literales

Se utilizan en las comparaciones de valores para asignar valores concretos a los identificadores, que actúan como variables o constantes.

- Los literales numéricos se expresan en notación decimal o exponencial: 234, +341, 2e3, -2E-3, 7.45, 8.1e3
- Los literales tipo carácter se delimitan con comillas simples.
- Los literales lógicos son TRUE y FLASE.
- El literal NULL indica que la variable no tiene valor.

Comentarios

No tienen efectos sobre el código pero ayudan a los programadores a recordar lo que se está haciendo. Existen dos tipos

- Comentarios de una línea, con delimitador — (doble guión).
- Comentario de varias líneas con /**/.

2.2. Tipos de datos simples, variables y constantes

En PL/SQL contamos con todos los tipos de datos simples utilizados en SQL y algunos más. Los más utilizados son:

Numéricos

- **BINARY_INTEGER.** Tipo numérico de rango entre -2147483647 y 2147483647, Además se definen algunos subtipos: NATURAL, NATURALN, POSITIVE, POSITIVEN, SIGNTYPE.
- **NUMBER:** Tipo numérico que almacena números racionales. Se puede especificar su escala (-84 a 127) y su precisión (1 a 38), La escala indica cuándo se redondea y hacia donde. Ejemplo: escala = 2: 8.234 → 8.23 / escala = -3: 7689 → 8000. Si definimos un NUMBER(6,2) indicamos que son 6 digitos numéricos de los cuales 2 son decimales. También se definen algunos subtipos como: DEC, DECIMAL, DOUBLE PRECISION, FLOAT, INTEGER, INT, NUMERIC, REAL SMALLINT.
- **PLS_INTERGER:** Tipo numérico, con rango igaul al BINARY_INTEGER. Su representación es distinta y las operaciones llevadas a cabo con ellos serán más eficientes.

Alfanumérico

- **CHAR(n)** Array de n caracteres, máximo 2000 bytes, la longitud por defecto es 1, ocupa en memoria n caracteres.
- **LONG:** Array de caracteres con un máximo de 32760 bytes.
- **RAW:** Array de bytes con número máximo de 2000.
- **LONGRAW:** Array de bytes con un máximo de 32760.
- **VARCHAR2(n): a**lmacena cadenas de longitud variable con un máximo de 32760. Ocupa en memoria n caracteres.

Grandes objetos

- **BFILE:** Puntero a un fichero del S.O.
- BLOB: Objeto binario con capacidad de 4GB.

- GLOB: Objeto carácter con capacidad de 2GB.

Otros

- BOOLEAN: TRUE/FALSE
- DATE: almacena valores de día y hora, desde el 1 de enero de 4712 a.c. hasta el 31 de diciembre de 4712 d.c.

Atributo %TYPE

Si utilizamos el atributo %TYPE nos aseguramos cuando declaramos la variable que tiene el mismo tipo de datos que la variable especificada. Normalmente se relaciona con una columna de la BD indicando el nombre de la tabla de la bd y el de la columna. Si se hace referencia a un variable declarada anteriormente, se indica el nombre de la variable previamente declarada a la variable por declarar.

El uso de este atributo ayuda a que si se cambia la definición de una columna no tendremos que que cambiar la declaración de la variable.

Ejemplos:

```
nombre_oficina oficinas.nombre%type; -- la variable nombre_oficina tomará el mismo tipo de dato que la columna nombre de  la tabla ofi

vx number(5,2); -- Variable numérica de 5 dígitos, dos de los cuales son decimales

vy vx%type; -- la variable vy tomará el mismo tipo de dato que tenga la variable vx
```

2.2.1. Subtipos

Los subtipos nos permiten definir subtipos de tipos de datos para darles un nombre distinto y aumentar legibilidad de nuestros programas. Las operaciones aplicables a estos subtipos son los mismas que con los que los preceden.

```
SUBTYPE subtipo IS tipo_base;
```

subtipo es el nombre que le damos a nuestro subtipo y tipo_base es cualquier tipo de dato que le damos a nuestro tipo de dato PL/SQL

Cuando especificamos el tipo base, se puede utilizar el modificador %TYPE para indicar el tipo de dato de una variable o columna de la bd y %ROWTYPE para especificar el tipo de un cursor o tabla de una base de datos:

```
SUBTYPE id_familia IS familias.identificador%TYPE; -- Permite nombrar a la columna identificador de la tabla famiias con el nombre id_
SUBTYPE agente IS agentes%ROWTYPE; -- Permite nombrar a las filas de la tabla agentes con el nombre agente
```

No se pueden restringir los subtipos, pero podemos seguir el mismo efecto por medio de una variable auxiliar:

```
BTYPE apodo IS varchar2(20);          --ilegal
aux varchar2(20);
SUBTYPE apodo IS aux%TYPE;               --legal
```

Los subtipos son intercambiables con su tipo base, también son intercambiables si tienen el mismo tipo base o si su tipo base pertenece a la misma familia.

```
DECLARE
     SUBTYPE numero IS NUMBER;
     numero_tres_digitos NUMBER(3);
     mi_numero_de_la_suerte numero;
     SUBTYPE encontrado IS BOOLEAN;
     SUBTYPE resultado IS BOOLEAN;
     lo_he_encontrado encontrado;
     resultado_busqueda resultado;
     SUBTYPE literal IS CHAR;
     SUBTYPE sentencia IS VARCHAR2;
     literal_nulo literal;
     sentencia_vacia sentencia;
BEGIN
     ...
     numero_tres_digitos := mi_numero_de_la_suerte;     --legal
     ...
     lo_he_encontrado := resultado_busqueda;               --legal
     ...
```

```
     sentencia_vacia := literal_nulo;                  --legal
     ...
END;
```

2.2.2. Variables y constantes

Para declarar variables o constantes se pone el nombre de la variable, seguido del tipo de dato y opcionalmente se le asigna un valor con el operador :=, si es una constante anteponemos la palabra CONSTANT al tipo de dato. Se puede sustituir el operador de asignación por la palabra reservada DEFAULT, o podemos forzar que no sea nula con NOT NULL después del tipo y antes de la asignación, la cual es obligatoria al declararla o se lanza una excepción VALUE_ERROR.

```
id SMALLINT;
hoy DATE := sysdate;
pi CONSTANT REAL:= 3.1415;
id SMALLINT NOT NULL; --ilegal, no está inicializada
id SMALLINT NOT NULL := 9999; --legal
nombre varchar2(30):= ;'Alejandro Magno';
num INTEGER DEFAULT 4; -- también se puede utilizar DEFAULT para inicializar una variable
```

Conversión de tipos

Aunque existe la conversión implícita de tipos para tipos parecidos siempre es aconsejable utilizar la conversión explícita por medio de funciones de conversión.

Precedencia de operadores

Se utilizan para realizar operaciones aritméticas, si dos operadores tienen la misma precedencia se evalúa de izquierda a derecha. Es aconsejable utilizar paréntesis para alterar la precedencia de los mismos.

Operador.	Operación.
**, NOT	Exponenciación, negación lógica.
+, -	Identidad, negación.
*, /	Multiplicación, división.
+, -, ||	Suma, resta y concatenación.
=, ¡=, <, >, <=, >=, IS NULL, LIKE, BETWEEN, IN	Comparaciones.
AND	Conjunción lógica
OR	Disyunción lógica.

2.3. El bloque PL/SQL

La unidad básica en PL/SQL es el bloque, que consta de tres zonas:

- **Declaraciones:** definición de variables, constantes, cursores y excepciones.
- **Procesos:** zona donde se realiza el proceso en sí, conteniendo sentencias ejecutables.
- **Excepciones:** zona de manejo de errores en tiempo de ejecución.

La sintaxis es:

```
[DECLARE
     [Declaración de variables, constantes, cursores y excepciones]]
BEGIN
     [Sentencias ejecutables]
[EXCEPTION
     Manejadores de excepciones]
END;
```

Los bloques se pueden anidar a cualquier nivel, el ámbito y la visibilidad de las variables es la misma que un lenguaje procedimental. En el siguiente ejemplo, aux es variable global, pero también está declarada como variable local y la visibilidad dominante es la de la variable local, por tanto, vale 5

```
DECLARE
     aux number := 10;-- Variable global
```

```
BEGIN
     DECLARE
          aux number := 5;    -- Variable local al bloque donde es definida
     BEGIN
          ...
          IF aux = 10 THEN     --evalúa a FALSE, no entraría
          ...
     END;
END;
```

2.4. Estructuras de control (I)

Como en todos los lenguajes de programación existen dos: condicionales e iterativas.

Control condicional. sentencia if

Sus variantes son

- **Sentencia IF -THEN.** Si la evaluación de la condición es TRUE se ejecuta la sentencia entre el then y el final de la sentencia.

```
IF condicion THEN
secuencia_de_sentencias;
END IF;
```

Ejemeplo:

```
SET SERVEROUTPUT ON
DECLARE a integer:=10;
B integer:=7;
BEGIN
IF a>b
 THEN dbms_output.put_line(a || ' es mayor'); -- Como la función put_line solo imprime un valor utilizamos la concatenación || para qu
 END IF;
END;
```

- **Sentencia IF-THEN-ELSE.** Si la evaluación es TRUE se ejecuta la primera secuencia de sentencias y si no, la segunda.

```
IF condicion
THEN Secuencia_de_sentencias1;
 ELSE Secuencia_de_sentencias2;
END IF;
```

```
DECLARE
a integer:=10;
b integer:=17;
BEGIN
IF a>b THEN
  dbms_output.put_line(a || ' es mayor');
ELSE
  dbms_output.put_line(b || ' es mayor o iguales');
END IF;
END;
/
```

- **Sentencia IF-THEN-ELSIF:** se trata de una condición múltiple, si la condición, podemos poner cuantos ELSIF queramos, se ejecutará la sentencia de la condición que sea TRUE

```
IF condicion1 THEN
     Secuencia_de_sentencias1;
ELSIF condicion2 THEN
     Secuencia_de_sentencias2;
     ...
[ELSE
     Secuencia_de_sentencias;]
END IF;
```

```
IF (operacion = ‘SUMA’) THEN
     resultado := arg1 + arg2;
ELSIF (operacion = ‘RESTA’) THEN
     resultado := arg1 – arg2;
ELSIF (operacion = ‘PRODUCTO’) THEN
     resultado := arg1 * arg2;
ELSIF (arg2 <> 0) AND (operacion = ‘DIVISION’) THEN
     resultado := arg1 / arg2;
ELSE
     RAISE operacion_no_permitida; -- Lanza un error de ejecución
END IF;
/
```

Sentencia CASE

Representa n sentencias if anidadas y es más fácil de interpretar cuando se compara con varios valores, permitiendo sustituir a las sentencias if encadenadas.

- Utilizando un selector y un manejador WHEN para cada posible valor de selector

```
CASE  selector
                    WHEN  expression1  THEN
                    ordenes;
                    [ WHEN  expression2 THEN
                    ordenes;
                    ...
                    [WHEN  expression  THEN
                    ordenes;]
                    [ ELSE
                    ordenes;]
                    END CASE ;
```

- Utilizando condiciones de búsqueda:

```
CASE
WHEN  condición1  THEN
ordenes;
[ WHEN  condición2  THEN
ordenes;
...
WHEN  condiciónN  THEN
ordenes;]
[ ELSE
ordenes;]
END CASE ;
```

Antes de terminar la sentencia CASE se puede especificar ELSE por si no se ha cumplido ninguna de las condiciones anteriores. Todas las condiciones se analizan en el orden listado. Desde que se encuentra una condición verdadera se deja de analizar el resto.

```
DECLARE
    nota INTEGER:=8; -- Se podría especificar nota INTEGER:=&nota
BEGIN
    CASE
       WHEN nota in(1,2) THEN
            DBMS_OUTPUT.PUT_LINE('Muy deficiente');
       WHEN nota in (3,4) THEN
            DBMS_OUTPUT.PUT_LINE('Insuficiente');
       WHEN nota = 5 THEN
            DBMS_OUTPUT.PUT_LINE('Suficiente');
       WHEN nota=6 THEN
            DBMS_OUTPUT.PUT_LINE('Bien');
        WHEN nota in(7,8) THEN
            DBMS_OUTPUT.PUT_LINE('Notable');
        WHEN nota in (9,10) THEN
            DBMS_OUTPUT.PUT_LINE('Sobresaliente');
      ELSE
            DBMS_OUTPUT.PUT_LINE('Error, no es una nota');
    END CASE;
END;
/
```

Para pedir datos por teclado se utilizan variables de sustitución escribiendo el carácter especial '&' y a continuación un identificador, si el dato a introducir es alfanumérico se escribe entre comillas simple:

```
DECLARE
cadena varchar2(25) :='&cad';
BEGIN
    DBMS_OUTPUT.PUT_LINE(cadena);
END;/
```

2.4.1 Estructuras de control (II). Bucles

Control iterativo

Ejecuta sentencias un determinado número de veces:

- **LOOP:** La forma simple es el bucle infinito:

```
LOOP
     secuencia_de_sentencias;
END LOOP;
```

- **EXIT.** Con ella forzamos a terminar el bucle. No fuerza nunca la salida de un bloque, solo del bucle.

```
LOOP
     ...
     IF encontrado = TRUE THEN
          EXIT;
     END IF;
END LOOP;
```

```
DECLARE
  a integer :=1;
BEGIN
  LOOP
    dbms_output.put_line(a);
    IF a>9 THEN
       EXIT;
    END IF;
    a:=a+1;
  END LOOP;
END;
/
```

- **EXIT WHEN condicion.** Fuerza a salir del bucle si se cumple una determinada condición.

```
LOOP
     ...
     EXIT WHEN encontrado;
END LOOP;
```

```
DECLARE
  a integer :=1;
BEGIN
  LOOP
    dbms_output.put_line(a);
    EXIT WHEN a>9;
    a:=a+1;
  END LOOP;
END;
/
```

- **WHILE LOOP:** ejecuta la secuencia de sentencias mientras la condición sea cierta.

```
WHILE condicion LOOP
     Secuencia_de_sentencias;
END LOOP;
```

```
DECLARE
  a integer :=1;
BEGIN
  WHILE a<10 LOOP
    dbms_output.put_line(a);
    a:=a+1;
  END LOOP;
END;/
```

FOR-LOOP. El bucle itera mientras el contador se encuentre en el rango definido.

```
FOR contador IN [REVERSE] limite_inferior..limite_superior LOOP
     Secuencia_de_sentencias;
END LOOP;
```

```
BEGIN
FOR a IN 1..10 LOOP -- ascendente de uno en uno
  dbms_output.put_line(a);
END LOOP;
FOR a IN REVERSE 1..10 LOOP -- descendente de uno en uno
  dbms_output.put_line(a);
END LOOP;
END;
/
```

2.5. Manejo de errores (I)

Cualquier situación de error se llama excepción, cuando se detecta se lanza una excepción, la ejecución normal se para y el control se transfiere a la parte de manejo de excepciones, la cual está etiquetada como EXCEPTION y cuenta con sentencias llamadas manejadores de excepciones.

Manejadores de excepciones

Sintaxis:

```
WHEN nombre_excepcion THEN
     <sentencias para su manejo>
     ....
WHEN OTHERS THEN
     <sentencias para su manejo>
```

Ejemplo:

```
DECLARE
     supervisor agentes%ROWTYPE;
BEGIN
     SELECT * INTO supervisor FROM agentes
     WHERE categoria = 2 AND oficina = 3;
          ...
EXCEPTION
     WHEN NO_DATA_FOUND THEN
     --Manejamos el no haber encontrado datos
WHEN OTHERS THEN
     --Manejamos cualquier error inesperado
END;
/
```

Las excepciones las puede definir el usuario o están definidas internamente, las predefinidas se lanzan automáticamente asociadas a un error Oracle, las definidas por el usuario se deben definir y lanzar explícitamente.

Para definir nuestras excepciones, lo hacemos en la parte DECLARE de cualquier bloque. Las podemos lanzar explícitamente on la sentencia RAISE nombre_excepción.

Excepciones definidas por el usuario

Sintaxis:

```
DECLARE
     nombre_excepcion EXCEPTION;
```

```
BEGIN
     ...
     RAISE nombre_excepcion;
     ...
END;
```

Ejemplo

```
DECLARE
     categoria_erronea EXCEPTION;
BEGIN
     ...
     IF categoria<0 OR categoria>3 THEN
          RAISE categoria_erronea;
     END IF;
     ...
EXCEPTION
     WHEN categoria_erronea THEN
          --manejamos la categoria errónea
END;
```

2.5.1. Manejo de errores (II)

Detalles de uso de las excepciones:

- Al igual que en las variables, las excepciones locales redefinidas (que ya eran globales para el bloque), prevalece la definición local y no podremos capturar esa excepción a menos que el bloque donde estaba fuese un bloque nombrado, con lo cual podremos capturarla usando: nombre_bloque.nombre_excepcion
- Las excepciones predefinidas están definidas globalmente, ni necesitamos, ni debemos redefinirlas.

```
DECLARE
     no_data_found EXCEPTION;
BEGIN
     SELECT * INTO ...
EXCEPTION
     WHEN no_data_found THEN     --captura la excepción local, no
                          --la global
END;
```

- Cuando manejamos una excepción se se puede continuar por la siguiente sentencia que se lanzó:

```
DECLARE
     ...
BEGIN
     ...
     INSERT INTO familias VALUES
(id_fam, nom_fam, NULL, oficina);
     INSERT INTO agentes VALUES
(id_ag, nom_ag, login, password, 0, 0, id_fam, NULL);
     ...
EXCEPTION
     WHEN DUP_VAL_ON_INDEX THEN
          --manejamos la excepción debida a que el nombre de
          --la familia ya existe, pero no podemos continuar por
          --el INSERT INTO agentes, a no ser que lo pongamos
          --explícitamente en el manejador
END;
```

Pero podemos encerrar la sentencia de dentro de un bloque y así capturar las posibles excepciones para continuar con otras sentencias.

```
DECLARE
     id_fam NUMBER;
     nom_fam VARCHAR2(40);
     oficina NUMBER;
     id_ag NUMBER;
     nom_ag VARCHAR2(60);
     usuario VARCHAR2(20);
     clave VARCHAR2(20);
BEGIN
     ...
     BEGIN
          INSERT INTO familias VALUES (id_fam, nom_fam, NULL, oficina);
```

```
     EXCEPTION
          WHEN DUP_VAL_ON_INDEX THEN
SELECT identificador INTO id_fam FROM familias WHERE nombre = nom_fam;
     END;
     INSERT INTO agentes VALUES (id_ag, nom_ag, login, password, 1, 1, id_fam, null);
     ...
END;
```

2.5.2. Manejo de errores (III)

- En vez de encapsular las excepciones en bloque para manejar cada una de ellas individualmente, podemos utilizar una variable localizadora para saber que sentencia la lanzó, pero de esta manera no podremos continuar con la siguiente sentencia a la excepción lanzada.

```
DECLARE
     sentencia NUMBER := 0;
BEGIN
     ...
     SELECT * FROM agentes ...
     sentencia := 1;
     SELECT * FROM familias ...
     sentencia := 2;
     SELECT * FROM oficinas ...
     ...
EXCEPTION
     WHEN NO_DATA_FOUND THEN
          IF sentencia = 0 THEN
               RAISE agente_no_encontrado;
          ELSIF sentencia = 1 THEN
               RAISE familia_no_encontrada;
          ELSIF sentencia = 2 THEN
               RAISE oficina_no_encontrada;
          END IF;
END;/
```

- Si la excepción capturada por un manejador de excepción apropiado, ésta es tratada y después el control se devolverá al entorno. Se puede dar que la excepción sea manejada en un bloque superior a falta de manejadores en bloques internos, propagándose al bloque superior, y así sucesivamente hasta que sea manejada o que no queden bloques superiores y el control se devuelve a l entorno.

2.5.3. Manejo de errores (IV)

Oracle también permite que lancemos nuestros propios mensajes de error a las aplicaciones y asociarlos a un código de error que Oracle reserva.

```
RAISE_APPLICATION_ERROR(error_number, message [, (TRUE|FALSE)]);
```

error_number es un número entero comprendido entre -20000 y -20099 y message es una cadena que devolvemos a la aplicación. El tercer parámetro especifica si el error se coloca en la pila de errores (TRUE) o se vacía la pila y se coloca únicamente el nuestro (FALSE). A este procedimiento podemos llamarlo desde un subprograma.

No hay excepciones predefinidas asociadas a todos los posibles errores, pero podemos asociar excepciones definidas propias a errores Oracle:

```
PRAGMA_INIT( nombre_excepcion, error_Oracle )
```

nombre_excepcion es el nombre de una excepción definida anteriormente y error_Oracle es el número negatovio asociado al error.

```
DECLARE
     no_null EXCEPTION;
     PRAGMA EXCEPTION_INIT(no_null, -1400);
     id familias.identificador%TYPE;
     nombre familias.nombre%TYPE;
BEGIN
     ...
     nombre := NULL;
     ...
     INSERT INTO familias VALUES (id, nombre, null, null);
```

```
EXCEPTION
     WHEN no_null THEN
          ...
END;
```

Oracle asocia 2 funciones para comprobar la ejecución de cualquier sentencia. SQLCODE nos devuelve el código de error y SQLERRM devuelve el mensaje de error asociado. Una sentencia ejecutada correctamente devuelve 0 en SQLCODE, en caso contrario devuelve un número negativo asociado al error (excepto NO_DATA_FOUND que tiene asociado el +100)

```
DECLARE
cod number;
msg varchar2(100);
BEGIN
...
EXCEPTION
WHEN OTHERS THEN
     cod := SQLCODE;
     msg := SUBSTR(SQLERRM, 1, 1000);
     INSERT INTO errores VALUES (cod, msg);
END;
```

2.6. Sentencias SQL en programas PL/SQL

Para trabajar con SQL dentro de un programa trabajaremos con SQL embebido incrustado, los datos devueltos de SQL los guardamos en variables y estructuras definidas para utilizarlas como convenga, añadiendo alguna cláusula en el SELECT.

PL/SQL soporta DML y comandos de control de transacciones. No soporta directamente DDL (CREATE TABLE, ALTER TABLE, etc.), ya que estas instrucciones son sentencias dinámicas en SQL. Tampoco soporta DCL (GRAND O REVOKE). Se puede utilizar SQL dinámico para ejecutarlas.

Si queremos recuperar datos de la BD utilizamos select:

```
SELECT  lista_campos INTO {nombre_variable[, nombre_variable]...| nombre_registro}
FROM tabla
[WHERE condición];
```

- La lista_campos contiene al menos una columna y puede incluir expresiones SQL, funciones de fila o de grupo.
- La cláusula INTO es obligatoria y se especifica entre SELECT Y FROM
- nombre_variable, donde se guarda el valor recuperado y se especifican tantas variables como campos indicados.
- nombre_registro, un dato compuesto de PL/SQL donde se guardan todos los valores recuperados.

Las instrucciones SELECT deben devolver una sola fila, una consulta que devuelve más de una fila o ninguna genera un error de tipo (TOO_MANY_ROWS o NO_DATA_FOUND). Para recuperar más de una fila usamos estructura de datos llamada cursor.

INSERT, UPDATE y DELETE se ejecutan de la misma manera que en SQL.

3. Tipos de datos compuestos

3.1. Registros

Un registro es un grupo de elementos asociados, referenciados por un único nombre y almacenados en campos, cada uno de ellos tiene su propio nombre y dato.

Hacen que la información sea más fácil de organizar. Por ejemplo. una dirección podría ser un registro con campos como calle, número, piso, puerta...

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

donde

```
decl_campo := nombre tipo [[NOT NULL] {:=|DEFAULT} expresion]
```

El tipo de campo es cualquier tipo válido en PL/SQL excepto REF CURSOR.

```
TYPE direccion IS RECORD
(
calle          VARCHAR2(50),
numero          INTEGER(4),
piso          INTEGER(4),
puerta          VARCHAR2(2),
codigo_postal     INTEGER(5),
ciudad          VARCHAR2(30),
provincia     VARCHAR2(20),
pais          VARCHAR2(20) := 'España'
);
mi_direccion direccion;
```

Para acceder a los campos utilizamos el operador punto.

```
...
mi_direccion.calle := 'Ramirez Arellano';

mi_direccion.numero := 15;

...
```

Para asignar un registro a otro, deben ser del mismo tipo, no basta que tengan el mismo número de campos y se emparejen uno a uno. Tampoco podemos comparar registros aunque sean del mismo tipo, ni comprobar si éstos son nulos.

```
DECLARE
TYPE familia IS RECORD
(
     identificador     NUMBER,
     nombre          VARCHAR2(40),
     padre          NUMBER,
     oficina          NUMBER
);
TYPE familia_aux IS RECORD
(
     identificador     NUMBER,
     nombre          VARCHAR2(40),
     padre          NUMBER,
     oficina          NUMBER
     );
     SUBTYPE familia_fila IS familias%ROWTYPE; -- tendrá los mismos campos que tenga la tabla familias
     mi_fam familia;
     mi_fam_aux familia_aux;
     mi_fam_fila familia_fila;
BEGIN
...
mi_fam := mi_fam_aux;                         --ilegal
mi_fam := mi_fam_fila;                         --legal
IF mi_fam IS NULL THEN ...                    --ilegal
IF mi_fam = mi_fam_fila THEN ...               --ilegal
SELECT * INTO mi_fam FROM familias ...          --legal
INSERT INTO familias VALUES (mi_fam_fila);          --ilegal
...
END;/
```

3.2. Colecciones. Arrays de longitud variable

Una colección es un grupo ordenado de elementos, todos del mismo tipo, cada elemento tiene un subíndice único que determina su posición en ella.

En PL/SQL las colecciones solo tienen una dimensión y ofrecen 2 clases: arrays de longitud variable y tablas anidadas.

Arrays de longitud variable

Son los elementos de tipo VARRAY. A la hora de declararlos se indica su tamaño máximo y el array puede ir creciendo dinámicamente hasta alcanzar su tamaño máximo.

```
TYPE nombre IS {VARRAY | VARYING} (tamaño_máximo) OF tipo_elementos [NOT NULL];
```

tamaño_máximo es un entero positivo y tipo_elementos será cualquier tipo de dato válido en PL/SQL excepto BINARY_INTEGER, BOOLEAN, LONG, LONG RAW, NATURAL, NATURALN, NCHAR, NCLOB, NVARCHAR2, objetos que tengan como atributos TABLE O VARRAY, PLS_INTEGER, POSITIVE, POSITIVEN, SIGNTYPE, STRING, TABLE, VARRAY. Si tipo_elementos es un registro, todos los campos deberían ser de un tipo escalar.

Cuando definimos un VARRAY es nulo y debemos inicializarlo para empezar a usarlo. Para ello se usa un constructor.

```
TYPE familias_hijas IS VARRAY(100) OF familia;
               familias_hijas1 familias_hijas := familias_hijas( familia(100, 'Fam100', 10, null), ..., familia(105, 'Fam105', 10, nu
```

Podemos usar constructores vacíos:

```
familias_hijas2 familias_hijas := familias_hijas();
```

Para referencias elementos en un VARRAY utilizamos sintaxis nombre_colección(subíndice). Si una función devuelve un VARRAY podemos utilizar la sintaxis: nombre_funcion(listaparametros)(subindice)

```
IF familias_hijas1(i).identificador = 100 THEN ...
               IF dame_familias_hijas(10)(i).identificador = 100 THEN ...
```

Un VARRAY se pude asignar a otro si ambos son del mismo tipo.

```
DECLARE
               TYPE tabla1 IS VARRAY(10) OF NUMBER;
               TYPE tabla2 IS VARRAY(10) OF NUMBER;
               mi_tabla1 tabla1 := tabla1();
               mi_tabla2 tabla2 := tabla2();
               mi_tabla tabla1 := tabla1();
               BEGIN
               ...
               mi_tabla := mi_tabla1;          --legal
               mi_tabla1 := mi_tabla2;     --ilegal
               ...
               END;
```

Para extender un VARRAY utilizamos el método EXTEND. Sin parámetros extendemos en 1 elemento nulo. EXTEND(n) añade n elementos nulos al VARRAY y EXTEND(n,i) añade n copias del i-ésimo elemento.

COUNT nos dice el número de elementos del VARRAY. LIMIT nos dice el tamaño máximo del VARRAY. FIRST siempre será 1. LAST siempre será igual a COUNT. PRIOR y NEXT devuelven el antecesor y sucesor del elemento.

Al trabajar con VARRAY podemos hacer que salte alguna de las siguientes excepciones por mal uso: COLLECTION_IS_NULL, SUBSCRIPT_BEYOND_COUNT, SUBSCRIPT_OUTSIDE_KIMIT Y VALUE_ERROR.

Ejemplos de uso:

Extender un VARRAY

```
DECLARE
          TYPE tab_num IS VARRAY(10) OF NUMBER;
          mi_tab tab_num;
BEGIN
           mi_tab := tab_num();
           FOR i IN 1..10 LOOP
                  mi_tab.EXTEND;
                   mi_tab(i) := calcular_elemento(i);
           END LOOP;
                     ...
END;
```

Consultar propiedades VARRAY

```
DECLARE
   TYPE numeros IS VARRAY(20) OF NUMBER;
   tabla_numeros numeros := numeros();
   num NUMBER;
BEGIN
    num := tabla_numeros.COUNT; --num := 0
    FOR i IN 1..10 LOOP
             tabla_numeros.EXTEND;
             tabla_numeros(i) := i;
    END LOOP;
    num := tabla_numeros.COUNT; --num := 10
    num := tabla_numeros.LIMIT; --num := 20
    num := tabla_numeros.FIRST; --num := 1;
    num := tabla_numeros.LAST; --num := 10;
...
END;
```

Posibles excepciones:

```
DECLARE
  TYPE numeros IS VARRAY(20) OF INTEGER;
  v_numeros numeros := numeros( 10, 20, 30, 40 );
  v_enteros numeros;
BEGIN
   v_enteros(1) := 15; --lanzaría COLECTION_IS_NULL
   v_numeros(5) := 20; --lanzaría SUBSCRIPT_BEYOND_COUNT
   v_numeros(-1) := 5; --lanzaría SUBSCRIPT_OUTSIDE_LIMIT v_numeros('A') := 25; --
 lanzaría VALUE_ERROR
....
```

3.2.1 Colecciones. Tablas anidadas

Las tablas anidadas son colecciones de elementos que no tienen límite superior fijo, pueden aumentar dinámicamente su tamaño y además podemos borrar elementos individuales.

```
TYPE nombre IS TABLE OF tipo_elementos [NOT NULL];
```

En tipo_elementos existen las mismas restricciones que en los VARRAY.

También son nulas al declararlas y se deben inicializar antes de usarlas.

```
TYPE hijos IS TABLE OF agente;
hijos_fam hijos := hijos( agente(...) ...);
```

También se pueden usar constructores nulos y para extenderlas también se hace igual que en los VARRAY.

COUNT nos dice el número de elementos, el cual no tiene por qué coincidir con LAST.

LIMIT no tiene sentido y devuelve NULL. EXISTS(n) devuelve TRUE si existe y FALSE si ha sido borrado.

FIRST devuelve el primer elemento, que no tiene que ser 1, ya que se puede borrar cualquier elemento del principio.

PRIOR y NEXT nos dicen el antecesor y el sucesor ignorando los borrados.

TRIM sin argumentos borra un elemento del final de la tabla. TRIM(n) borra n elementos del final de la tabla. TRIM opera en el tamaño interno, si encuentra un elemento borrado con DELETE lo incluye para ser eliminado de la colección.

DELETE(n) borra el n-ésimo elemento. DELETE (n, m) borra del elemento n al m. Si después de hacer DELETE consultamos si el elemento existe devuelve FALSE.

Al trabajar con tablas anidadas podemos recibir las siguientes excepciones: COLLECTION_IS_NULL, NO_DATA_FOUND, SUBSCRIPT_BEYOND_COUNT y VALUE_ERROR

Diferentes operaciones sobre tablas anidadas

```
DECLARE
TYPE numeros IS TABLE OF NUMBER;
tabla_numeros numeros := numeros();
num NUMBER;
BEGIN
```

```
num := tabla_numeros.COUNT;     --num := 0
FOR i IN 1..10 LOOP
     tabla_numeros.EXTEND;
     tabla_numeros(i) := i;
END LOOP;
num := tabla_numeros.COUNT;     --num := 10
tabla_numeros.DELETE(10);
num := tabla_numeros.LAST;     --num := 9
num := tabla_numeros.FIRST;     --num := 1
tabla_numeros.DELETE(1);
num := tabla_numeros.FIRST;     --num := 2
FOR i IN 1..4 LOOP
     tabla_numeros.DELETE(2*i);
END LOOP;
num := tabla_numeros.COUNT;     --num := 4
num := tabla_numeros.LAST;     --num := 9
...
END;
```

Posibles excepciones de uso:

```
DECLARE
TYPE numeros IS TABLE OF NUMBER;
tabla_num numeros := numeros();
tabla1 numeros;
BEGIN
tabla1(5) := 0;     --lanzaría COLECTION_IS_NULL
tabla_num.EXTEND(5);
tabla_num.DELETE(4);
tabla_num(4) := 3;     --lanzaría NO_DATA_FOUND
tabla_num(6) := 10;     --lanzaría SUBSCRIPT_BEYOND_COUNT
tabla_num(-1) := 0;     --lanzaría SUBSCRIPT_OUTSIDE_LIMIT
tabla_num('y') := 5;--lanzaría VALUE_ERROR
END;
```

3.3. Cursores

El cursor es una estructura que almacena el conjunto devuelto por una consulta a la base de datos.

Oracle usa áreas de trabajo para ejecutar consultas SQL y almacenar la información procesada. Hay 2 clases de cursores: implícitos y explícitos. PS/SQL declara implícitamente un cursor para todas las DML, incluyendo las que devuelven una fila. Para todas las que devuelven más de una, se debe declarar explícitamente un cursor para procesar las filas individualmente.

Cursores implícitos.

Con un cursor implícito no podemos usar las sentencias OPEN, FETCH Y CLOSE para controlarlo, pero podemos usar los atributos del cursor para obtener información sobre las sentencias SQL recientemente ejecutadas.

Atributos de un cursor

Cada cursor tiene 4 atributos, pueden ser usados por PL/SQL pero nen SQL.

- %FOUND. Después de que el cursor esté abierto y antes del primer FETCH %FOUND devuelve NULL. Después de %FOUND devuelve TRUE y si el último FETCH ha devuelto una fila y FALSE en caso contrario. Para cursores implícitos %FOUND devuelve TRUE si un INSERT, UPDATE o DELETE afectan a una o más de una fila, o un SELECT… INTO… devuelve una o más fila, en caso contrario devuelve FALSE.
- %NOTFOUND. Es el caso contrario a %FOUND.
- %ISOPEN. devuelve TRUE si el cursor está abierto y FALSE si está cerrado, para cursores implícitos Oracle cierra automáticamente, así que siempre es FALSE.
- %ROWCOUNT. Para un cursor abierto y antes del primer FETCH %ROWCOUNT devuelve 0. Después de cada FETCH %ROWCOUNT es incrementado y evalúa al número de filas que hemos procesado. Para cursores implícitos %ROWCOUNT evalúa el número de filas afectadas por un INSERT, UODATE, O DELENTE, o el número de filas devueltas por SELECT… INTO.

3.3.1. Cursores explícitos

Si una consulta devuelve múltiples filas debemos declarar explícitamente un cursor para procesarlas. Para ello se les da un nombre y se asocian a una consulta:

```
CURSOR nombre_cursor [(parametro [, parametro] ...)] [RETURN tipo_devuelto] IS sentencia_select;
```

tipo_devuelto debe representar un registro o una fila de una tabla de la bd y parámetro sigue la siguiente sintáxis.

```
parametro := nombre_parametro [IN] tipo_dato [{:= | DEFAULT} expresion]
```

Ejemplos:

```
CURSOR cAgentes IS SELECT * FROM agentes;
CURSOR cFamilias RETURN familias%ROWTYPE IS SELECT * FROM familias WHERE ...
```

Un cursor puede tomar parámetros que pueden aparecer en la consulta asociada como constantes. Los parámetros son de entrega, ya que un cursor no puede devolver valores en los parámetros actuales. A un parámetro de cursor no se puede imponer la restricción NOT NULL

```
CURSOR c1 (cat INTEGER DEFAULT 0) IS SELECT * FROM agentes WHERE categoria = cat;
```

Al abrir un cursor, se ejecuta la consulta asociada y se identifica el conjunto resultado, que serán todas las filas emparejadas al criterio de búsqueda de la consulta.

Un cursor se abre así:

```
OPEN nombre_cursor [(parametro [, parametro] ...)];
```

```
OPEN cAgentes;
OPEN c1(1);
OPEN c1;
```

FETCH devuelve una fila del conjunto resultado. Después de cada FETCH el cursor avanza a la siguiente fila.

```
FETCH cFamilias INTO mi_id, mi_nom, mi_fam, mi_ofi;
```

Por cada valor de columna en la consulta select del cursor, debe haber una variable que corresponda en la lista de variables después de INTO.

Se procesan los cursores por medio de bucles.

```
BEGIN
...
OPEN cFamilias;
LOOP
     FETCH cFamilias INTO mi_id, mi_nom, mi_fam, mi_ofi;
     EXIT WHEN cFamilias%NOTFOUND;
     ...
END LOOP;
CLOSE cFamilias;
...
END;
```

Cuando cerramos el cursor se puede reabrir, pero no se puede realizar ninguna operación con el cursor cerrado, ya que se lanza la excepción INVALID_CURSOR.

Otro bucle consiste en declarar una variable índice definida como %ROWTYPE para el cursor, se abre el cursor, se extraen los valores de cada fila del curso, se almacena en la variable índice y se cierra el cursor.

```
BEGIN
     ...
     FOR cFamilias_rec IN cFamilias LOOP
          --Procesamos las filas accediendo a
          --cFamilias_rec.identificador, cFamilias_rec.nombre,
          --cFamilias_rec.familia, ...
```

```
     END LOOP;
     ...
END;
```

3.3.2. Cursores variables

También podemos definir cursores variables, que son como punteros a cursores, y podemos usarlos para referirnos a cualquier tipo de consulta. Los cursores son estáticos y los cursores variables son dinámicos.

Para declarar un cursor variable:

- Definimos un tipo REF cursor y declaramos una variable de ese tipo.

```
TYPE tipo_cursor IS REF CURSOR RETURN agentes%ROWTYPE;
cAgentes tipo_cursor;
```

- Una vez definido, debemos asociarlo a una consulta, esto se hace en la parte de ejecución y no en la declarativa, y lo hacemos con la sentencia OPEN-FOR utilizando la siguiente sintaxis:

```
TYPE tipo_cursor IS REF CURSOR RETURN agentes%ROWTYPE;
cAgentes tipo_cursor;
```

Los cursores variables no pueden tomar parámetros.

También podemos usar varios OPEN-FOR Para abrir el mismo cursor variable con diferentes consultas y no necesitamos cerrarlo antes de reabrirlo. Cuando reabrimos un cursor variable para una consulta diferente, la consulta previa se pierde.

Una vez abierto, su manejo es idéntico al de un cursor.

```
DECLARE
TYPE cursor_Agentes IS REF CURSOR RETURN agentes%ROWTYPE;
cAgentes cursor_Agentes;
agente cAgentes%ROWTYPE;
BEGIN
...
OPEN cAgentes FOR SELECT * FROM agentes WHERE oficina = 1;
LOOP
     FETCH cAgentes INTO agente;
     EXIT WHEN cAgentes%NOTFOUND;
     ...
END LOOP;
CLOSE cAgentes;
...
END;
```

También se puede utilizar el bucle FOR con cursores variables definidos dentro del mismo bucle

```
for va_cursor in (select ...) loop
...
end loop;
```

4. Abstracción en PL/SQL

PL/SQL permite definir funciones y procedimientos, además podemos agrupar todas las que tengan relación en paquetes. También permite utilización de objetos.

4.1. Subprogramas

Son bloques de código PL/SQL, referenciados bajo un nombre y crean una acción determinada. Le podemos pasar parámetros y los podemos invocar.

Suelen estar almacenados en la base de datos o encerrados en otros bloques. Si el programa está en la bd, podemos invocarlo con los permisos suficientes, y si está encerrado en el bloque podremos invocarlo si tenemos visibilidad sobre el mismo.

Existen dos subprogramas: funciones (devuelven valor) y procedimientos (no devuelven valor)

Sintaxis de funciones:

```
FUNCTION nombre [(parametro [, parametro] ...)]
   RETURN tipo_dato IS
   [declaraciones_locales]
BEGIN
   sentencias_ejecutables
[EXCEPTION
   manejadores_de_excepciones]
END [nombre];
```

Sintaxis de procedimientos

```
PROCEDURE nombre [( parametro [, parametro] ... )] IS
   [declaraciones_locales]
BEGIN
   sentencias_ejecutables
[EXCEPTION manejadores_de_excepciones]
END [nombre];
```

```
parametro := nombre_parametro [IN|OUT|IN OUT] tipo_dato [{:=|DEFAULT} expresion]
```

- No podemos imponer restricción NOT NULL a un parámetro.
- No podemos especificar una restricción del tipo.
- Una función debe acabar con la sentencia RETURN

En Oracle, cualquier identificador se debe declarar antes de usarse y eso también pasa con los subprogramas:

```
DECLARE
hijos NUMBER;
FUNCTION hijos_familia( id_familia NUMBER )
     RETURN NUMBER IS
     hijos NUMBER;
BEGIN
     SELECT COUNT(*) INTO hijos FROM agentes
          WHERE familia = id_familia;
     RETURN hijos;
END hijos_familia;
BEGIN
...
END;
```

Si queremos definir subprogramas en orden alfabético o lógico, o necesitamos definir subprogramas mutuamente recursivos debemos usar la definición hacia delante para evitar errores de compilación.

```
DECLARE
     PROCEDURE calculo(...);          --declaración hacia delante
     --Definimos subprogramas agrupados lógicamente
     PROCEDURE inicio(...) IS
     BEGIN
          ...
          calculo(...);
          ...
     END;
     ...
BEGIN
     ..
```

4.1.1. Almacenar subprogramas en la base de datos

Para almacenar subprogramas en la base de datos utilizamos la misma sintaxis para declararlo, anteponiendo CREATE [OR REPLACE] a PROCEDURE o FUNCTION.

y finalizamos el subprograma con una línea que contendrá el carácter / para indicar que termina ahí. Con REPLACE indicamos que si el subprograma ya existe, sea reemplazado.

```
CREATE OR REPLACE FUNCTION hijos_familia(id_familia NUMBER)
   RETURN NUMBER IS hijos NUMBER;
BEGIN
  SELECT COUNT(*) INTO hijos FROM agentes
       WHERE familia = id_familia;
RETURN hijos;
END;
/
```

Cuando se almacenan subprogramas en la bd, no se pueden utilizar declaraciones hacia delante. Cualquier subprograma almacenado debe conocer todos los subprogramas que utiliza. Invocamos un subprograma con la siguiente sintaxis:

```
nombre_procedimiento [(parametro [,parametro] ...)];
variable := nombre_funcion [(parametro [, parametro] ...)];
BEGIN
...
  hijos := hijos_familia(10);
...
END;
```

Si el subprograma está almacenado en la bd y queremos invocarlo desde SQL Plus usamos la sintaxis:

```
EXECUTE nombre_procedimiento [(parametros)];
EXECUTE :variable_sql := nombre_funcion [(parametros)];
```

Los subprogramas de las bd son compilados antes, si hay algún error se informa de los mismos y se deben corregir creándolo de nuevo por medio de la cláusula OR REPLACE antes de que sea utilizado.

Hay varias vistas del diccionario de datos que nos ayudan a llevar control de subprogramas, tanto para ver su código como para los errores de compilación. También hay comandos que nos ayudan a hacer lo mismo de forma menos engorrosa. El comando show errors tras compilar muestra los errores que hay.

Vistas y comandos asociados a los subprogramas.

Información almacenada.	Vista del diccionario.	Comando.
Código fuente.	USER_SOURCE	DESCRIBE
Errores de compilación.	USER_ERRORS	SHOW ERRORS
Ocupación de memoria.	USER_OBJECT_SIZE	

También existe la vista USER_OBJECTS de la cual podemos obtener los nombres de todos los subprogramas almacenados.

4.1.2. Parámetros de los subprogramas

Las variables pasadas como parámetros a un subprograma son llamadas parámetros actuales. Las referenciadas en la especificación del subprograma como parámetros son llamadas parámetros formales.

Cuando llamamos a un subprograma, los parámetros actuales podemos escribirlos utilizando notación posicional o nombrada. La asociación entre parámetros actuales y formales las hacemos por posición o por nombre.

En la notación posicional, el primer parámetro actual se asocia con el primer formal, y así con el resto. En la notación nombrada usamos el operador ⇒ para asociarlos. También podemos usar notación mixta.

Los parámetros pueden ser de entrada al subprograma, de salida, o de entrada y salida. Por defecto serán de entrada, si es de salida o de entrada y salida el parámetro actual debe ser una variable.

Un parámetro de entrada permite que le pasemos valores al subprograma y no puede ser modificado en el cuerpo del subprograma. El parámetro actual pasado a un subprograma como parámetro formal de entrada puede ser una constante o variable.

Un parámetro de salida permite devolver valores y dentro del subprograma actúa como variable no inicializada. El parámetro formal debe ser siempre una variable.

Un parámetro de entrada-salida se utiliza para pasar valores al subprograma y/o recibirlos, por lo que un parámetro formal que actúe como parámetro actual siempre debe ser una variable.

Los parámetros de entrada los podemos inicializar a un valor por defecto, si un subprograma tiene un parámetro inicializado con un valor por defecto podemos invocarlo prescindiendo del parámetro y aceptando el valor por defecto o pasando el parámetro y sobreescribiendo el valor por defecto. Si queremos prescindir de un parámetro colocado entre medias de otros, debemos usar notación nombrada, o si los parámetros restantes también tienen valor por defecto, omitirlos todos.

4.1.3. Sobrecarga de subprogramas y recursividad

PL/SQL nos ofrece la posibilidad de sobrecargar funciones o procedimientos: llamar con el mismo nombre subprogramas que realizan el mismo cometido aceptando distinto número y/o tipo de parámetros. No se pueden sobrecargar subprogramas con el mismo número de parámetros aunque sus tipos sean diferentes pero de la misma familia o subtipos de la misma familia.

4.2. Paquetes

Un paquete es un objeto que agrupa tipos, elementos y subprogramas. Tienen dos partes. Especificación y cuerpo, aunque el cuerpo puede ser no necesario.

En la especificación es declara la interfaz del paquete con nuestra aplicación y en el cuerpo implementamos esa interfaz.

```
CREATE [OR REPLACE] PACKAGE nombre AS
     [declaraciones públicas y especificación  subprogramas]
END [nombre]
CREATE [OR REPLACE] PACKAGE BODY nombre AS
     [declaraciones privadas y cuerpo  subprogramas especificados]
[BEGIN
     sentencias inicialización]
END [nombre];
```

4.2.1. Ejemplos de utilización del paquete DBMS_OUTPUT

Oracle suministra un paquete público con el que podemos enviar mensajes desde subprogramas almacenados, paquetes y disparadores, colocarlos en un buffer y leerlos desde subprogramas almacenados, paquetes o disparadores.

SQL Plus permite visualizar los mensajes del buffer por medio de SET SERVEROUTPUT ON. Es fundamental su utilización para depurar nuestros subprogramas.

Los subprogramas que nos suministra este paquete son:

- Habilita las llamadas a los demás subprogramas. No es necesario cuando está activada la opción SERVEROUTPUT. Podemos pasarle un parámetro indicando el tamaño del buffer

```
ENABLE
ENABLE(buffer_size IN INTEGER DEFAULT 2000);
```

- Deshabilita las llamadas a los subprogramas y purga el buffer. Como con ENABLE, no es necesario si estamos usando la opción SERVEROUTPUT.

```
DISABLE
DISABLE();
```

Coloca elementos en el buffer, para convertirlos a VARCHAR2.

```
PUT
PUT(item IN NUMBER);
PUT(item IN VARCHAR2);
PUT(item IN DATE);
```

Coloca elementos en el buffer y los termina con un salto de línea.

```
PUT_LINE
PUT_LINE(item IN NUMBER);
PUT_LINE(item IN VARCHAR2);
PUT_LINE(item IN DATE);
```

Coloca un salto de línea en el buffer utilizado cuando componemos una línea usando varios PUT.

```
NEW_LINE
NEW_LINE();
```

lee una línea del buffer colocándola en el parámetro line y obviando el salto de línea. El parámetro status devuelve 0 si nos hemos traído alguna línea y 1 en caso contrario.

```
GET_LINE
GET_LINE(line OUT VARCHAR2, status OUT VARCHAR2);
```

Intenta leer el número de líneas indicado en numlines, una vez ejecutado, numlines contendrá el número de líneas que se ha traído. Las líneas traídas las coloca en el parámetro lines de tipo CHARARR, tipo definido en el paquete DBMS_OUTPUT como una tabla de VARCHAR2(255).

En el siguiente ejemplo creamos un procedimiento que visualice todos los agentes, su nombre, nombre de la familia y/o oficina a la que pertenece.

```
En el siguiente ejemplo creamos un procedimiento que visualice todos los agentes, su nombre, nombre de la familia y/o oficina a la que
```

Para ejecutarlo en SQL Plus se deben ejecutar las siguientes sentencias:

```
SQL>SET SERVEROUTPUT ON;
SQL>EXEC lista_agentes;
```

4.3. Objetos

En PL/SQL las variables son los atributos y los subprogramas son métodos de los objetos. Podemos pensar en un tipo de objeto como una entidad que posee unos atributos y un comportamiento.

- Cuando creamos un tipo de objeto, creamos una entidad abstracta que especifica los atributos que tendrán los objetos de ese tipo y define su comportamiento.
- Cuando lo instanciamos, particularizamos la entidad abstracta en una particular, con los atributos del objeto con sus propios valores y comportamiento.

Los objetos tienen 2 partes. Especificación y cuerpo. La especificación declara primero los atributos y después los métodos. El cuerpo implementa la parte de especificación.

Todos los atributos son públicos (visibles). No podemos declarar atributos en el cuerpo, pero podemos declarar subprogramas locales que serán visibles en el cuerpo del objeto y que nos ayudan a implementar nuestros métodos.

Los atributos pueden ser de cualquier tipo excepto:

LONG y LONG RAW.
NCHAR, NCLOB y NVARCHAR2.
MLSLABEL y ROWID.
Tipos específicos de PL/SQL: BINARY_INTEGER, BOOLEAN, PLS_INTEGER, RECORD, REF CURSOR, %TYPE y %ROWTYPE.
Tipos definidos dentro de un paquete PL/SQL.

No podemos inicializar un atributo en la declaración y tampoco asignarle la restricción NOT NULL.

Un método es un subprograma declarado en la parte de especificación de un tipo de objeto por medio de MEMBER. No puede llamarse igual que el tipo de objeto o que cualquier atributo, para cada método en la especificación, debe haber un método implementado en en el cuerpo con la misma cabecera.

Todos los métodos en un tipo de objeto aceptan de primer parámetro una instancia de su tipo. Este parámetros es SELF y siempre es accesible a un método. Se se declara explícitamente, por defecto será IN para las funciones e IN OUT para los procedimientos.

Los métodos se pueden sobrecargar, no podemos sobrecargarlos si los parámetros formales solo difieren en el modo o pertenecen a la misma familia. Tampoco se puede sobrecargar una función miembro si solo difiere del tipo devuelto.

Una vez creado el objeto, se puede utilizar en cualquier declaración. Un objeto declarado sigue las mismas normas que cualquier variable.

Un objeto declarado es NULL, dejará de serlo cuando se inicie por medio de su constructor o le asignemos otro. Intentar acceder a atributos de un objeto NULL lanzará una excepción ACCES_INTO_NULL.

Todos los objetos tienen constructores por defecto con el mismo nombre del tipo de objeto y acepta tantos parámetros como atributos del tipo de objeto y con el mismo tipo. PL/SQL no llama implícitamente a los constructores, debemos hacerlo nosotros:

```
DECLARE
     familia1 Familia;
BEGIN
     ...
     familia1 := Familia( 10, 'Fam10', 1, NULL );
     ...
END;
```

Un tipo de objeto puede tener a otro tipo de objeto entre sus atributos. El tipo de objeto que hace de atributo debe ser conocido por Oracle. Si 2 tipos de objetos son mutuamente dependientes, podemos usar una declaración hacia delante para evitar errores de compilación.

Ejemplo:

```
CREATE OBJECT Oficina;     --Definición hacia delante
CREATE OBJECT Familia AS OBJECT (
     identificador     NUMBER,
     nombre          VARCHAR2(20),
     familia_     Familia,
     oficina_     Oficina,
     ...
);
CREATE OBJECT Agente AS OBJECT (
     identificador     NUMBER,
     nombre          VARCHAR2(20),
     familia_     Familia,
     oficina_     Oficina,
     ...
);
CREATE OBJECT Oficina AS OBJECT (
     identificador     NUMBER,
     nombre          VARCHAR2(20),
     jefe          Agente,
     ...
);
```

4.3.1. Objetos. Funciones mapa y funciones de orden

Los tipos de objetos no tienen orden predefinido, no pueden ser comparados ni ordenados. Se puede definir el orden que seguirá un tipo de objeto por medio de funciones mapa y funciones orden.

Una función miembro mapa es una función sin parámetros que devuelve un tipo de dato DATE, NUMBER o VARCHAR2, siendo similar a una función hash. Se definen anteponiendo la palabra clave MAP. Solo puede haber una para cada tipo de objeto.

```
CREATE TYPE Familia AS OBJECT (
identificador      NUMBER,
nombre           VARCHAR2(20),
familia_          NUMBER,
oficina_          NUMBER,
MAP MEMBER FUNCTION orden RETURN NUMBER,
...
);

CREATE TYPE BODY Familia AS
MAP MEMBER FUNCTION orden RETURN NUMBER IS
```

```
BEGIN
     RETURN identificador;
END;
...
END;
```

Una función miembro de orden es una función que acepta un parámetro del mismo tipo del tipo de objeto, y devuelve un número negativo si el objeto pasado es menor, cero si son iguales y uno si el objeto pasado es mayor.

```
CREATE TYPE Oficina AS OBJECT (
     identificador          NUMBER,
     nombre               VARCHAR2(20),
     ...
     ORDER MEMBER FUNCTION igual ( ofi Oficina ) RETURN INTEGER,
     ...
);

CREATE TYPE BODY Oficina AS
     ORDER MEMBER FUNCTION igual ( ofi Oficina ) RETURN INTEGER IS
     BEGIN
          IF (identificador < ofi.identificador) THEN
               RETURN –1;
          ELSIF (identificador = ofi.identificador) THEN
               RETURN 0;
          ELSE
               RETURN 1;
          END IF;
     END;
     ...
END;
```

5. Disparadores

PL/SQL ofrece para programar nuestra base de datos y mantener la integridad y seguridad, los disparadores o triggers.

Un disparador es un procedimiento que es ejecutado cuando se realiza alguna sentencia sobre la BD, bajo unas circunstancias a la hora de definirlo.

Pueden ser de tres tipos: de tablas, de sustitución o de sistema.

Puede ser lanzado antes (BEFORE) o después (AFTER) de realizar la operación que lo lanza.

Disparadores de tablas

Previenen transacciones erróneas y permiten implementar restricciones de integridad o seguridad, o automatizar procesos. Son los más utilizados.

Se ejecutan cuando se realiza alguna sentencia DML sobre una tabla, se puede lanzar al insertar, actualizar o borrar de una tabla. Podemos tener disparadores INSERT, UPDATE, DELETE o mezclados.

Puede ser lanzado una vez por sentencia (FOR STATEMENT) o una por cada fila a la que afecta (FOR EACH ROW).

De sustitución

No se ejecutan ni antes ni después, sino en lugar de (INSTEAD OF), solo se pueden asociar a vistas o a nivel de fila.

De Sistema

Se ejecuta cuando se produce una determinada operación sobre la BD, crear una tabla, conexión de usuario, etc. Se pueden detectar eventos como por ejemplo: LOGON, LOGOFF, CREATE, DROP, etc... y puede ser tanto BEFORE como AFTER.

Un disparador es simplemente un procedimiento que se puede utilizar para

- Llevar auditorías sobre la historia de los datos de nuestra BD.
- Garantizar complejas reglas de integridad.
- Automatizar la generación de valores derivados de consultas.
- Etc.

Cuando diseñamos un disparador debemos tener en cuenta que:

- No debemos definir disparadores que dupliquen la funcionalidad que ya incorpora Oracle.

- Debemos limitar el tamaño de nuestros disparadores, y si estos son muy grandes codificarlos por medio de subprogramas que sean llamados desde el disparador.
- Cuidar la creación de disparadores recursivos.

5.1. Definición de disparadores de tablas

Para definir un disparador debemos indicar si se lanza antes o después de la ejecución de la sentencia que lo lanza. Si se lanza una vez por sentencia o una vez por fila y si será lanzado cuando se inserta, actualiza o borra.

Sintaxis:

```
CREATE [OR REPLACE] TRIGGER nombre
momento acontecimiento ON tabla
[[REFERENCING (old AS alias_old|new AS alias_new)
FOR EACH ROW
[WHEN condicion]]
bloque_PL/SQL;
```

el **nombre** es el nombre del disparador, **momento** es BEFORE O AFTER acontecimiento es la acción: INSERT, UPDATE, DELETE. REFERENCING nos permite asignar alias a los valores NEW y/o OLD de las filas afectadas yWHEN nos permite indicar al disparador que solo sea lanzado cuando sea true cierta condición para cada fila (ambos solo pueden realizarse con disparadores para filas).

Un disparador de fila puede acceder a valores antiguos y nuevos de la fila afectada, referenciados como :old y :new, asignándole un alias a cada uno.

INSERT. Tiene sentido solo el valor nuevo.

UPDATE. El valor antiguo contiene la fila antes de actualizar y el valor nuevo contiene la fila a actualizar.

DELETE. Para un disparador lanzado al borrar solo tiene sentido el valor antiguo.

En el cuerpo de un disparador también podemos acceder a unos predicados que nos dicen el tipo que se está llevando a cabo: INSERTING, UPDATING, DELETING.

Un disparador de fila no puede acceder a la tabla asociada. Se dice que la tabla está mutando, su un disparador es lanzado en cascada por otro, éste no podrá acceder a ninguna de las tablas asociadas y así recursivamente.

```
CREATE TRIGGER prueba BEFORE UPDATE ON agentes
FOR EACH ROW
BEGIN
     ...
     SELECT identificador FROM agentes WHERE ...
/*devolvería el error ORA-04091: table AGENTES is mutating, trigger/function may not see it*/
     ...
END;
/
```

Si tenemos varios disparadores de una misma tabla el orden de ejecución es:

- Triggers before de sentencia.
- Triggers before de fila.
- Triggers after de fila.
- Triggers after de sentencia.

Existe una vista del diccionario de datos con información sobre disparadores: USER_TRIGGERS

```
SQL>DESC USER_TRIGGERS;
Name                           Null?    Type
------------------------------ -------- ----
TRIGGER_NAME                   NOT NULL VARCHAR2(30)
TRIGGER_TYPE                            VARCHAR2(16)
TRIGGERING_EVENT                        VARCHAR2(26)
TABLE_OWNER                    NOT NULL VARCHAR2(30)
TABLE_NAME                     NOT NULL VARCHAR2(30)
REFERENCING_NAMES                       VARCHAR2(87)
WHEN_CLAUSE                             VARCHAR2(4000)
STATUS                                  VARCHAR2(8)
DESCRIPTION                             VARCHAR2(4000)
TRIGGER_BODY                            LONG
```

5.2. Ejemplos de disparadores

Ejemplo 1:

Como un agente debe pertenecer a una familia o una oficina pero no puede pertenecer a una familia y a una oficina a la vez, deberemos implementar un disparador para llevar a cabo esta restricción que Oracle no nos permite definir desde el DDL.

Para este cometido definiremos un disparador de fila que saltará antes de que insertemos o actualicemos una fila en la tabla agentes, cuyo código podría ser el siguiente:

```
CREATE OR REPLACE TRIGGER integridad_agentes
BEFORE INSERT OR UPDATE ON agentes
FOR EACH ROW
BEGIN
     IF (:new.familia IS NULL and :new.oficina IS NULL) THEN -- Si los dos valores son nulos
          RAISE_APPLICATION_ERROR(-20201, 'Un agente no puede ser huérfano');-- Lanzauna excepción
     ELSIF (:new.familia IS NOT NULL and :new.oficina IS NOT NULL) THEN -- Si los dos tienen valores
          RAISE_APPLICATION_ERROR(-20202, 'Un agente no puede tener dos padres');
     END IF;
END;
/
```

Ejemplo 2:

Supongamos que tenemos una tabla de históricos para agentes que nos permita auditar las familias y oficinas por la que ha ido pasando un agente. La tabla tiene la fecha de inicio y la fecha de finalización del agente en esa familia u oficina, el identificador del agente, el nombre del agente, el nombre de la familia y el nombre de la oficina. Queremos hacer un disparador que inserte en esa tabla.

Para llevar a cabo esta tarea definiremos un disparador de fila que saltará después de insertar, actualizar o borrar una fila en la tabla agentes, cuyo código podría ser el siguiente:

```
CREATE OR REPLACE TRIGGER historico_agentes
AFTER INSERT OR UPDATE OR DELETE ON agentes
FOR EACH ROW
DECLARE
     oficina VARCHAR2(40);
     familia VARCHAR2(40);
     ahora DATE := sysdate;
BEGIN
     IF INSERTING THEN
          IF (:new.familia IS NOT NULL) THEN
               SELECT nombre INTO familia FROM familias WHERE identificador = :new.familia;
               oficina := NULL;
          ELSIF
               SELECT nombre INTO oficina FROM oficinas WHERE identificador = :new.oficina;
               familia := NULL;
          END IF;
          INSERT INTO histagentes VALUES (ahora, NULL, :new.identificador, :new.nombre, familia, oficina);
          COMMIT;
     ELSIF UPDATING THEN
          UPDATE histagentes SET fecha_hasta = ahora WHERE identificador = :old.identificador and fecha_hasta IS NULL;
          IF (:new.familia IS NOT NULL) THEN
               SELECT nombre INTO familia FROM familias WHERE identificador = :new.familia;
               oficina := NULL;
          ELSE
               SELECT nombre INTO oficina FROM oficinas WHERE identificador = :new.oficina;
               familia := NULL;
          END IF;
          INSERT INTO histagentes VALUES (ahora, NULL, :new.identificador, :new.identificador, familia, oficina);
          COMMIT;
     ELSE
          UPDATE histagentes SET fecha_hasta = ahora WHERE identificador = :old.identificador and fecha_hasta IS NULL;
          COMMIT;
     END IF;
END;
/
```

Ejemplo 3

Queremos realizar un disparador que no nos permita llevar a cabo operaciones con familias si no estamos en la jornada laboral.

```
CREATE OR REPLACE TRIGGER jornada_familias
BEFORE INSERT OR DELETE OR UPDATE ON familias
DECLARE
     ahora DATE := sysdate;
BEGIN
     IF (TO_CHAR(ahora, 'DY') = 'SAT' OR TO_CHAR(ahora, 'DY') = 'SUN') THEN
          RAISE_APPLICATION_ERROR(-20301, 'No podemos manipular familias en fines de semana');
     END IF;
     IF (TO_CHAR(ahora, 'HH24') < 8 OR TO_CHAR(ahora, 'HH24') > 18) THEN
          RAISE_APPLICATION_ERROR(-20302, 'No podemos manipular familias fuera del horario de trabajo');
     END IF;
END;
/
```

6. Interfaces de programación de aplicaciones para lenguajes externos

para acceder a bases de datos desde un lenguaje de programación externo se utilizan APIs disponibles para distintos lenguajes de programación. Podemos encontrar en estos enlaces más información:

https://www.oracle.com/java/technologies/javase/javase-tech-database.html

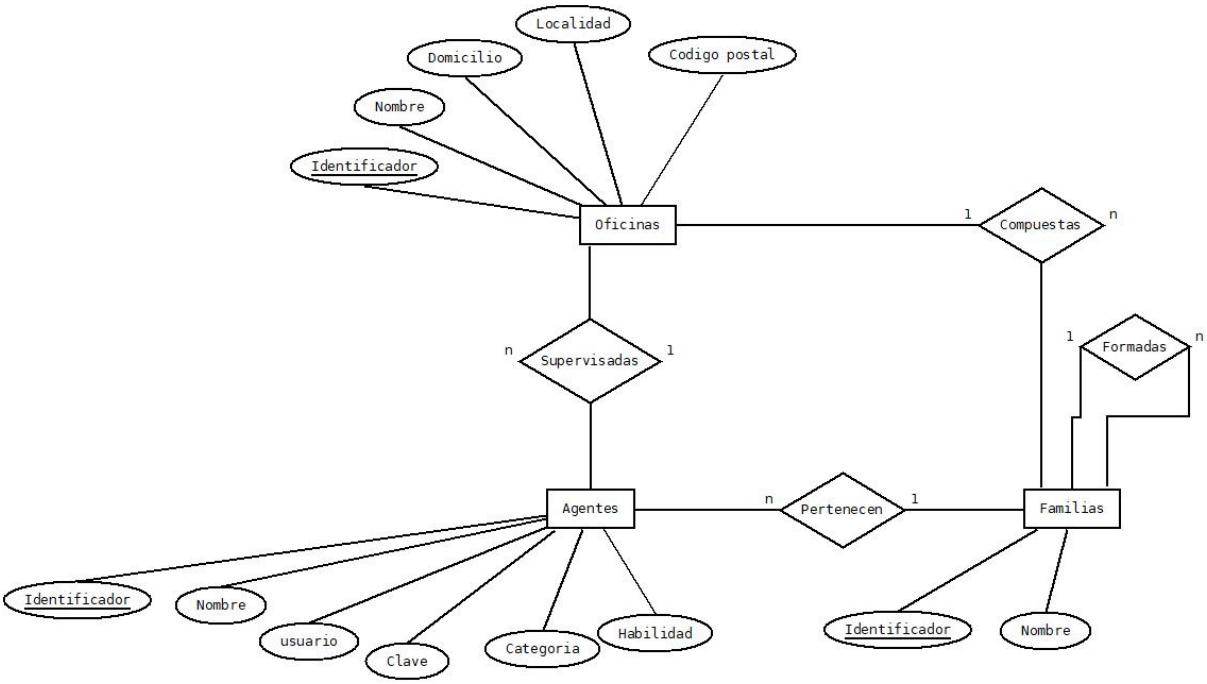
https://en.wikipedia.org/wiki/Open_Database_Connectivity

Anexo 1. Caso de estudio.

Una empresa de telefonía tiene sus centros de llamadas distribuidos por la geografía española en diferentes oficinas. Estas oficinas están jerarquizadas en familias de agentes telefónicos. Cada familia, por tanto, podrá contener agentes u otras familias. Los agentes telefónicos, según su categoría, además se encargarán de supervisar el trabajo de todos los agentes de una oficina o de coordinar el trabajo de los agentes de una familia dada. El único agente que pertenecerá directamente a una oficina y que no formará parte de ninguna familia será el supervisor de dicha oficina, cuya categoría es la 2. Los coordinadores de las familias deben pertenecer a dicha familia y su categoría será 1 (no todas las familias tienen por qué tener un coordinador y dependerá del tamaño de la oficina, ya que de ese trabajo también se puede encargar el supervisor de la oficina). Los demás agentes deberán pertenecer a una familia, su categoría será 0 y serán los que principalmente se ocupen de atender las llamadas.

- De los agentes queremos conocer su nombre, su clave y contraseña para entrar al sistema, su categoría y su habilidad que será un número entre 0 y 9 indicando su habilidad para atender llamadas.
- Para las familias sólo nos interesa conocer su nombre.
- Finalmente, para las oficinas queremos saber su nombre, domicilio, localidad y código postal de la misma.

Modelo entidad relación:



El **Modelo Relacional** resultante sería:

OFICINAS (identificador, nombre, domicilio, localidad, codigo_postal)

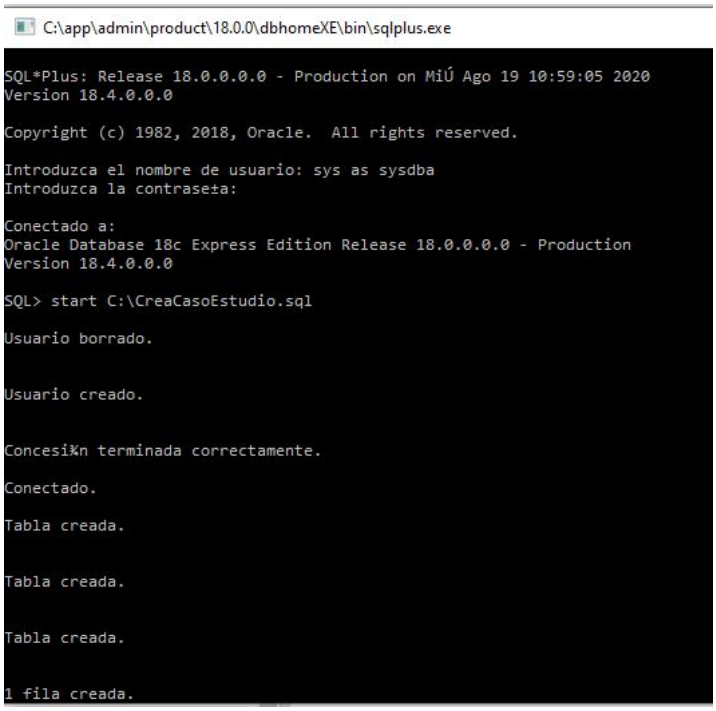
FAMILIAS (identificador, nombre, familia (fk), oficina (fk))

AGENTES (identificador, nombre, usuario, clave, habilidad, categoría, familia (fk), oficina (fk))

De este modelo de datos surgen tres tablas, que puedes crear en Oracle con el script del siguiente enlace:

El script crea un usuario llamado c##agencia con clave agencia. Para que puedas ejecutarlo y comenzar de cero, cuantas veces quieras, el script elimina el usuario c##agencia y sus tablas antes de volver a crearlo.

Conecta como administrador con SYS as SYSDBA y ejecútalo anteponiendo el símbolo @ o la palabra start antes del nombre del script.



Habilitando la Salida/OUTPUT en Bloques PL/SQL.

PL/SQL no proporciona funcionalidad de entrada o salida directamente siendo necesario utilizar paquetes predefinidos de **Oracle** para tales fines. Para generar una salida debes realizar lo siguiente :

1. Ejecutar el siguiente comando tanto en **SQL*Plus** como en cualquier otro entorno:

SET SERVEROUTPUT ON

2. En el bloque **PL/SQL**, hay que utilizar el procedimiento PUT LINE del paquete DMBS_OUTPUT para mostrar la salida. El valor a mostrar en pantalla se pasará como argumento del procedimiento.

Ejecuta el siguiente código desde SQLPlus y desde SQLDeveloper para comprobar su funcionamiento.

```
SET SERVEROUTPUT ON
BEGIN
    DBMS_OUTPUT.PUT_LINE('Hola mundo');
END;
```

Anexo II - Excepciones predefinidas en Oracle

Excepción.	SQLCODE	Lanzada cuando ...
ACCES_INTO_NULL	-6530	Intentamos asignar valor a atributos de objetos no inicializados.
COLECTION_IS_NULL	-6531	Intentamos asignar valor a elementos de colecciones no inicializadas, o acceder a métodos distintos de EXISTS.
CURSOR_ALREADY_OPEN	-6511	Intentamos abrir un cursor ya abierto.
DUP_VAL_ON_INDEX	-1	Índice único violado.
INVALID_CURSOR	-1001	Intentamos hacer una operación con un cursor que no está abierto.
INVALID_NUMBER	-1722	Conversión de cadena a número falla.
LOGIN_DENIED	-1403	El usuario y/o contraseña para conectarnos a Oracle no es válido.
NO_DATA_FOUND	+100	Una sentencia SELECT no devuelve valores, o intentamos acceder a un elemento borrado de una tabla anidada.
NOT_LOGGED_ON	-1012	No estamos conectados a Oracle.
PROGRAM_ERROR	-6501	Ha ocurrido un error interno en PL/SQL.
ROWTYPE_MISMATCH	-6504	Diferentes tipos en la asignación de 2 cursores.
STORAGE_ERROR	-6500	Memoria corrupta.
SUBSCRIPT_BEYOND_COUNT	-6533	El índice al que intentamos acceder en una colección sobrepasa su límite superior.
SUBSCRIPT_OUTSIDE_LIMIT	-6532	Intentamos acceder a un rango no válido dentro de una colección (-1 por ejemplo).
TIMEOUT_ON_RESOURCE	-51	Un timeout ocurre mientras Oracle espera por un recurso.
TOO_MANY_ROWS	-1422	Una sentencia SELECT...INTO... devuelve más de una fila.
VALUE_ERROR	-6502	Ocurre un error de conversión, aritmético, de truncado o de restricción de tamaño.
ZERO_DIVIDE	-1476	Intentamos dividir un número por 0.

Anexo III - Evaluación de los atributos de un cursor explícito

Operación realizada.	%FOUND	%NOTFOUND	%ISOPEN	%ROWCOUNT
Antes del OPEN	Excepción.	Excepción.	FALSE	Excepción.
Después del OPEN	NULL	NULL	TRUE	0
Antes del primer FETCH	NULL	NULL	TRUE	0
Después del primer FETCH	TRUE	FALSE	TRUE	1
Antes de los siguientes FETCH	TRUE	FALSE	TRUE	1
Después de los siguientes FETCH	TRUE	FALSE	TRUE	Depende datos.
Antes del último FETCH	TRUE	FALSE	TRUE	Depende datos.
Después del último FETCH	FALSE	TRUE	TRUE	Depende datos.
Antes del CLOSE	FALSE	TRUE	TRUE	Depende datos.
Después del CLOSE	Excepción.	Excepción.	FALSE	Excepción.

Anexo IV - Paso de parámetros a subprogramas

Notación mixta

```
DECLARE
     PROCEDURE prueba( formal1 NUMBER, formal2 VARCHAR2) IS
     BEGIN
          ...
     END;
     actual1 NUMBER;
     actual2 VARCHAR2;
BEGIN
     ...
     prueba(actual1, actual2);              --posicional
     prueba(formal2=>actual2,formal1=>actual1);     --nombrada
     prueba(actual1, formal2=>actual2);          --mixta
END;
```

Parámetros de entrada

```
FUNCTION categoria( id_agente IN NUMBER )
RETURN NUMBER IS
     cat NUMBER;
BEGIN
     ...
     SELECT categoria INTO cat FROM agentes
WHERE identificador = id_agente;
RETURN cat;
EXCEPTION
     WHEN NO_DATA_FOUND THEN
          id_agente := -1; --ilegal, parámetro de entrada
END;
```

Parámetros de salida

```
PROCEDURE nombre( id_agente NUMBER, nombre OUT VARCHAR2) IS
BEGIN
     IF (nombre = 'LUIS') THEN     --error de sintaxis
     END IF;
     ...
END;
```

Parámetros con valor por defecto de los que podemos prescindir.

```
DECLARE
     SUBTYPE familia IS familias%ROWTYPE;
     SUBTYPE agente IS agentes%ROWTYPE;
     SUBTYPE tabla_agentes IS TABLE OF agente;
     familia1 familia;
     familia2 familia;
     hijos_fam tabla_agentes;
     FUNCTION inserta_familia( mi_familia familia,
          mis_agentes tabla_agentes := tabla_agentes() )
RETURN NUMBER IS
     BEGIN
          INSERT INTO familias VALUES (mi_familia);
          FOR i IN 1..mis_agentes.COUNT LOOP
               IF (mis_agentes(i).oficina IS NOT NULL) or (mis_agentes(i).familia ¡= mi_familia.identificador) THEN
                    ROLLBACK;
                    RETURN –1;
               END IF;
               INSERT INTO agentes VALUES (mis_agentes(i));
          END LOOP;
          COMMIT;
          RETURN 0;
     EXCEPTION
          WHEN DUP_VAL_ON_INDEX THEN
               ROLLABACK;
               RETURN –1;
          WHEN OTHERS THEN
               ROLLBACK;
               RETURN SQLCODE;
     END inserta_familia;
BEGIN
     ...
     resultado := inserta_familia( familia1 );
     ...
     resultado := inserta_familia( familia2, hijos_fam2 );
     ...
END;
```

Anexo V - Sobrecarga de subprogramas

```
DECLARE
     TYPE agente IS agentes%ROWTYPE;
     TYPE familia IS familias%ROWTYPE;
     TYPE tAgentes IS TABLE OF agente;
     TYPE tFamilias IS TABLE OF familia;

     FUNCTION inserta_familia( mi_familia familia )
     RETURN NUMBER IS
     BEGIN
          INSERT INTO familias VALUES (mi_familia.identificador, mi_familia.nombre, mi_familia.familia, mi_familia.oficina );
          COMMIT;
          RETURN 0;
     EXCEPTION
          WHEN DUP_VAL_ON_INDEX THEN
               ROLLBACK;
               RETURN -1;
          WHEN OTHERS THEN
               ROLLBACK;
               RETURN SQLCODE;
     END inserta_familia;

     FUNCTION inserta_familia( mi_familia familia, hijas tFamilias )
     RETURN NUMBER IS
     BEGIN
          INSERT INTO familias VALUES (mi_famila.identificador, mi_familia.nombre, mi_familia.familia, mi_familia.oficina );
          IF (hijas IS NOT NULL) THEN
               FOR i IN 1..hijas.COUNT LOOP
                    IF (hijas(i).oficina IS NOT NULL) or (hijas(i).familia ¡= mi_familia.identificador) THEN
                         ROLLBACK;
                         RETURN -1;
                    END IF;
                    INSERT INTO familias VALUES (hijas(i).identificador, hijas(i).nombre, hijas(i).familia, hijas(i).oficina );
               END LOOP;
          END IF;
          COMMIT;
          RETURN 0;
     EXCEPTION
          WHEN DUP_VAL_ON_INDEX THEN
               ROLLBACK;
               RETURN -1;
          WHEN OTHERS THEN
               ROLLBACK;
               RETURN -1;
     END inserta_familia;

     FUNCTION inserta_familia( mi_familia familia, hijos tAgentes )
     RETURN NUMBER IS
     BEGIN
          INSERT INTO familias VALUES (mi_familia.identificador, mi_familia.nombre, mi_familia.familia, mi_familia.oficina );
          IF (hijos IS NOT NULL) THEN
               FOR i IN 1..hijos.COUNT LOOP
                    IF (hijos(i).oficina IS NOT NULL) or (hijos(i).familia ¡= mi_familia.identificador) THEN
                         ROLLBACK;
                         RETURN -1;
                    END IF;
                    INSERT INTO agentes VALUES (hijos(i).identificador, hijos(i).nombre, hijos(i).usuario, hijos(i).clave, hijos(i).ha
               END LOOP;
          END IF;
          COMMIT;
          RETURN 0;
     EXCEPTION
          WHEN DUP_VAL_ON_INDEX THEN
               ROLLBACK;
               RETURN -1;
          WHEN OTHERS THEN
               ROLLBACK;
               RETURN -1;
     END inserta_familias;

     mi_familia familia;
     mi_familia1 familia;
     familias_hijas tFamilias;
     mi_familia2 familia;
     hijos tAgentes;
BEGIN
     ...
     resultado := inserta_familia(mi_familia);
     ...
     resultado := inserta_familia(mi_familia1, familias_hijas);
     ...
     resultado := inserta_familia(mi_familia2, hijos);
```

```
     ...
END;
```

Anexo VI. Ejemplo de recursividad

```
DECLARE
     TYPE agente IS agentes%ROWTYPE;
     TYPE tAgentes IS TABLE OF agente;
     hijos10 tAgentes;
     PROCEDURE dame_hijos( id_familia NUMBER,
                    hijos IN OUT tAgentes ) IS
          CURSOR hijas IS SELECT identificador FROM familias WHERE familia = id_familia;
hija NUMBER;
     CURSOR cHijos IS SELECT * FROM agentes WHERE familia = id_familia;
hijo agente;
     BEGIN
          --Si la tabla no está inicializada -> la inicializamos
          IF hijos IS NULL THEN
               hijos = tAgentes();
          END IF;
          --Metemos en la tabla los hijos directos de esta familia
          OPEN cHijos;
          LOOP
               FETCH cHijos INTO hijo;
               EXIT WHEN cHijos%NOTFOUND;
               hijos.EXTEND;
               hijos(hijos.LAST) := hijo;
          END LOOP;
          CLOSE cHijos;
          --Hacemos lo mismo para todas las familias hijas de la actual
          OPEN hijas;
          LOOP
               FETCH hijas INTO hija;
               EXIT WHEN hijas%NOTFOUND;
               dame_hijos( hija, hijos );
          END LOOP;
          CLOSE hijas;
     END dame_hijos;
BEGIN
     ...
     dame_hijos( 10, hijos10 );
     ...
END;
```

Anexo VII. Ejemplo de paquete

```
CREATE OR REPLACE PACKAGE call_center AS     --inicialización
     --Definimos los tipos que utilizaremos
     SUBTYPE agente IS agentes%ROWTYPE;
     SUBTYPE familia IS familias%ROWTYPE;
     SUBTYPE oficina IS oficinas%ROWTYPE;
     TYPE tAgentes IS TABLE OF agente;
     TYPE tFamilias IS TABLE OF familia;
     TYPE tOficinas IS TABLE OF oficina;

     --Definimos las excepciones propias
     referencia_no_encontrada exception;
     referencia_encontrada exception;
     no_null exception;
     PRAGMA EXCEPTION_INIT(referencia_no_encontrada, -2291);
     PRAGMA EXCEPTION_INIT(referencia_encontrada, -2292);
     PRAGMA EXCEPTION_INIT(no_null, -1400);

     --Definimos los errores que vamos a tratar
     todo_bien               CONSTANT NUMBER := 0;
     elemento_existente           CONSTANT NUMBER:= -1;
     elemento_inexistente           CONSTANT NUMBER:= -2;
     padre_existente          CONSTANT NUMBER:= -3;
     padre_inexistente          CONSTANT NUMBER:= -4;
     no_null_violado          CONSTANT NUMBER:= -5;
     operacion_no_permitida          CONSTANT NUMBER:= -6;

     --Definimos los subprogramas públicos
     --Nos devuelve la oficina padre de un agente
     PROCEDURE oficina_padre( mi_agente agente, padre OUT oficina );

     --Nos devuelve la oficina padre de una familia
     PROCEDURE oficina_padre( mi_familia familia, padre OUT oficina );
```

```
     --Nos da los hijos de una familia
     PROCEDURE dame_hijos( mi_familia familia, hijos IN OUT tAgentes );

     --Nos da los hijos de una oficina
     PROCEDURE dame_hijos( mi_oficina oficina, hijos IN OUT tAgentes );

     --Inserta un agente
     FUNCTION inserta_agente ( mi_agente agente )
     RETURN NUMBER;

     --Inserta una familia
     FUNCTION inserta_familia( mi_familia familia )
     RETURN NUMBER;

     --Inserta una oficina
     FUNCTION inserta_oficina ( mi_oficina oficina )
     RETURN NUMBER;

     --Borramos una oficina
     FUNCTION borra_oficina( id_oficina NUMBER )
     RETURN NUMBER;

     --Borramos una familia
     FUNCTION borra_familia( id_familia NUMBER )
     RETURN NUMBER;

     --Borramos un agente
     FUNCTION borra_agente( id_agente NUMBER )
     RETURN NUMBER;
END call_center;
/
CREATE OR REPLACE PACKAGE BODY call_center AS          --cuerpo
     --Implemento las funciones definidas en la especificación

     --Nos devuelve la oficina padre de un agente
     PROCEDURE oficina_padre( mi_agente agente, padre OUT oficina ) IS
          mi_familia familia;
     BEGIN
          IF (mi_agente.oficina IS NOT NULL) THEN
               SELECT * INTO padre FROM oficinas
               WHERE identificador = mi_agente.oficina;
          ELSE
               SELECT * INTO mi_familia FROM familias
               WHERE identificador = mi_agente.familia;
               oficina_padre( mi_familia, padre );
          END IF;
     EXCEPTION
          WHEN OTHERS THEN
               padre := NULL;
     END oficina_padre;

     --Nos devuelve la oficina padre de una familia
     PROCEDURE oficina_padre( mi_familia familia, padre OUT oficina ) IS
          madre familia;
     BEGIN
          IF (mi_familia.oficina IS NOT NULL) THEN
               SELECT * INTO padre FROM oficinas
               WHERE identificador = mi_familia.oficina;
          ELSE
               SELECT * INTO madre FROM familias
               WHERE identificador = mi_familia.familia;
               oficina_padre( madre, padre );
          END IF;
     EXCEPTION
          WHEN OTHERS THEN
               padre := NULL;
     END oficina_padre;

     --Nos da los hijos de una familia
     PROCEDURE dame_hijos( mi_familia familia, hijos IN OUT tAgentes ) IS
          CURSOR cHijos IS SELECT * FROM agentes
          WHERE familia = mi_familia.identificador;
          CURSOR cHijas IS SELECT * FROM familias
          WHERE familia = mi_familia.identificador;
          hijo agente;
          hija familia;
     BEGIN
          --inicializamos la tabla si no lo está
          if (hijos IS NULL) THEN
               hijos := tAgentes();
          END IF;
          --metemos en la tabla los hijos directos
          OPEN cHijos;
          LOOP
               FETCH cHijos INTO hijo;
               EXIT WHEN cHijos%NOTFOUND;
```

```
                hijos.EXTEND;
                hijos(hijos.LAST) := hijo;
           END LOOP;
           CLOSE cHijos;
           --hacemos lo mismo para las familias hijas
           OPEN cHijas;
           LOOP
                FETCH cHijas INTO hija;
                EXIT WHEN cHijas%NOTFOUND;
                dame_hijos( hija, hijos );
           END LOOP;
           CLOSE cHijas;
      EXCEPTION
           WHEN OTHERS THEN
                hijos := tAgentes();
      END dame_hijos;

      --Nos da los hijos de una oficina
      PROCEDURE dame_hijos( mi_oficina oficina, hijos IN OUT tAgentes ) IS
           CURSOR cHijos IS SELECT * FROM agentes
           WHERE oficina = mi_oficina.identificador;
           CURSOR cHijas IS SELECT * FROM familias
           WHERE oficina = mi_oficina.identificador;
           hijo agente;
           hija familia;
      BEGIN
           --inicializamos la tabla si no lo está
           if (hijos IS NULL) THEN
                hijos := tAgentes();
           END IF;
           --metemos en la tabla los hijos directos
           OPEN cHijos;
           LOOP
                FETCH cHijos INTO hijo;
                EXIT WHEN cHijos%NOTFOUND;
                hijos.EXTEND;
                hijos(hijos.LAST) := hijo;
           END LOOP;
           CLOSE cHijos;
           --hacemos lo mismo para las familias hijas
           OPEN cHijas;
           LOOP
                FETCH cHijas INTO hija;
                EXIT WHEN cHijas%NOTFOUND;
                dame_hijos( hija, hijos );
           END LOOP;
           CLOSE cHijas;
      EXCEPTION
           WHEN OTHERS THEN
                hijos := tAgentes();
      END dame_hijos;

      --Inserta un agente
      FUNCTION inserta_agente ( mi_agente agente )
      RETURN NUMBER IS
      BEGIN
           IF (mi_agente.familia IS NULL and mi_agente.oficina IS NULL) THEN
                RETURN operacion_no_permitida;
           END IF;
           IF (mi_agente.familia IS NOT NULL and mi_agente.oficina IS NOT NULL) THEN
                RETURN operacion_no_permitida;
           END IF;
           INSERT INTO agentes VALUES (mi_agente.identificador, mi_agente.nombre, mi_agente.usuario, mi_agente.clave, mi_agente.habilid
           COMMIT;
           RETURN todo_bien;
      EXCEPTION
           WHEN referencia_no_encontrada THEN
                ROLLBACK;
                RETURN padre_inexistente;
           WHEN no_null THEN
                ROLLBACK;
                RETURN no_null_violado;
           WHEN DUP_VAL_ON_INDEX THEN
                ROLLBACK;
                RETURN elemento_existente;
           WHEN OTHERS THEN
                ROLLBACK;
                RETURN SQLCODE;
      END inserta_agente;

      --Inserta una familia
      FUNCTION inserta_familia( mi_familia familia )
      RETURN NUMBER IS
      BEGIN
           IF (mi_familia.familia IS NULL and mi_familia.oficina IS NULL) THEN
                RETURN operacion_no_permitida;
           END IF;
```

```
          IF (mi_familia.familia IS NOT NULL and mi_familia.oficina IS NOT NULL) THEN
               RETURN operacion_no_permitida;
          END IF;
          INSERT INTO familias VALUES ( mi_familia.identificador, mi_familia.nombre, mi_familia.familia, mi_familia.oficina );
          COMMIT;
          RETURN todo_bien;
     EXCEPTION
          WHEN referencia_no_encontrada THEN
               ROLLBACK;
               RETURN padre_inexistente;
          WHEN no_null THEN
               ROLLBACK;
               RETURN no_null_violado;
           WHEN DUP_VAL_ON_INDEX THEN
               ROLLBACK;
               RETURN elemento_existente;
          WHEN OTHERS THEN
               ROLLBACK;
               RETURN SQLCODE;
     END inserta_familia;

     --Inserta una oficina
     FUNCTION inserta_oficina ( mi_oficina oficina )
     RETURN NUMBER IS
     BEGIN
          INSERT INTO oficinas VALUES (mi_oficina.identificador, mi_oficina.nombre, mi_oficina.domicilio, mi_oficina.localidad, mi_ofi
          COMMIT;
          RETURN todo_bien;
     EXCEPTION
          WHEN no_null THEN
               ROLLBACK;
               RETURN no_null_violado;
          WHEN DUP_VAL_ON_INDEX THEN
               ROLLBACK;
               RETURN elemento_existente;
          WHEN OTHERS THEN
               ROLLBACK;
               RETURN SQLCODE;
     END inserta_oficina;

     --Borramos una oficina
     FUNCTION borra_oficina( id_oficina NUMBER )
     RETURN NUMBER IS
          num_ofi NUMBER;
     BEGIN
          SELECT COUNT(*) INTO num_ofi FROM oficinas
          WHERE identificador = id_oficina;
          IF (num_ofi = 0) THEN
               RETURN elemento_inexistente;
          END IF;
          DELETE oficinas WHERE identificador = id_oficina;
          COMMIT;
          RETURN todo_bien;
     EXCEPTION
          WHEN OTHERS THEN
               ROLLBACK;
               RETURN SQLCODE;
     END borra_oficina;

     --Borramos una familia
     FUNCTION borra_familia( id_familia NUMBER )
     RETURN NUMBER IS
          num_fam NUMBER;
     BEGIN
          SELECT COUNT(*) INTO num_fam FROM familias
          WHERE identificador = id_familia;
          IF (num_fam = 0) THEN
               RETURN elemento_inexistente;
          END IF;
          DELETE familias WHERE identificador = id_familia;
          COMMIT;
          RETURN todo_bien;
     EXCEPTION
          WHEN OTHERS THEN
               ROLLBACK;
               RETURN SQLCODE;
     END borra_familia;

     --Borramos un agente
     FUNCTION borra_agente( id_agente NUMBER )
     RETURN NUMBER IS
          num_ag NUMBER;
     BEGIN
          SELECT COUNT(*) INTO num_ag FROM agentes
          WHERE identificador = id_agente;
          IF (num_ag = 0) THEN
               RETURN elemento_inexistente;
```

```
          END IF;
          DELETE agentes WHERE identificador = id_agente;
          COMMIT;
          RETURN todo_bien;
     EXCEPTION
          WHEN OTHERS THEN
               ROLLBACK;
               RETURN SQLCODE;
     END borra_agente;
END call_center;
/
```

Mapa conceptual