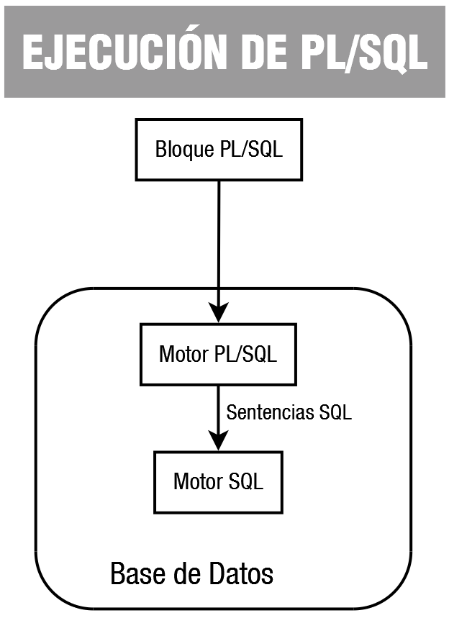
Programación de bases de datos

# Introducción

**Pero, ¿qué es realmente PL/SQL?**

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

# Conceptos básicos

## Unidades léxicas

PL/SQL es un lenguaje

* no sensible a las mayúsculas
* Cada unidad léxica puede estar separada
  + por espacios (debe estar separada por espacios si se trata de 2identificadores),
  + por saltos de línea
  + por tabuladores para aumentar la legibilidad del código escrito.

*/\*3 formas de escribir lo mismo\*/*

*/\*1\*/*IF A=CLAVE THEN ENCONTRADO:=TRUE;ELSE ENCONTRADO:=FALSE;END IF;

*/\*2\*/*if a=clave then encontrado:=true;else encontrado:=false;end if;

*/\*3\*/*

IF a = clave THEN

encontrado := TRUE;

ELSE

encontrado := FALSE;

END IF;

Las unidades léxicas se pueden clasificar en:

* **Delimitadores**: utilizados para representar operaciones entre tipos de datos, delimitar comentarios, etc.
* **Identificadores**. Nombrar elementos de nuestros programas
  + Un identificador es una letra seguida opcionalmente de letras, números, $, \_, #.
  + No podemos utilizar como identificador una palabra reservada.
    - Ejemplos válidos: X, A1, codigo\_postal.
    - Ejemplos no válidos: rock&roll, on/off.
  + PL/SQL nos permite además definir los identificadores acotados, en los que podemos usar cualquier carácter con una longitud máxima de 30 y deben estar delimitados por ". Ejemplo: "X\*Y".
  + En PL/SQL existen algunos identificadores predefinidos y que tienen un significado especial ya que nos permitirán darle sentido sintáctico a nuestros programas. Estos identificadores son las palabras reservadas y no las podemos utilizar como identificadores en nuestros programas. Ejemplo: IF, THEN, ELSE ...
  + Algunas palabras reservadas para PL/SQL no lo son para SQL, por lo que podríamos tener una tabla con una columna llamada 'type' por ejemplo, que nos daría un error de compilación al referirnos a ella en PL/SQL. La solución sería acotarlos. SELECT "TYPE" …
* **Literales**. comparaciones de valores o para asignar valores concretos a los identificadores que actúan como variables o constantes.
  + Los literales numéricos se expresarán por medio de notación decimal o de notación exponencial. Ejemplos: 234, +341, 2e3, -2E-3, 7.45, 8.1e3.
  + Los literales tipo carácter y tipo cadena se deben delimitar con unas comillas simples.
  + Los literales lógicos son TRUE y FALSE.
  + El literal NULL que expresa que una variable no tiene ningún valor asignado.
* **Comentarios**. muy conveniente utilizar comentarios en mitad del código

|  |  |  |  |
| --- | --- | --- | --- |
| **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 | || | Concatenación |
| : | Variable host | << | Delimitador de etiquetas |
| , | Separador de elementos | >> | Delimitador de etiquetas |
| \* | Producto | -- | Comentario de una línea |
| “ | Delimitador de identificador acotado | /\* | Comentarios de varias líneas |
| = | Igual relacional | \*/ | Comentarios de varias líneas |
| < | Menor | := | Asignación |
| > | Mayor | => | Selector de nombre de parámetro |
| @ | Indicador de acceso remoto | .. | Rango |
| ; | Terminador de sentencias |  |  |
| - | Resta/negación |  |  |

## Tipos de datos simples, variables y constantes

### Subtipos

PL/SQL nos permite definir subtipos de tipos de datos para darles un nombre diferente y así aumentar la legibilidad de nuestros programas. Los tipos de operaciones aplicables a estos subtipos serán las mismas que los tipos de datos de los que proceden. La sintaxis será:

SUBTYPE subtipo IS tipo\_base;

Donde subtipo será el nombre que le demos a nuestro subtipo y tipo\_base será cualquier tipo de dato en PL/SQL.

A la hora de especificar el tipo base, podemos utilizar el modificador %TYPE para indicar el tipo de dato de una variable o de una columna de la base de datos y %ROWTYPE para especificar el tipo de un cursor o tabla de una base de dato

SUBTYPE id\_familia IS familias.identificador%TYPE;

SUBTYPE agente IS agentes%ROWTYPE;

Los subtipos no podemos restringirlos, pero podemos usar un truco para conseguir el mismo efecto y es por medio de una variable auxiliar:

SUBTYPE 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;

### Variables y constantes

Para declarar variables o constantes pondremos el nombre de la variable, seguido del tipo de datos y opcionalmente una asignación. Si es una constante antepondremos la palabra CONSTANT al tipo de dato (lo que querrá decir que no podemos cambiar su valor). Podremos sustituir el operador de asignación en las declaraciones por la palabra reservada DEFAULT. También podremos forzar a que no sea nula utilizando la palabra NOT NULL después del tipo y antes de la asignación. Si restringimos una variable con NOT NULL deberemos asignarle un valor al declararla, de lo contrario PL/SQL lanzará la 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*

#### Conversión de tipos.

Aunque en PL/SQL existe la conversión implícita de tipos para tipos parecidos, siempre es aconsejable utilizar la conversión explícita de tipos por medio de funciones de conversión (TO\_CHAR, TO\_DATE, TO\_NUMBER, …) y así evitar resultados inesperados.

#### Precedencia de operadores.

Al igual que en nuestro lenguaje matemático se utiliza una precedencia entre operadores a la hora de realizar las operaciones aritméticas, en PL/SQL también se establece dicha precedencia para evitar confusiones. Si dos operadores tienen la misma precedencia lo aconsejable es utilizar los paréntesis (al igual que hacemos en nuestro lenguaje matemático) para alterar la precedencia de los mismos ya que las operaciones encerradas entre paréntesis tienen mayor precedencia. En la tabla siguiente se muestra la precedencia de los operadores de mayor a menor.

|  |  |
| --- | --- |
| 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. |

## El BLOQUE PL/SQL

Un bloque PL/SQL consta de tres zonas:

* Declaraciones: definiciones de variables, constantes, cursores y excepciones.
* Proceso: zona donde se realizará el proceso en sí, conteniendo las sentencias ejecutables.
* Excepciones: zona de manejo de errores en tiempo de ejecución.

[DECLARE

    [Declaración de variables, constantes, cursores y excepciones]]

BEGIN

    [Sentencias ejecutables]

[EXCEPTION

    Manejadores de excepciones]

END;

DECLARE

    aux **number** := 10;

BEGIN

    DECLARE

        aux **number** := 5;

    BEGIN

        ...

        IF aux = 10 THEN *--evalúa a FALSE, no entraría*

        ...

    END;

END;

## Estructuras de control

### Control adicional

Las estructuras de control condicional nos permiten llevar a cabo una acción u otra dependiendo de una condición. Vemos sus diferentes variantes:

**IF-THEN**: Forma más simple de las sentencias de control condicional. Si la evaluación de la condición es TRUE,entonces se ejecuta la secuencia de sentencias encerradas entre el THEN y el final de la sentencia.

*--sintaxis*

IF condicion THEN

secuencia\_de\_sentencias;

END IF;

*--ejemplo*

IF (b<>0) THEN

c:=a/b;

END IF;

**IF-THEN-ELSE**: Con esta forma de la sentencia ejecutaremos la primera secuencia de sentencias si la condición evalúa a TRUE y en caso contrario ejecutaremos la segunda secuencia de sentencias.

*--sintaxis*

IF condicion THEN

    Secuencia\_de\_sentencias1;

ELSE

    Secuencia\_de\_sentencias2;

END IF;

*--ejemplo*

IF (b<>0) THEN

    c:=a/b;

END IF;

**IF-THEN-ELSIF**: Con esta última forma de la sentencia condicional podemos hacer una selección múltiple. Si la evaluación de la condición 1 da TRUE, ejecutamos la secuencia de sentencias 1, sino evaluamos la condición 2. Si esta evalúa a TRUE ejecutamos la secuencia de sentencias 2 y así para todos los ELSIF que haya. El último ELSE es opcional y es por si no se cumple ninguna de las condiciones anteriores.

*--sintaxis*

IF condicion1 THEN

    Secuencia\_de\_sentencias1;

ELSIF condicion2 THEN

    Secuencia\_de\_sentencias2;

...

[ELSE

    Secuencia\_de\_sentencias;]

END IF;

*--ejemplo*

IF (operacion = ‘SUMA’) THEN

    resultado := arg1 + ar2;

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;

END IF;

### Control iterativo

Estas estructuras nos permiten ejecutar una secuencia de sentencias un determinado número de veces.

**LOOP**: La forma más simple es el bucle infinito, cuya sintaxis es:

*--sintaxis*

LOOP

    secuencia\_de\_sentencias;

END LOOP;

**EXIT**: Con esta sentencia forzamos a un bucle a terminar y pasa el control a la siguiente sentencia después del bucle. Un EXIT no fuerza la salida de un bloque PL/SQL, sólo la salida del bucle.

*--sintaxis*

LOOP

    ...

    IF encontrado = TRUE THEN

        EXIT;

    END IF;

END LOOP;

**EXIT WHEN condicion**: Fuerza a salir del bucle cuando se cumple una determinada condición.

*--sintaxis*

LOOP

    ...

    EXIT WHEN encontrado;

END LOOP;

**WHILE LOOP**: Este tipo de bucle ejecuta la secuencia de sentencias mientras la condición sea cierta.

*--sintaxis*

WHILE condicion LOOP

    Secuencia\_de\_sentencias;

END LOOP;

*--ejemplo*

WHILE (not encontrado) LOOP

    ...

END LOOP;

**FOR LOOP**: Este bucle itera mientras el contador se encuentre en el rango definido.

*--sintaxis*

FOR contador IN [REVERSE] limite\_inferior..limite\_superior LOOP

Secuencia\_de\_sentencias;

END LOOP;

*--ejemplo*

FOR i IN 1..3 loop      *--i=1, i=2, i=3*

    ...

end loop;

select count(\*) INTO num\_agentes FROM agentes;

FOR i IN 1..num\_agentes LOOP

  ...

END LOOP;

FOR i IN REVERSE 1..3 LOOP *--i=3, i=2, i=1*

    ...begin

END LOOP;

## Manejo de errores

Cualquier situación de error es llamada excepción en PL/SQL. Cuando se detecta un error, una excepción es lanzada, es decir, la ejecución normal se para y el control se transfiere a la parte de manejo de excepciones. La parte de manejo de excepciones es la parte etiquetada como EXCEPTION y constará de sentencias para el manejo de dichas excepciones, llamadas 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;

La parte OTHERS captura cualquier excepción no capturada.

Las excepciones pueden estar definidas por el usuario o definidas internamente. Las excepciones predefinidas se lanzarán automáticamente asociadas a un error de Oracle. Las excepciones definidas por el usuario deberán definirse y lanzarse explícitamente.

En PL/SQL nosotros podemos definir nuestras propias excepciones en la parte DECLARE de cualquier bloque. Estas excepciones podemos lanzarlas explícitamente por medio de la sentencia RAISE nombre\_excepción.

*--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 erronea*

END;

El alcance de una excepción sigue las mismas reglas que el de una variable, por lo que si nosotros redefinimos una excepción que ya es global para el bloque, la definición local prevalecerá y no podremos capturar esa excepción a menos que el bloque en la que estaba definida esa excepción fuese un bloque nombrado, y podremos capturarla usando la sintaxis: nombre\_bloque.nombre\_excepcion.

Las excepciones predefinidas están definidas globalmente. No necesitamos (ni debemos) redefinir las excepciones predefinidas.

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 no podemos continuar por la siguiente sentencia a la que la 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 sí podemos encerrar la sentencia dentro de un bloque, y ahí capturar las posibles excepciones, para continuar con las siguientes 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

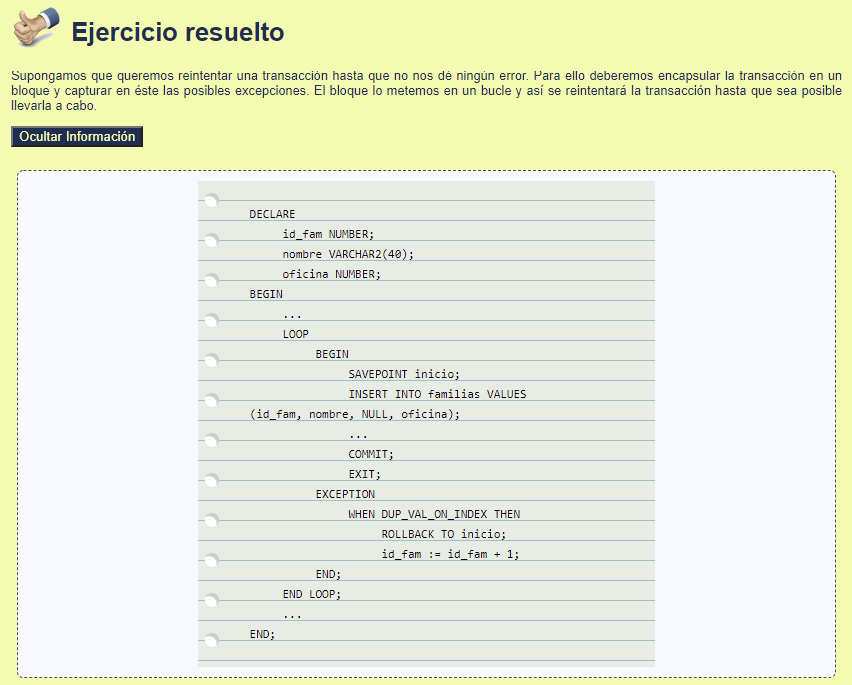
SELECT identificador INTO id\_fam FROM familias WHERE nombre = nom\_fam;

    END;

    INSERT INTO agentes VALUES (id\_ag, nom\_ag, login, password, 1, 1, id\_fam, null);

    ...

END;



Cuando ejecutamos varias sentencias seguidas del mismo tipo y queremos capturar alguna posible excepción debida al tipo de sentencia, podemos encapsular cada sentencia en un bloque y manejar en cada bloque la excepción, o podemos utilizar una variable localizadora para saber qué sentencia ha sido la que ha lanzado la excepción (aunque de esta manera no podremos continuar por la siguiente sentencia).

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 es capturada por un manejador de excepción apropiado, ésta es tratada y posteriormente el control es devuelto al bloque superior. Si la excepción no es capturada y no existe bloque superior, el control se devolverá al entorno. También puede darse que la excepción sea manejada en un bloque superior a falta de manejadores para ella en los bloques internos, la excepción se propaga de un bloque al superior y así hasta que sea manejada o no queden bloques superiores con lo que el control se devuelve al entorno. Una excepción también puede ser relanzada en un manejador. En la siguiente presentación puedes ver cómo se propagan diferentes excepciones entre diferentes bloques.

Oracle también permite que nosotros lancemos nuestros propios mensajes de error a las aplicaciones y asociarlos a un código de error que Oracle reserva, para no interferir con los demás códigos de error.

RAISE\_APPLICATION\_ERROR(error\_number, message [, (TRUE|FALSE)]);

Donde error\_number es un entero negativo comprendido entre –20000..-20999 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). Sólo podemos llamar a este procedimiento desde un subprograma.

No hay excepciones predefinidas asociadas a todos los posibles errores de Oracle, por lo que nosotros podremos asociar excepciones definidas por nosotros a errores Oracle, por medio de la directiva al compilador (o pseudoinstrucción):

PRAGMA\_INIT( nombre\_excepcion, error\_Oracle )

Donde nombre\_excepcion es el nombre de una excepción definida anteriormente, y error\_Oracle es el número negativo 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. Si una sentencia es ejecutada correctamente, SQLCODE nos devuelve 0 y en caso contrario devolverá 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;

# Tipos de datos compuestos

Un **registro** es un grupo de elementos relacionados almacenados en campos, cada uno de los cuales tiene su propio nombre y tipo de dato.

Por ejemplo, una dirección podría ser un registro con campos como calle, número, piso, puerta, código postal, ciudad, provincia y país. Los registros hacen que la información sea más fácil de organizar y representar. Para declarar un registro seguiremos la siguiente sintaxis:

TYPE nombre\_tipo IS RECORD (decl\_campo[, decl\_campo] ...);

Donde:

decl\_campo := nombre tipo [[NOT NULL] {:=|DEFAULT} expresion]

El tipo del campo será cualquier tipo de dato válido en PL/SQL excepto REF CURSOR. La expresión será cualquier expresión que evalúe al tipo de dato del campo.

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 usaremos la notación del punto.

...

mi\_direccion.calle := ‘Ramirez Arellano’;

mi\_direccion.numero := 15;

...

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

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;

    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;

## 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 la colección.

Los elementos del tipo VARRAY son los llamados arrays de longitud variable.

Son como los arrays de cualquier otro lenguaje de programación, pero con la salvedad de que a la hora de declararlos, nosotros indicamos su tamaño máximo y el array podrá ir creciendo dinámicamente hasta alcanzar ese tamaño. Un VARRAY siempre tiene un límite inferior igual a 1 y un límite superior igual al tamaño máximo.

Para declarar un VARRAY usaremos la sintaxis:

TYPE nombre IS {VARRAY | VARYING} (tamaño\_máximo) OF tipo\_elementos [NOT NULL];

Donde tamaño\_máximo será 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, éste es automáticamente nulo, por lo que para empezar a utilizarlo deberemos inicializarlo. Para ello podemos usar 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,null));

También podemos usar constructores vacíos

familias\_hijas2 familias\_hijas := familias\_hijas();

Para referenciar elementos en un VARRAY utilizaremos la sintaxis nombre\_colección(subíndice). Si una función devuelve un VARRAY, podemos usar la sintaxis: nombre\_funcion(lista\_parametros)(subindice).

IF familias\_hijas1(i).identificador = 100 THEN ...

IF dame\_familias\_hijas(10)(i).identificador = 100 THEN ...

Un VARRAY puede ser asignado 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 usaremos el método EXTEND. Sin parámetros, extendemos en 1 elemento nulo el VARRAY. EXTEND(n) añade n elementos nulos al VARRAY y EXTEND(n,i) añade n copias del i-ésimo elemento.

COUNT nos dirá 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 devolverá el antecesor y el sucesor del elemento.

Al trabajar con VARRAY podemos hacer que salte alguna de las siguientes excepciones, debidas a un mal uso de los mismos: COLECTION\_IS\_NULL, SUBSCRIPT\_BEYOND\_COUNT, SUBSCRIPT\_OUTSIDE\_LIMIT y VALUE\_ERROR.

*--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\_enteros(5) :=20;  *--Lanzaría SUBSCRIPT BEYOND COUNT*

    v\_enteros(-1) :=5;  *--Lanzaría SUBSCRIPT OUTSIDE LIMITÇ*

    v\_enteros('A') :=25;    *--Lanzaría VALUE ERROR*

END;

### Tablas anidadas

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

Para declararlos utilizaremos la siguiente sintaxis:

TYPE nombre IS TABLE OF tipo\_elementos [NOT NULL];

Donde tipo\_elementos tendrá las mismas restricciones que para los VARRAY.

Al igual que pasaba con los VARRAY, al declarar una tabla anidada, ésta es automáticamente nula, por lo que deberemos inicializarla antes de usarla.

TYPE hijos IS TABLE OF agente;

hijos\_fam hijos := hijos( agente(...) ...);

También podemos usar un constructor nulo.

Para referenciar elementos usamos la misma sintaxis que para los VARRAY.

Para extender una tabla usamos EXTEND exactamente igual que para los VARRAY. COUNT nos dirá el número de elementos, que no tiene por qué coincidir con LAST. LIMIT no tiene sentido y devuelve NULL. EXISTS(n) devuelve TRUE si el elemento existe, y FALSE en otro caso (el elemento ha sido borrado). FIRST devuelve el primer elemento que no siempre será 1, ya que hemos podido borrar elementos del principio. LAST devuelve el último elemento. PRIOR y NEXT nos dicen el antecesor y sucesor del elemento (ignorando elementos 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, por lo que 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 nos devolverá FALSE.

Al trabajar con tablas anidadas podemos hacer que salte alguna de las siguientes excepciones, debidas a un mal uso de las mismas: COLECTION\_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 en su 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;  *--Lnzaría SUBSCRIPT BEYOND COUNT*

tabla\_num(-1) := 0; *--lanzaria SUBSCRIPT OUTSIDE LIMIT*

tabla\_num(‘y’) := 5*--Lanzaría VALUE-ERROR*

END;

## Cursores

Un cursor no es más que una estructura que almacena el conjunto de filas devuelto por una consulta a la base de datos.

**Atributos de un cursor**.

Cada cursor tiene 4 atributos que podemos usar para obtener información sobre la ejecución del mismo o sobre los datos. Estos atributos pueden ser usados en PL/SQL, pero no en SQL. Aunque estos atributos se refieren en general a cursores explícitos y tienen que ver con las operaciones que hayamos realizado con el cursor, es deseable comentarlas aquí y en el siguiente apartado tomarán pleno sentido.

* %FOUND: Después de que el cursor esté abierto y antes del primer FETCH, %FOUND devuelve NULL. Después del primer FETCH, %FOUND devolverá TRUE 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 filas. En otro caso %FOUND devuelve FALSE.
* %NOTFOUND: Es lógicamente lo contrario a %FOUND.
* %ISOPEN: Evalúa a TRUE si el cursor está abierto y FALSE en caso contrario. Par cursores implícitos, como Oracle los cierra automáticamente, %ISOPEN evalúa siempre a FALSE.
* %ROWCOUNT: Para un cursor abierto y antes del primer FETCH, %ROWCOUNT evalúa a 0. Después de cada FETCH, %ROWCOUNT es incrementado y evalúa al número de filas que hemos procesado. Para cursores implícitos %ROWCOUNT evalúa al número de filas afectadas por un INSERT, UPDATE o DELETE o el número de filas devueltas por un SELECT ... INTO ...

### Cursores explícitos

Cuando una consulta devuelve múltiples filas, podemos declarar explícitamente un cursor para procesar las filas devueltas. Cuando declaramos un cursor, lo que hacemos es darle un nombre y asociarle una consulta usando la siguiente sintaxis:

CURSOR nombre\_cursor [(parametro [, parametro] ...)] [RETURN tipo\_devuelto] IS sentencia\_sele

Donde tipo\_devuelto debe representar un registro o una fila de una tabla de la base de datos, y parámetro sigue la siguiente sintaxis:

*--Sintaxis*

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 ...

*--A un parámetro de un cursor no podemos imponerle la restricción NOT NULL.*

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

*--Para abrir un cursor usamos la sintaxis:*

OPEN nombre\_cursor [(parametro [, parametro] ...)];

*--Ejemplo*

OPEN cAgentes;

OPEN c1(1);

OPEN c1;

La sentencia FETCH devuelve una fila del conjunto resultado. Después de cada FETCH, el cursor avanza a la próxima fila en el conjunto resultado.

FETCH cFamilias INTO mi\_id, mi\_nom, mi\_fam, mi\_ofi;

*-- Para procesar un cursor entero deberemos hacerlo por medio de un bucle.*

BEGIN

...

OPEN cFamilias;

LOOP

    FETCH cFamilias INTO mi\_id, mi\_nom, mi\_fam, mi\_ofi;

    EXIT WHEN cFamilias%NOTFOUND;

    ...

END LOOP;

...

END;

*-- Una vez procesado el cursor, deberemos cerrarlo*

CLOSE cFamilias;

Una vez cerrado el cursor podemos reabrirlo, pero cualquier otra operación que hagamos con el cursor cerrado lanzará la excepción INVALID\_CURSOR.

También podemos simplificar la operación de procesamiento de un cursor, por medio de los bucles para cursores, los cuales declaran implícitamente una variable índice definida como %ROWTYPE para el cursor, abren el cursor, se van trayendo los valores de cada fila del cursor, almacenándolas en la variable índice, y finalmente cierran 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;

### Cursores variables

Para declarar un cursor variable debemos seguir 2 pasos:

*Definir un tipo REF CURSOR y entonces declarar una variable de ese tipo.*

TYPE tipo\_cursor IS REF CURSOR RETURN agentes%ROWTYPE;

cAgentes tipo\_cursor;

*--asociarlo a una consulta*

OPEN nombre\_variable\_cursor FOR sentencia\_select;

OPEN cAgentes FOR SELECT \* FROM agentes WHERE oficina = 1;

Un cursor variable no puede tomar parámetros. Podemos usar los atributos de los cursores para cursores variables.

Además, podemos usar varios OPEN-FOR para abrir el mismo cursor variable para diferentes consultas. No necesitamos cerrarlo antes de reabrirlo. Cuando abrimos un cursor variable para una consulta diferente, la consulta previa se pierde.

Una vez abierto el cursor variable, su manejo es idéntico a un cursor. Usaremos FETCH para traernos las filas, usaremos sus atributos para hacer comprobaciones y lo cerraremos cuando dejemos de usarlo.

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;

# Abstracción en PL/SQL.

## Subprogramas

Los subprogramas son bloques nombrados a los cuales les podemos pasar parámetros y los podemos invocar. Además, los subprogramas pueden estar almacenados en la base de datos o estar encerrados en otros bloques. Si el programa está almacenado en la base de datos, podremos invocarlo si tenemos permisos suficientes y si está encerrado en otro bloque lo podremos invocar si tenemos visibilidad sobre el mismo.

Hay dos clases de subprogramas: las funciones y los procedimientos. Las funciones devuelven un valor y los procedimientos no.

*--Funciones*

FUNCTION nombre [(parametro [, parametro] ...)]

RETURN tipo\_dato IS

[declaraciones\_locales]

BEGIN

sentencias\_ejecutables

[EXCEPTION

manejadores\_de\_excepciones]

END [nombre];

*--Procedimientos*

PROCEDURE nombre[(parametro [,parametro]...)]

[declaraciones\_locales]

BEGIN

sentencias\_ejecutables

[EXCEPTION

manejadores\_de\_excepciones

END [nombre];

Donde

parametro := nombre\_parametro [IN|OUT|IN OUT] tipo\_dato [{:=|DEFAULT} expresion]

Algunas consideraciones que debes tener en cuenta son las siguientes:

* No podemos imponer una restricción NOT NULL a un parámetro.
* No podemos especificar una restricción del tipo:

PROCEDURE KK(a **NUMBER**(10)) IS ... *--ilegal*

* Una función siempre debe acabar con la sentencia RETURN.

Podemos definir subprogramas al final de la parte declarativa de cualquier bloque. En Oracle, cualquier identificador debe estar declarado antes de usarse, y eso mismo pasa con los subprogramas, por lo que deberemos declararlos antes de usarlos.

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 quisiéramos definir subprogramas en orden alfabético o lógico, o necesitamos definir subprogramas mutuamente recursivos (uno llama a otro, y éste a su vez llama al anterior), deberemos 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

    ...

END;

### Almacenar subprogramas en la base de datos.

Para almacenar un subprograma en la base de datos utilizaremos la misma sintaxis que para declararlo, anteponiendo CREATE [OR REPLACE] a PROCEDURE o FUNCTION, y finalizando el subprograma con una línea que simplemente contendrá el carácter '/' para indicarle a Oracle que termina ahí. Si especificamos OR REPLACE y el subprograma ya existía, éste será reemplazado. Si no lo especificamos y el subprograma ya existe, Oracle nos devolverá un error indicando que el nombre ya está siendo utilizado por otro objeto de la base de datos.

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 los subprogramas son almacenados en la base de datos, para ellos no podemos utilizar las declaraciones hacia delante, por lo que cualquier subprograma almacenado en la base de datos deberá conocer todos los subprogramas que utilice.

Para invocar un subprograma usaremos la sintaxis:

nombre\_procedimiento [(parametro [,parametro] ...)];

variable := nombre\_funcion [(parametro [, parametro] ...)];

BEGIN

...

hijos := hijos\_familia(10);

...

END;

Si el subprograma está almacenado en la base de datos y queremos invocarlo desde SQL\*Plus usaremos la sintaxis:

EXECUTE nombre\_procedimiento [(parametros)];

EXECUTE :variable\_sql := nombre\_funcion [(parametros)];

Cuando almacenamos un subprograma en la base de datos éste es compilado antes. Si hay algún error se nos informará de los mismos y deberemos corregirlos por medio de la cláusula OR REPLACE, antes de que el subprograma pueda ser utilizado.

Hay varias vistas del diccionario de datos que nos ayudan a llevar un control de los subprogramas, tanto para ver su código, como los errores de compilación. También hay algunos comandos de SQL\*Plus que nos ayudan a hacer lo mismo pero de forma algo menos engorrosa.

|  |  |  |
| --- | --- | --- |
| 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.

### Parámetros de los subprogramas.

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

### Sobrecarga de subprogramas y recursividad

PL/SQL también nos ofrece la posibilidad de sobrecargar funciones o procedimientos, es decir, llamar con el mismo nombre subprogramas que realizan el mismo cometido y que aceptan distinto número y/o tipo de parámetros. No podemos sobrecargar subprogramas que aceptan el mismo número y tipo de parámetros y sólo difieren en el modo. Tampoco podemos sobrecargar subprogramas con el mismo número de parámetros y que los tipos de los parámetros sean diferentes, pero de la misma familia, o sean subtipos basados en la misma familia.

PL/SQL también nos ofrece la posibilidad de utilizar la recursión en nuestros subprogramas. Un subprograma es recursivo si éste se invoca a él mismo.

## Paquetes.

Un paquete es un objeto que agrupa tipos, elementos y subprogramas. Suelen tener dos partes: la especificación y el cuerpo, aunque algunas veces el cuerpo no es necesario.

En la parte de especificación declararemos la interfaz del paquete con nuestra aplicación y en el cuerpo es donde implementaremos esa interfaz.

Para crear un paquete usaremos la siguiente sintaxis:

CREATE [OR REPLACE] PACKAGE nombre AS

    [declaraciones públicas y especificación de subprogramas]

END [nombre];

CREATE [OR REPLACE] PACKAGE BODY nombre AS

    [declaraciones privadas y cuerpo de los subprogramas especificados]

[BEGIN

    sentencias de inicialización]

END [nombre];

Para referenciar las partes visibles de un paquete, lo haremos por medio de la notación del punto.

BEGIN

    ...

    call\_center.borra\_agente( 10 );

    ...

END;

### Ejemplos de utilización del paquete DBMS\_OUTPUT.

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

SQL\*Plus permite visualizar los mensajes que hay en el buffer, por medio del comando SET SERVEROUTPUT ON. La utilización fundamental de este paquete es para la depuración de nuestros subprogramas.

Veamos uno a uno los subprogramas que nos suministra este paquete:

* 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 demás subprogramas y purga el buffer. Como con ENABLE no es necesario si estamos usando la opción SERVEROUTPUT.

DISABLE

DISABLE();

*--Coloca elementos en el buffer, los cuales son convertidos 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 devolverá 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 del tipo CHARARR, tipo definido el paquete DBMS\_OUTPUT como una tabla de VARCHAR2(255).

GET\_LINES

GET\_LINES(lines OUT CHARARR, numlines IN OUT **INTEGER**);

## Objetos

Los tipos de objetos tiene 2 partes: una especificación y un cuerpo. La parte de especificación declara los atributos y los métodos que harán de interfaz de nuestro tipo de objeto. En el cuerpo se implementa la parte de especificación. En la parte de especificación debemos declarar primero los atributos y después los métodos. Todos los atributos son públicos (visibles). No podemos declarar atributos en el cuerpo, pero sí podemos declarar subprogramas locales que serán visibles en el cuerpo del objeto y que nos ayudarán a implementar nuestros métodos.

Los atributos pueden ser de cualquier tipo de datos Oracle, 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. Tampoco podemos imponerle la restricción NOT NULL.

Un método es un subprograma declarado en la parte de especificación de un tipo de objeto por medio de:

MEMBER. Un método no puede llamarse igual que el tipo de objeto o que cualquier atributo. Para cada método en la parte de especificación, debe haber un método implementado en el cuerpo con la misma cabecera.

Todos los métodos en un tipo de objeto aceptan como primer parámetro una instancia de su tipo. Este parámetro es SELF y siempre está accesible a un método. Si lo declaramos explícitamente debe ser el primer parámetro, con el nombre SELF y del tipo del tipo de objeto. Si SELF no está declarado explícitamente, por defecto será IN para las funciones e IN OUT para los procedimientos.

Los métodos dentro de un tipo de objeto pueden sobrecargarse. No podemos sobrecargarlos si los parámetros

formales sólo difieren en el modo o pertenecen a la misma familia. Tampoco podremos sobrecargar una función miembro si sólo difiere en el tipo devuelto.

Una vez que tenemos creado el objeto, podemos usarlo en cualquier declaración. Un objeto cuando se declara sigue las mismas reglas de alcance y visibilidad que cualquier otra variable.

Cuando un objeto se declara éste es automáticamente NULL. Dejará de ser nulo cuando lo inicialicemos por medio de su constructor o cuando le asignemos otro. Si intentamos acceder a los atributos de un objeto NULL saltará la excepción ACCES\_INTO\_NULL.

Todos los objetos tienen constructores por defecto con el mismo nombre que el 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, deberemos hacerlo nosotros explícitamente.

DECLARE

    familia1 Familia;

BEGIN

    ...

    familia1 := Familia( 10, ‘Fam10’, 1, NULL );

    ...

end;

### Objetos. Funciones mapa y funciones de orden.

Una función miembro mapa es una función sin parámetros que devuelve un tipo de dato: DATE, NUMBER o VARCHAR2 y sería similar a una función hash. Se definen anteponiendo la palabra clave MAP y sólo puede haber una para el 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 que devuelve un número negativo si el objeto pasado es menor, cero si son iguales y un número positivo 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;

    ...begin

end;

# Disparadores

Un **disparador** no es más que un procedimiento que es ejecutado cuando se realiza alguna sentencia de manipulación de datos sobre una tabla dada y bajo unas circunstancias establecidas a la hora de definirlo.

Por lo que un disparador puede ser usado para:

* Llevar a cabo auditorías sobre la historia de los datos en nuestra base de datos.
* Garantizar complejas reglas de integridad.
* Automatizar la generación de valores derivados de columnas.
* 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.

Un disparador puede ser lanzado antes o después de realizar la operación que lo lanza. Por lo que tendremos disparadores BEFORE y disparadores AFTER.

Un disparador puede ser lanzado una vez por sentencia o una vez por cada fila a la que afecta. Por lo que tendremos disparadores de sentencia y disparadores de fila.

Un disparador puede ser lanzado al insertar, al actualizar o al borrar de una tabla, por lo que tendremos disparadores INSERT, UPDATE o DELETE (o mezclados).

## Definición de disparadores

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;

Donde nombre nos indica el nombre que le damos al disparador, momento nos dice cuando será lanzado el disparador (BEFORE o AFTER), acontecimiento será la acción que provoca el lanzamiento del disparador (INSERT y/o DELETE y/o UPDATE). REFERENCING y WHEN sólo podrán ser utilizados con disparadores para filas. REFERENCING nos permite asignar un alias a los valores NEW o/y OLD de las filas afectadas por la operación, y WHEN nos permite indicar al disparador que sólo sea lanzado cuando sea TRUE una cierta condición evaluada para cada fila afectada.

En un disparador de fila, podemos acceder a los valores antiguos y nuevos de la fila afectada por la operación, referenciados como :old y :new (de ahí que podamos utilizar la opción REFERENCING para asignar un alias). Si el disparador es lanzado al insertar, el valor antiguo no tendrá sentido y el valor nuevo será la fila que estamos insertando. Para un disparador lanzado al actualizar el valor antiguo contendrá la fila antes de actualizar y el valor nuevo contendrá la fila que vamos actualizar. Para un disparador lanzado al borrar sólo tendrá sentido el valor antiguo.

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

Un disparador de fila no puede acceder a la tabla asociada. Se dice que esa tabla está mutando. Si un disparador es lanzado en cascada por otro disparador, 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 tipos de disparadores sobre una misma tabla, el orden de ejecución será:

* 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 los 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)

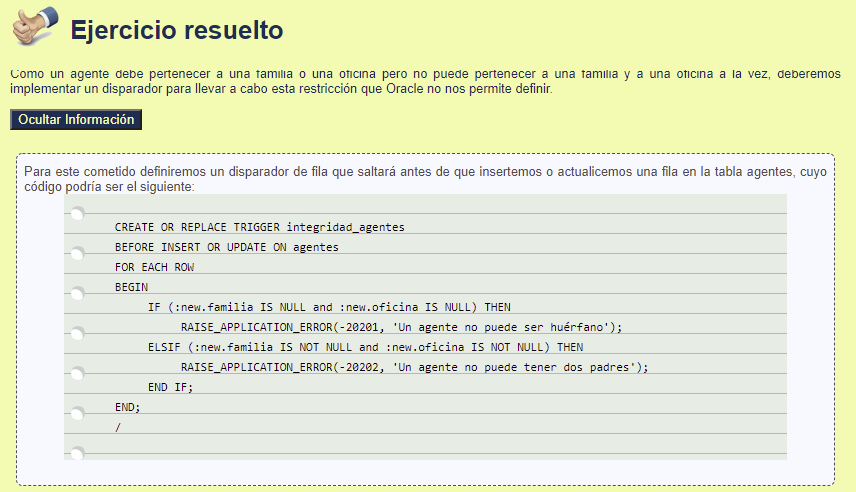
REFERENCING\_NAMES **VARCHAR2**(87)

WHEN\_CLAUSE **VARCHAR2**(4000)

STATUS **VARCHAR2**(8)

DESCRIPTION **VARCHAR2**(4000)

TRIGGER\_BODY **LONG**



# Interfaces de programación de aplicaciones para lenguajes externos.

Las primeras APIs utilizadas para acceder a bases de datos Oracle fueron las que el mismo Oracle proporcionaba: Pro\*C, Pro\*Fortran y Pro\*Cobol. Todas permitían embeber llamadas a la base de datos en nuestro programa y a la hora de compilarlo, primero debíamos pasar el precompilador adecuado que trasladaba esas llamadas embebidas en llamadas a una librería utilizada en tiempo de ejecución. Sin duda, el más utilizado fue Pro\*C, ya que el lenguaje C y C++ tuvieron un gran auge.

Hoy día existen muchas más APIs para acceder a las bases de datos ya que tanto los lenguajes de programación como las tecnologías han evolucionado mucho. Antes la programación para la web casi no existía y por eso todos los programas que accedían a bases de datos lo hacían bien en local o bien a través de una red local. Hoy día eso sería impensable, de ahí que las APIs también hayan evolucionado mucho y tengamos una gran oferta a nuestro alcance para elegir la que más se adecue a nuestra necesidades.

**Pregunta 1: Si deseamos que una variable sea del mismo tipo que un registro de una tabla usamos:**

1. No se puede hacer
2. %VARTYPE
3. %TYPE
4. %ROWTYPE

**Pregunta 2: Qué bucle es mejor si conocemos el número de iteraciones:**

1. LOOPWHEN
2. LOOPIFEXIT
3. FOR
4. WHILE

**Pregunta 3:Cómo se llaman los parámetros que escribimos cuando invocamos a una función:**

1. Parámetros actuales
2. Parámetros formales
3. IN
4. INOUT

**Pregunta 4:Podemos crear una tabla desde un programa PL/SQL:**

1. Sí
2. No
3. Solo mediante una subconsulta
4. Ninguna opción anterior es posible

**Pregunta 5:A qué corresponde esta cabecera EsCodigoValido(a int) RETURN BOOLEAN IS ... corresponde a:**

1. Un procedimiento
2. Una función
3. Un trigger
4. Una excepción

**Pregunta 6:Si la variable A es TRUE y la variable B es FALSO qué valor tendrá la expresión A AND B:**

1. FALSE.
2. NULL.
3. TRUE.
4. Ninguna opción anterior es posible.

**Pregunta 7:Señala cuánto vale la siguiente expresión 2-(-3\*\*(2\*0)+2:**

1. 7.
2. 3.
3. 9.
4. 5.

**Pregunta 8:¿Cuál es el operador para concatenar cadenas?**

1. .
2. ++.
3. c)
4. \*\*.

**Pregunta 9:¿Con qué orden lanzarías una excepción?**

1. RETURN
2. EXCEPTION
3. RAISE
4. EXCEPCION

Soluciones:

Pregunta 1- d

Pregunta 2- c

Pregunta 3- a

Pregunta 4- b

Pregunta 5- b

Pregunta 6- b

Pregunta 7- d

Pregunta 8- a

Pregunta 9- c