Programación con PL/SQL

1. Introducción a PL/SQL

Oracle es una base de datos relacional. El lenguaje para acceder a las bases de datos relacionales es SQL. Sql es un lenguaje de cuarta generación, lo que quiere decir que el lenguaje describe lo que debe hacerse, pero no la manera de llevarlo a cabo.

Los lenguajes de tercera generación, como C o COBOL son de naturaleza procedimental. Un programa escrito en un lenguaje de tercera generación implementa un algoritmo paso a paso para resolver un problema. Los lenguajes orientados a objetos como C++ o Java también llevan a cabo algoritmos para resolver un problema.

Cada lenguaje tiene sus ventajas y sus desventajas. Los lenguajes de cuarta generación son por lo general, bastante simples (comparados con los lenguajes de tercera generación) y tienen menos instrucciones. Sin embargo, en algunos casos, las estructuras procedimentales disponibles en los lenguajes de tercera generación, resultan útiles para expresar un determinado programa.

PL/SQL aporta la potencia y flexibilidad del lenguaje SQL con las estructuras procedimentales de un lenguaje de tercera generación.

PL/SQL significa Procedural Language/SQL (Lenguaje procedimental/SQL). Como su propio nombre indica, PL/SQL amplia la funcionalidad de SQL añadiendo las estructuras que pueden encontrarse en otros lenguajes procedimentales.

PL/SQL y el tráfico de red.

Muchas aplicaciones de bases de datos se construyen utilizando el modelo cliente-servidor. En este modelo, el programa en sí reside en la máquina cliente y envía solicitudes a un servidor de base de datos para obtener información. Las solicitudes se envían mediante SQL. Normalmente, esto da lugar a múltiples comunicaciones por la red: una por cada instrucción SQL.
PL/SQL permite empaquetar varias instrucciones SQL en un único bloque PL/SQL, que se envía al servidor como una sola unidad. Esto reduce el tráfico de red y aumenta la velocidad de la aplicación.

2. Fundamentos de PL/SQL.

Estructuras de Bloque

PL/SQL es un lenguaje *estructurado en bloques*, lo que quiere decir que la unidad básica de codificación son bloques lógicos, los que a su vez pueden contener otros sub-bloques dentro de ellos, con las mismas características.

Un bloque (o sub-bloque) permite agrupar en forma lógica un grupo de sentencias. Hay dos tipos diferentes de bloques: **anónimos y nominados**.

Los **bloques anónimos** se construyen, por lo general, de manera dinámica, y se ejecutan una sola vez. Este tipo de bloques se suele generar desde un programa cliente para llamar a un subprograma almacenado en la base de datos.

Los **bloques con nombre** son bloques a los que se ha asignado un nombre concreto y que se pueden dividir en las siguientes categorías:

- **Bloques etiquetados**: Son bloques anónimos con una etiqueta que le da nombre al bloque. Los bloques etiquetados se utilizan generalmente como los bloques anónimos, pero la etiqueta permite hacer referencia a variables, que de otro modo no serían visibles.
- **Subprogramas**: Consisten en procedimientos y funciones. Se pueden almacenar en la base de datos como objetos independientes, como parte de un paquete o como métodos de un tipo de objeto.
- **Disparadores**: Son bloques PL/SQL asociados a un suceso que tiene lugar en la base de datos. El suceso de disparo puede ser una instrucción del lenguaje de manipulación de datos que se ejecuta sobre una tabla de la base de datos (INSERT, UPDATE O DELETE). También puede ser una instrucción de lenguaje de definición de datos (CREATE o DROP).

Un bloque PL/SQL tiene tres partes: una sección de *declaración*, una sección de *ejecución* y otra de manejo de *excepciones*. Sólo el bloque de ejecución es obligatorio en un programa PL/SQL.
Es posible anidar sub-bloques en la sección ejecutable y de excepciones, pero no en la sección de declaraciones.

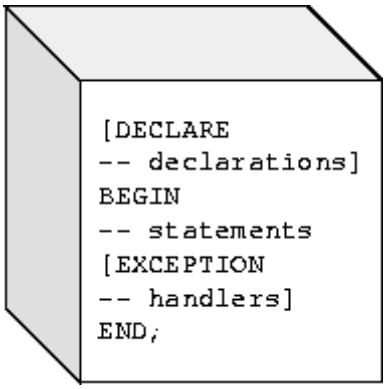


Figura 1: Estructura de bloques de un programa PL/SQL

Ejemplo de bloque PL/SQL:

```
DECLARE
  /* Comienzo de la sección declarativa */
  v_StudentID NUMBER(5) := 10000;
  v_FirstName VARCHAR2(20);
BEGIN   /* Comienzo de la sección ejecutable */
      -- Obtener el nombre del estudiante con identificador 10000
SELECT first_name
    INTO v_FirstName
    FROM students
    WHERE id = v_StudentID;

EXCEPTION  /* Comienzo de la sección de manejo de excepciones */
  WHEN NO_DATA_FOUND THEN
    -- Manejar la condición de error
    INSERT INTO log_table (info)
      VALUES ('Student 10,000 does not exist!');
END;
/
```

Ejemplo de bloques anidados:

```
DECLARE
  /* Comienzo de la sección declarativa */
  v_StudentID NUMBER(5) := 10000;
  v_FirstName VARCHAR2(20);
BEGIN
  /* Comienzo de la sección ejecutable */
-- Obtener el nombre del estudiante con identificador 10000
    SELECT first_name
    INTO v_FirstName
    FROM students
    WHERE id = v_StudentID;
-- Comienzo de un bloque anidado que contiene solo una sección ejecutable

      BEGIN  -- Es buena idea identar los bloques anidados

            INSERT INTO log_table (info) VALUES (‘Hola desde un bloque
            anidado!');
      END;
EXCEPTION
  /* Comienzo de la sección de manejo de excepciones */
  WHEN NO_DATA_FOUND THEN
-- Comienzo de un bloque anidado, que el mismo contiene una sección
ejecutable y
-- manejo de excepciones
      BEGIN
      -- Tratamiento de la condición de error
            INSERT INTO log_table (info) VALUES ('El estudiante 10,000 no e
            xiste');
      EXCEPTION
            WHEN OTHERS THEN
        -- Algo ha fallado en la inserción en la tabla log_table
                  DBMS_OUTPUT.PUT_LINE('Error al insertar en log_table!');
      END;
END;
```

Unidades léxicas

Cualquier programa PL/SQL está compuesto por unidades léxicas: los bloques componentes de un lenguaje. Esencialmente una unidad léxica es una secuencia de caracteres, donde los caracteres pertenecen al conjunto de caracteres permitido en el lenguaje PL/SQL. Este conjunto incluye:

- Letras mayúsculas y minúsculas de la A a la Z
- Números del 0 al 9
- Los símbolos ( ) + - * / < > = ! ~ ^ ; . ' @ % , " # $ & _ | { } ? [ ]
- Tabuladores, espacios y saltos de carro

Identificadores

Se emplean para dar nombre a los objetos PL/SQL, tales como variables, cursores, tipos y subprogramas. Los identificadores constan de una letra, seguida por una secuencia opcional de caracteres, que puede incluir letras, números, signos de dólar ($), caracteres de subrayado y símbolo de almohadilla (#). Los demás caracteres no pueden emplearse. La longitud máxima de un identificador es de 30 caracteres. **PL/SQL no diferencia entre mayúsculas y minúsculas.**

Palabras reservadas

Muchos identificadores, conocidos como **palabras reservadas**, tienen un significado especial para PL/SQL. No podemos utilizar estas palabras para nombrar a nuestros identificadores. Por ejemplo, BEGIN y END se emplear para delimitar los bloques PL/SQL.

Delimitadores

Son símbolos que tienen un significado especial para PL/SQL, y que se utilizan para separar unos identificadores de otros. La siguiente tabla proporciona un listado de los delimitadores disponibles en PL/SQL junto con su significado:

Símbolo	Significado
+	operador de suma
%	indicador de atributo
'	delimitador de caracteres
.	selector de componente
/	operador de división
(	expresión o delimitador de lista
)	expresión o delimitador de lista
:	indicador de variable host
,	separador de elementos

*	operador de multiplicación
“	delimitador de un identificador entre comillas
=	operador relacional
<	operador relacional
>	operador relacional
@	indicador de acceso remoto
;	terminador de sentencias
-	negación u operador de substracción

Los delimitadores compuestos consisten de dos caracteres, como por ejemplo:

Símbolo	Significado
:=	operador de asignación
=>	operador de asociación
||	operador de concatenación
**	operador de exponenciación
<<	comienzo de un rótulo
>>	fin de un rótulo
/*	comienzo de un comentario de varias líneas
*/	fin de un comentario de varias líneas
..	operador de rango
<>	operador relacional
!=	operador relacional
^=	operador relacional
<=	operador relacional
>=	operador relacional
--	comentario en una línea

Literales

Un literal es un valor numérico, booleano o de carácter que no es un identificador. Dos ejemplos de literales serían –23.456 y NULL. Los tipos numéricos, booleanos y de caracteres los explicaremos más adelante.

Literales de carácter

Los literales de carácter, también conocidos como literales de cadena, constan de uno o más caracteres delimitados por comillas simples (se trata del mismo estándar que se utiliza en SQL). Se pueden asignar a variables de tipo CHAR o VARCHAR2, sin que sea necesario efectuar ningún tipo de conversión.
Ejemplos:
'12345'
''''
Todos los literales de cadena se consideran de tipo CHAR. En un literal se puede incluir cualquier carácter imprimible del conjunto de caracteres de PL/SQL incluyendo el propio carácter de comilla simple. Para poder incluir una comilla simple dentro de una cadena de caracteres es necesario poner dos comillas simples una a continuación de otra. Por ejemplo, para incluir la cadena "Peter's friends " en un literal, habría que escribir:

'Peter''s friends'

Literales numéricos

Representan un valor entero o real, y puede ser asignado a una variable de tipo NUMBER sin necesidad de efectuar conversión alguna.
Los literales enteros consisten en una serie de dígitos, precedidos opcionalmente por un signo(+ o -).

Ejemplos de literales enteros: 123 -7 +12

Un literal real consta de un signo, opcional, y una serie de dígitos que contiene un punto decimal, como en los siguientes ejemplos:

-17.1 23.0 3.

Estos literales se pueden también escribir utilizando notación científica, como muestran estos ejemplos:
1.34E4
9.8e-3

Después de la E o la e, sólo puede escribirse un literal entero. La E significa exponente, y puede interpretarse como <<multiplicado por 10 elevado a la potencia de>>.
Los ejemplos anteriores se podrían leer entonces así:

- 1.34 por 10 elevado a la potencia de 4=1.34x1000=13400
- 9.8 por 10 a la potencia de –3 =9.8*0.001=0.0098

Literales booleanos

Sólo hay tres posibles literales booleanos: **TRUE** (verdadero), **FALSE** (falso) y **NULL** (nulo).
El valor NULL marca la diferencia con otros lenguajes que solo admiten TRUE o FALSE.

Comentarios

Tienen por objeto facilitar la lectura. Existen dos tipos:

- **Comentarios monolínea:**

Un comentario monolínea comienza con dos guiones y continúa hasta el final de la línea.

- **Comentarios multilínea**

Comienzan por delimitador /* y terminan por el delimitador */.

Este es el estilo de comentario empleado en el lenguaje C.

Ejemplo:

```
DECLARE
  v_Department  CHAR(3);  -- Variable para almacenar los 3 caracteres
del código de                          -- departamento
  v_Course      NUMBER;   -- Variable para almacenar el código de
curso.
BEGIN
/* Inserta el curso identficado por v_Departament y v_Curso
   en la tabla classes de la base de datos */
INSERT INTO classes (department, course)
       VALUES (v_Department, v_Course);
END;
```

Declaraciones de variables

La comunicación con la base de datos tiene lugar mediante el uso de variables incluidas en los bloques PL/SQL. Las variables son espacios de memoria que pueden contener valores de datos.
A medida que se ejecuta el programa, el contenido de las variables puede cambiar. Se puede asignar información de la base de datos a una variable, y también puede insertarse el contenido de una variable en la base de datos. Las variables se pueden modificar directamente con instrucciones PL/SQL.
Se declaran en la sección declarativa de un bloque, y cada una de ellas tiene un tipo específico, que describe el tipo de información que puede almacenarse en ella.

Sintaxis de las declaraciones

La declaración tiene lugar en la sección declarativa de un bloque. La sintaxis general para la declaración de variables es la siguiente:

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

Ejemplos:

```
DECLARE
      V_descripcion varchar2(50);
      V_numerodeusuarios NUMBER:=30;
```

Se puede utilizar como nombre de variable cualquier identificador legal PL/SQL.

Si a una variable no se le da valor inicial se le asigna como valor predeterminado NULL.
Si se incluye la cláusula NOT NULL, en la declaración, entonces es obligatorio inicializar la variable al declararla, y además ésta no podrá tomar valor NULL.
Si se utiliza la opción CONSTANT en la declaración, es preciso dar un valor inicial a la variable, y su valor no puede ser luego cambiado. Estamos definiendo por tanto una constante.

Tipos PL/SQL

Tipos escalares

Los tipos escalares válidos son los mismos tipos que pueden usarse en una columna de base de datos, más algunos tipos adicionales.
Existen siete familias de tipos escalares: numéricos, de carácter, raw, de fecha/intervalo, rowid, booleanos y trusted.

Numéricos

Almacenan valores enteros o reales. Existen tres tipos básicos: NUMBER, PLS_INTEGER Y BINARY_INTEGER. Las variables de tipo NUMBER pueden contener valores enteros o reales, mientras que las de los tipos BINARY_INTEGER o PLS_INTEGER solo pueden contener valores enteros.

NUMBER

Pueden contener un valor numérico o de punto flotante. Es igual al tipo NUMBER de la base de datos. El rango de valores es 1E-130..10E125.

Para declarar números en coma fija, se debe especificar la precsión y la escala.

El formato es:

NUMBER(precisión,escala)

Para declarar un número en coma flotante, no se debe definir la precisión ni la escala. El formato sería simplemente.

NUMBER

Para declarar números enteros sin parte decimal, usaremos este formato:

NUMBER(precision)

La precisión es el número de dígitos en la parte entera, y la escala es el número de dígitos a la derecha del punto decimal. La escala puede ser negativa, lo que indica que el valor se redondea al número especificado de lugares a la izquierda del punto decimal.

Un subtipo es un nombre alternativo para un tipo, que puede, adicionalmente restringir el conjunto de valores legales para una variable de dicho subtipo. Pueden utilizarse estos nombres alternativos para facilitar la lectura de un programa, o por cuestiones de compatibilidad con los tipos de datos de otras bases de datos.

Los tipos equivalentes:

- DEC
- DECIMAL
- DOUBLE PRECISION
- FLOAT
- NUMERIC
- REAL

Ejemplos de uso:

Declaración	Valor asignado	Valor almacenado
NUMBER	1234.5678	1234.5678
NUMBER(3)	123	123
NUMBER(3)	1234	Error: excede la precisión
NUMBER(4,3)	123.4567	Error: excede la precisión
NUMBER(4,3)	1.234567	1.235 (redondea)
NUMBER(3,-3)	1234	1000
NUMBER(3,-1)	1234	1230

BINARY_INTEGER

El tipo number se almacena en formato decimal, optimizado en cuanto a precisión y eficiencia de almacenamiento. Debido a esto, no es posible realizar operaciones aritméticas de forma directa con el tipo NUMBER.
El motor PL/SQL realiza la conversión a un tipo binario de manera automática en cuanto encuentra valores de tipo NUMBER en una expresión aritmética, y también convierte el resultado a tipo NUMBER cuando sea necesario.
Sin embargo, para valores que sólo se usen durante los cálculos, y no vayan a ser almacenados en la base de datos, se puede utilizar el tipo BINARY_INTEGER. Este tipo permite almacenar valores enteros con signo en el rango –2147483647 a +2147483647.
Los valores se almacenan en formato binario con complemento a 2.

PLS_INTEGER

Tiene el mismo rango de valores que el BINARY_INTEGER. Sin embargo, cuando se produce un desbordamiento en un cálculo en el que esté involucrado un valor PLS_INTEGER, se genera un error, cosa que no sucede en el caso de los BINARY_INTEGER.

Tipos de carácter

Permiten almacenar cadenas o datos de tipo carácter. Esta familia está compuesta por VARCHAR2, CHAR, LONG, junto con NCHAR y NVARCHAR2.

VARCHAR2

Este tipo se comporta de forma similar al tipo VARCHAR2 de la base de datos. Pueden contener cadenas de caracteres de longitud variable, con una longitud máxima especificada. La sintaxis para la declaración es: VARCHAR2(L), donde L es la longitud máxima de la variable. El límite máximo para una variable VARCHAR2 es 32767 bytes. Una columna de base de datos

tiene un limite máximo de 4000 bytes. Si una variable PL/SQL tiene un tamaño superior a 4000 bytes, solo puede insertarse en una columna de base de datos de tipo LONG.

CHAR

Las variables de este tipo son cadenas de caracteres de longitud fija. La sintaxis para la declaración de una variable CHAR es: CHAR(L), donde L es la longitud máxima en bytes. Puesto que las variables de tipo CHAR son de longitud fija, sus valores se rellenan con los caracteres en blanco necesarios para completar la longitud máxima.
La longitud máxima de una columna de base de datos de tipo CHAR es de 2000 bytes, dato a tener en cuenta si queremos almacenar una variable PL/SQL de tipo CHAR en una columna de base de datos.

LONG

A diferencia del tipo LONG de la base de datos, que puede contener hasta dos gigabytes de datos, el tipo LONG de PL/SQL es una cadena de longitud variable con una longitud máxima de 32760 bytes. Son por tanto similares a las variables VARCHAR2.

NCHAR y NVARCHAR2

Los tipos de carácter NLS NCHAR y NVARCHAR2 se emplean para almacenar cadenas de caracteres en un conjunto de caracteres distinto del propio lenguaje PL/SQL.

Tipos Raw

Los tipos de la familia raw se emplean para almacenar datos binarios.

RAW

Las variables RAW son similares a las variables CHAR, excepto en que no se realizan conversiones entre conjuntos de caracteres. La sintaxis es : RAW(L) donde L es la longitud en bytes de la variable. La longitud máxima de una variable de tipo raw es de 32767 bytes. La longitud máxima de una columna de base de datos de tipo raw es de 2000.

LONG RAW

Son similares a los datos LONG, excepto por el hecho de que PL/SQL no realizará conversiones entre conjuntos de caracteres. La longitud máxima de una variable de tipo long raw es de 32767 bytes. La longitud máxima de una columna de base de datos de tipo long raw es de 2 gigabytes.

Aclaración sobre los datos LONG y LONG RAW

LONG y LONG RAW se mantienen por compatibilidad en versiones antiguas, aunque al igual que dijimos cuando estudiamos SQL, Oracle desaconseja su uso en favor de los tipos de datos LOB.
En lugar de LONG usaremos VARCHAR2(32760), si no es suficiente usaremos un BLOB, CLOB O NLOB.
En lugar de datos de tipo LONG RAW usaremos BLOB.
Tanto BLOB, CLOB O NLOB, permiten almacenar hasta 128 terabytes.

DATE

El tipo de dato DATE de PL/SQL se comporta de la misma manera que el tipo equivalente de la base de datos. El tipo DATE, se emplea para almacenar información sobre la fecha y la hora, incluyendo el siglo, año, mes, día, hora, minuto y segundo. Internamente, una variable de tipo DATE tiene 7 bytes, con un byte para cada componente, desde el siglo al segundo. Los valores de las variables DATE se asignan bien desde una variable de caracteres o bien, desde otra variable DATE. Las variables de caracteres se convierten implícitamente utilizando el formato de fecha predeterminado de la sesión actual. Si se desea tener el máximo control, es preferible utilizar la función predefinida TO_DATE para asignar valores y la función TO_CHAR para extraerlos.

Tipos Booleanos.

El único tipo de la familia de tipos booleanos es BOOLEAN. Una variable de tipo BOOLEAN puede contener los valores TRUE, FALSE o NULL, únicamente.

Tipos compuestos.

Los tipos compuestos disponibles en PL/SQL son: registros, tablas y varrays. Un tipo compuesto es aquel que consta de una serie de componentes. Una variable de tipo compuesto contendrá una o más variables escalares (conocidas también como atributos). Los tipos compuestos los veremos en profundidad más adelante en este capítulo.

Utilización de %TYPE.

En muchos casos, las variables PL/SQL se emplean para manipular datos almacenados en una tabla de la base de datos. En este caso, la variable debe tener el mismo tipo que la columna de la tabla. Por ejemplo, supongamos que tenemos una tabla ALUMNO en nuestra base de datos y que esta tiene una columna llamada NOMBR de tipo VARCHAR2(20).
Teniendo en cuenta esto, se puede declarar una variable de la siguiente manera:

```
DECLARE
  v_nombre alumno.nombre%type;
```

Al utilizar %type, v_nombre tendrá el tipo de la columna nombre de la tabla alumno.

Conversiones entre tipos de datos

PL/SQL puede gestionar las conversiones entre tipos de datos escalares de distintas familias. Dentro de una familia, se puede convertir un tipo de datos en otro, respetando las restricciones de las variables.
Por ejemplo, una variable CHAR(10) no puede convertirse a una VARCHAR2(5), porque no hay suficiente espacio en ésta. De igual forma, las restricciones de precisión y escala pueden prohibir la conversión entre NUMBER(3,2) y NUMBER(3).
Independientemente del tipo de datos, existen dos tipos de conversión: implícita y explícita.

Conversión explícita de datos.

Las funciones de conversión predefinidas de SQL también están disponibles en PL/SQL.
Las más usadas son las siguientes:

- **TO_CHAR:** Convierte su argumento al tipo VARCHAR2, dependiendo del tipo de formato opcional.
- **TO_DATE:** Convierte su argumento al tipo date, dependiendo del especificador de formato especial.
- **TO_NUMBER:** Convierte su argumento al tipo NUMBER, dependiendo del especificador de formato opcional.

Aún cuando PL/SQL pueda realizar conversiones implícitas entre tipos de datos, es una buena práctica de programación el utilizar funciones explícitas de conversión.

Ámbito y visibilidad de las variables

El ámbito de una variable es aquella parte del programa en la que se puede acceder a dicha variable. Para una variable PL/SQL, será desde la definición de la variable hasta el final de bloque.
Cuando la variable cae fuera de ámbito, el motor de PL/SQL libera la memoria utilizada para almacenar dicha variable.
El siguiente bloque de código muestra esta situación:

```
DECLARE
  V_Numero number(3,2);
BEGIN
      DECLARE
         v_caracter varchar2(10);
      BEGIN
      ..................
      END;
.....
END;
```

El ámbito de la variable v_caracter se limita al bloque interno; después de la línea END del bloque interno, la variable cae fuera de ámbito. El ámbito de v_numero abarca hasta la línea END del bloque exterior. Ambas variables están dentro del ámbito en el bloque interno.

La visibilidad de una variable es aquella parte del programa donde se puede acceder a la variable sin necesidad de cualificar la referencia. La visibilidad sólo es posible dentro del ámbito; si una variable cae fuera de ámbito no puede ser visible.

Analicemos el bloque que se muestra a continuación:

```
DECLARE
  V_bandera BOOLEAN;
```

```
  V_SSN number(9);
BEGIN
      →1

      DECLARE
            V_SSN char(11);
            V_fechacomienzo DATE;
      BEGIN
            → 2
      END;
      → 3
END;
```

En el punto 1 tanto v_bandera como v_ssn están dentro de ámbito y son visibles.
En el punto 2, las dos variables siguen estando dentro de ámbito, pero solo es visible v_bandera. La redeclaración de v_ssn como una variable char(11) ha hecho que deje de ser visible la declaración de tipo number(9). Las cuatro variables están dentro de ámbito en el punto 2, pero solo 3 de ellas son visibles: v_bandera, v_fechacomienzo y v_ssn de tipo char(11).
En el punto 3 v_fechacomienzo y la variable v_ssn de tipo char(11) dejan de estar dentro de ámbito, y por tanto, de ser visibles. Serán visibles, y caerán dentro de ámbito, las mismas dos variables que en el punto 1.
Si una variable cae dentro de ámbito pero no es visible se puede hacer visible etiquetando el bloque adecuadamente como el siguiente bloque de código:

```
<< externo >>
DECLARE
  V_bandera BOOLEAN;
  V_SSN number(9);
BEGIN
      → 1

      DECLARE
            V_SSN char(11);
            V_fechacomienzo DATE;
      BEGIN
      → 2  Aquí v_ssn Number(9) sería visible mediante: externo.v_ssn
      END;
      → 3
END;
```

Expresiones y operadores

Los operadores definen como se asignan valores a las variables, y cómo se manipulan dichos valores. Una expresión es una secuencia de variables y literales, separados por operadores. El valor de una expresión se determina a partir de los valores de las variables y literales que la componen, y de la definición de los operadores.

Asignación

El operador más básico es el de asignación. La sintaxis es:

Variable:= expresión;

Donde variable y expresión son una variable y una expresión PL/SQL. El resultado de la evaluación de la expresión es almacenado en la posición de memoria asignado a variable. En una instrucción PL/SQL sólo puede haber una asignación.
Sería **ilegal** por ejemplo esta construcción:

```
DECLARE
      V1 NUMBER;
      V2 NUMBER;
      V3 NUMBER;
BEGIN
      V1:=v2:=v3:=0;
END;
```

Expresiones

Las expresiones PL/SQL son valores. Como tales, una expresión no es válida como instrucción independiente, sino que tiene que ser parte de otra instrucción. Por ejemplo, una expresión puede aparecer en el lado derecho de un operador de asignación o puede aparecer como parte de una instrucción SQL. Los operadores que componen una expresión determinan, junto con el tipo de los operandos, cuál es el tipo de la expresión.
Un **operando** es el argumento de un operador. Los operadores PL/SQL admiten uno o dos argumentos. La tabla **Operadores de PL/SQL** muestra la clasificación de los operadores PL/SQL por precedencia o prioridad, listándose en primer lugar los que tienen una precedencia mayor. La precedencia de los operadores determina, en una expresión, el orden de evaluación. Consideremos la siguiente expresión numérica:

3+5*7

Dado que la multiplicación tiene mayor precedencia que la suma, el resultado de esta expresión es 38 (3+35), en lugar de 56 (8*7). Para cambiar el orden de precedencia predefinido se utilizan los paréntesis.
La expresión anterior usando ahora los paréntesis daría como resultado 56:

(3+5)*8

Tabla: Operadores de PL/SQL

Operador	Tipo	Descripción
**, NOT	binario	Exponenciación, negación lógica
+,-	Unario	Identidad, negación
*,/	Binario	Multiplicación, división
+,-,||	Binario	Suma, resta, concatenación de cadenas
=,!=,<,>,<=,>=,IS NULL, LIKE, BETWEEN, IN	Binario excepto is null que es unario	Comparación lógica
AND	Binario	Conjunción lógica

OR	Binario	Disyunción lógica.

Expresiones de carácter

El único operador de caracteres es la concatenación (||). Este operador junta dos o más cadenas (o argumentos que puedan ser convertidos de modo implícito a una cadena).
Por ejemplo, la expresión:

```
'HOLA'|| 'MUNDO'
```

da como resultado 'Hola Mundo'.

Si todos los operandos de una expresión de concatenación son de tipo CHAR, entonces la expresión también lo es. Si alguno de los operandos es de tipo VARCHAR2, entonces la expresión es de tipo VARCHAR2. Los literales de cadena se consideran de tipo CHAR, de forma que el ejemplo anterior da un resultado de tipo CHAR.

Expresiones booleanas.

Todas las estructuras de control PL/SQL incluyen expresiones booleanas, también denominadas condiciones. Una expresión booleana es una expresión que tiene como resultado un valor booleano (TRUE, FALSE, o NULL). Por ejemplo:

```
x>y
NULL
(4>5) OR (-1!=z)
```

Hay tres operadores (AND, OR, NOT) que admiten argumentos booleanos y devuelven valores booleanos.
Su comportamiento se describe según sus tablas de verdad que se muestran a continuación.

Estos operadores implementan la lógica trivaluada estándar.

NOT	TRUE	FALSE	NULL
	FALSE	TRUE	NULL

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

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

Los operadores de comparación son los siguientes:

Operador	Significado
=	Igual a
!=, <>	Distinto de
<	Menor que
>	Mayor que
<=	Menor o igual
>=	Mayor o igual que

El operador **IS NULL** devuelve TRUE sólo si su operando es NULL. Los valores NULL no pueden comprobarse mediante operadores relacionales, porque cualquier expresión relacional con un operando nulo devuelve el valor NULL.

El operador LIKE se usa para comparación con patrones en cadenas de caracteres. Los caracteres comodines para la construcción de patrones son los mismos que se usaron en SQL.
(_) se corresponde con exactamente un carácter.
% Se corresponde con cero o más caracteres.

El operador **BETWEEN** combina <= y >= en una única expresión.

Ejemplo:

X>=10 and x<=15 puede reescribirse x between 10 and 15.

El operador IN devuelve TRUE si su primer operando está contenido en el conjunto identificado por su segundo operando.
Ejemplo:

X in (5,8,9,11) devolverá true si x es igual a uno de los valores especificados entre paréntesis.

Estructuras de control PL/SQL

IF-THEN-ELSE

La sintaxis para una instrucción IF-THEN-ELSE es:

```
  IF condicion1 THEN
     Secuencia de instrucciones;
     [ELSIF condicion2 THEN
          secuencia de instrucciones;]
     ...
[ELSE
     Secuencia de instrucciones;]
END IF;
```

Donde condición es cualquier expresión que de cómo resultado un valor booleano. Las cláusulas ELSIF y ELSE son opcionales, y puede ha ver tantas cláusulas ELSIF como se desee.

Una secuencia de instrucciones dentro de una instrucción IF-THEN-ELSE se ejecuta sólo si su condición asociada es verdadera (toma el valor TRUE). Si la condición toma el valor FALSE o NULL, la secuencia de instrucciones no se ejecuta.

CASE.

Introducido a partir de la versión 9i.
El formato de la sentencia CASE es el siguiente:

```
CASE variable_de_testeo
     WHEN valor 1 THEN secuencia_instrucciones1;
     WHEN valor 2 THEN secuencia_instrucciones2;
...
     WHEN valor n THEN secuencia_instruccionesn;
     [ELSE  secuencia de instruccionesm;]
END CASE;
```

Donde variable_de_testeo es la variable o expresión que se ha de comprobar, los valores valor1, valor2, ..,.valorn son los valores de comprobación y, secuencia_instrucciones_1,... secuencia_de_instruccionesn son los fragmentos de código correspndientes que se han de ejecutar. Si variable_de_testeo toma el valor 2, por ejemplo, se ejecutarán la secuencia de instrucciones 2.

La cláusula opcional ELSE se utiliza para tener la posibilidad de ejecutar un bloque de instrucciones en el caso de que la variable de testeo no tome ningún valor de los asociados a las cláusulas WHEN.

El siguiente segmento de código muestra como se puede simplificar un conjunto de sentencias if then else mediante la instrucción CASE.

```
DECLARE
  v_Major students.major%TYPE;
  v_CourseName VARCHAR2(10);
BEGIN
  -- Recupera la asignatura de un estudiante dado.
  SELECT major
    INTO v_Major
    FROM students
    WHERE ID = 10011;
  -- En función de la asignatura, selecciona un curso
  IF v_Major = 'Computer Science' THEN
    v_CourseName := 'CS  101';
  ELSIF v_Major = 'Economics' THEN
    v_CourseName := 'ECN 203';
  ELSIF v_Major = 'History' THEN
    v_CourseName := 'HIS 101';
  ELSIF v_Major = 'Music' THEN
    v_CourseName := 'MUS 100';
  ELSIF v_Major = 'Nutrition' THEN
    v_CourseName := 'NUT 307';
  ELSE
    v_CourseName := 'Unknown';
  END IF;
  DBMS_OUTPUT.PUT_LINE(v_CourseName);
END;
```

El mismo bloque reescrito con una sentence CASE

```
DECLARE
  v_Major students.major%TYPE;
  v_CourseName VARCHAR2(10);
BEGIN
-- Recupera la asignatura de un estudiante dado.
  SELECT major
    INTO v_Major
    FROM students
    WHERE ID = 10011;
-- En función de la asignatura, selecciona un curso
  CASE v_Major
    WHEN 'Computer Science' THEN
      v_CourseName := 'CS  101';
    WHEN 'Economics' THEN
      v_CourseName :='ECN 203';
    WHEN 'History' THEN
      v_CourseName := 'HIS 101';
    WHEN 'Music' THEN
      v_CourseName := 'MUS 100';
    WHEN 'Nutrition' THEN
      v_CourseName := 'NUT 307';
    ELSE
      v_CourseName := 'Unknown';
  END CASE;
  DBMS_OUTPUT.PUT_LINE(v_CourseName);
END;
```

```
DECLARE
  v_TestVar NUMBER := 1;
BEGIN
-- Puesto que el resultado de ninguna de las comprobaciones de las
cláusulas WHEN
-- toma el valor 1, se producirá el error
CASE v_TestVar
    WHEN 2 THEN DBMS_OUTPUT.PUT_LINE('Two!');
    WHEN 3 THEN DBMS_OUTPUT.PUT_LINE('Three!');
    WHEN 4 THEN DBMS_OUTPUT.PUT_LINE('Four!');
  END CASE;
END;
```

BUCLES

PL/SQL permite ejecutar instrucciones de forma repetida, utilizando bucles.

Bucles simples

Su sintaxis es:

```
LOOP
   Secuencia de instrucciones;
END LOOP;
```

La secuencia de instrucciones se ejecutará indefinidadmente, puesto que ese bucle no tiene ninguna condición de parada.
Se puede, sin embargo, añadir una condición mediante la instrucción EXIT, cuya sintaxis:

```
EXIT [WHEN condición];
```

El siguiente bucle, por ejemplo, inserta 50 filas en la tabla temp_table

```
DECLARE
  v_Counter BINARY_INTEGER := 1;
BEGIN
  LOOP
      -- Inserta una fila en la tabla temp_table con el valor del
      -- contador de bucle
      INSERT INTO temp_table VALUES (v_Counter, 'Loop index');
      v_Counter := v_Counter + 1;
      -- Condición de salida – cuando el contador de bucle sea
      -- superior a 50 saldremos
      IF v_Counter > 50 THEN
            EXIT;
      END IF;
  END LOOP;
END;
```

Reescribimos el código anterior incluyendo la instrucción EXIT WHEN:

```
DECLARE
  v_Counter BINARY_INTEGER := 1;
BEGIN
  LOOP
      -- Inserta una fila en la tabla temp_table con el valor del
      -- contador de bucle
      INSERT INTO temp_table VALUES (v_Counter, 'Loop index');
      v_Counter := v_Counter + 1;
      -- Condición de salida – cuando el contador de bucle sea
      -- superior a 50 saldremos
      EXIT WHEN v_Counter > 50;
   END LOOP;
END;
```

Bucles WHILE

La sintaxis de un bucle WHILE es

```
WHILE condición LOOP
   Secuencia_de_instrucciones
END LOOP;
```

La condición se evalúa antes de cada iteración del bucle. Si es verdadera, se ejecuta la secuencia de instrucciones. Si la condición es falsa o nula, el bucle termina y el control se transfiere a lo que esté a continuación de la instrucción END LOOP.
Podemos reescribir el ejemplo anterior mediante un bucle WHILE, de la forma siguiente:

```
DECLARE
  v_Counter BINARY_INTEGER := 1;
BEGIN
  -- Testea el contador de bucle antes de cada iteración
WHILE v_Counter <= 50 LOOP
    INSERT INTO temp_table
      VALUES (v_Counter, 'Loop index');
    v_Counter := v_Counter + 1;
  END LOOP;
END;
```

Bucles FOR numéricos

El número de iteraciones de los bucles simples y de los bucles WHILE no se conoce de antemano, sino que depende de la condición de bucle. Los bucles FOR numéricos, por el contrario, tienen un número de iteraciones definido. La sintaxis:

```
FOR contador_bucle IN [REVERSE] limite_inferior..límite_superior LOOP
  Secuencia_de_instrucciones;
END LOOP;
```

Donde contador_bucle es la variable de índice declarada de modo implícito, limite_inferior y límite_superior especifican el número de iteraciones y secuencia_de_instrucciones es el contenido del bucle.
Los límites del bucle sólo se evalúan una vez. Dichos valores determinan el número total de iteraciones, en las que contador_bucle varía entre los valores limite_inferior y limite_superior incrementándose en una unidad cada vez, hasta que el bucle se completa. Podríamos reescribir nuestro de bucle utilizando un bucle FOR, de la forma siguiente:

```
BEGIN
  FOR v_Counter IN 1..50 LOOP
    INSERT INTO temp_table
      VALUES (v_Counter, 'Loop Index');
  END LOOP;
END;
```

Utilización de REVERSE

Si se incluye la palabra REVERSE en el bucle FOR, el índice del bucle realizará las iteraciones desde el límite superior al límite inferior. Observe que la sintaxis es la misma, referenciándose el límite inferior en primer lugar.

Registros PL/SQL

Los tipos escalares (NUMBER, VARCHAR2, DATE, etc.) están ya predefinidos en el paquete STANDARD. Por tanto, para utilizar uno de estos tipos en un programa basta con declarar una variable del tipo deseado. Los tipos compuestos, por el contrario son tipos definidos por el usuario. Para poder utilizar un tipo compuesto, debe definirse primero el tipo, y luego declarar variables del tipo así definido. Veremos este proceso en las siguientes secciones.
Los registros de PL/SQL son similares a las estructuras del lenguaje C. Un registro proporciona un mecanismo para tratar con variables diferentes, pero relacionadas, como si fueran una unidad. Observe la siguiente sección declarativa:

```
DECLARE
      V_idestudiante NUMBER(5);
      V_Nombre VARCHAR2(20);
      V_apellido VARCHAR2(20);
```

Estas tres variables tienen una relación lógica entre si. Si declaramos un tipo de registro para estas variables, la relación entre ellas se volverá aparente, y se las podrá manipular de manera conjunta. Consideremos el siguiente ejemplo:

```
DECLARE
-- Definición de un tipo de registro para almacenar
TYPE t_RegEstudiante  IS RECORD(
IdEstudiante NUMBER(5),
Nombre VARCHAR2(20),
Apellido VARCHAR2(20));

-- Declaración de una variable de este tipo
v_estudiante t_RegEstudiante;
```

La sintaxis general para definir el tipo registro es

```
TYPE tipo_registro IS RECORD(
      Campo1 tipo1 [NOT NULL][:=expr1],
      Campo2 tipo2 [NOT NULL][:=expr2],
      ....
      Campon tipon [NOT NULL][:=exprn]);
```

Donde tipo_registro es el nombre del nuevo tipo, campo1 a campon son los nombres de los campos incluidos en el registro, y tipo1 a tipon son los tipos correspondientes a dichos campos.
Para hacer referencia a uno de los campos de un registro, se emplea la denominada notación de punto. La sintaxis es:
Nombre_registro.nombrecampo

Ejemplo:

```
V_estudiante.idestudiante:=1;
V_estudiante.nombre:='JUAN';
```

Asignación de registros.

Se pueden asignar dos registros siempre que sean del mismo tipo.

Utilización de %ROWTYPE

Resulta habitual definir un registro en PL/SQL con los mismos tipos que una fila de una base de datos.
Para facilitar esta tarea, PL/SQL incorpora el operador **%ROWTYPE**. De forma similar a %TYPE, %ROWTYPE devuelve un tipo basándose en la definición de la tabla.
Por ejemplo:

```
DECLARE
      V_estudiante estudiante%ROWTYPE;
```

Esto declararía un registro cuyos campos se corresponderían con las columnas de la tabla estudiante.

3. Colecciones

Una colección es un grupo ordenado de elementos todos ellos del mismo tipo. Cada elemento tiene un índice que determina su posición dentro de la colección. Tipos:

Tablas indexadas

Introducidas en la versión 7 de Oracle. Son similares sintácticamente a las matrices de C o Java. Para declarar una tabla indexada, primero se define el **tipo de tabla** dentro de un bloque PL/SQL y luego se declara una variable de dicho tipo.
La sintaxis general para definir un tipo tabla indexada es:

```
TYPE tipotabla IS TABLE OF tipo INDEX BY BINARY_INTEGER;
```

Donde tipotabla es el nombre del nuevo tipo que se está definiendo y tipo es un tipo predefinido o una referencia a un tipo efectuada mediante %TYPE o %ROWTYPE.

La cláusula INDEX BY BINARY_INTEGER es obligatoria como parte de la definición de tabla. Esta cláusula no está disponible para las tablas anidadas.

Una vez que se han declarado el tipo y la variable, se puede hacer referencia a un elemento individual de la tabla PL/SQL usando la sintaxis:

Nombretabla(índice)

Donde nombretabla es el nombre de la tabla e índice es una variable de tipo BINARY_INTEGER o una variable o expresión que puedan convertirse al tipo BINARY_INTEGER.

Ejemplo:

```
DECLARE
  TYPE NameTab IS TABLE OF students.first_name%TYPE
    INDEX BY BINARY_INTEGER;
  TYPE DateTab IS TABLE OF DATE
    INDEX BY BINARY_INTEGER;
  v_Names NameTab;
  v_Dates DateTab;
BEGIN
  v_Names(1) := 'Scott';
  v_Dates(-4) := SYSDATE - 1;
END;
```

Tablas indexadas y matrices C o Java.

Consideremos el siguiente ejemplo:

```
DECLARE
  TYPE CharacterTab IS TABLE OF VARCHAR2(10)
    INDEX BY BINARY_INTEGER;
  v_Characters CharacterTab;
BEGIN
-- Asigna valores a tres elementos de la tabla.
-- Observad que los valores de la clave no son secuenciales.
 v_Characters(0) := 'Harold';
  v_Characters(-7) := 'Susan';
  v_Characters(3) := 'Steve';
END;
```

Aunque la asignación de valores a elementos de tabla es sintácticamente similar a la asignación de valores en matrices C o Java, las tablas indexadas se implementan de forma distinta. Una tabla indexada es similar a una tabla de base de datos con dos columnas: **clave y valor**. El tipo de clave es BINARY_INTEGER y el tipo de valor puede ser cualquier tipo de datos especificado en la definición: varchar2(10) en el ejemplo anterior.
Después de las asignaciones del ejemplo, la estructura de datos de la tabla v_characters seria la siguiente:

Clave	Valor
0	Harold
-7	Susan
3	Steve

Puntos a tener en cuenta sobre las tablas indexadas:

- Las tablas indexadas no están restringidas. El único límite (aparte de la memoria disponible) sobre el número de filas es que la clave es de tipo BINARY_INTEGER y por tanto debe estar en el rango (-2147483647.. +2147483647)
- Los elementos contenidos en una tabla indexada no necesariamente se encuentran en un orden particular y no tienen porque almacenarse de forma continua en memoria.
- Las claves utilizadas para una tabla indexada no tienen por qué ser secuenciales (aunque en la mayoría de los casos seguro que lo haremos).

Elementos inexistentes

La asignación de un valor al elemento i de una tabla indexada crea dicho elemento si todavía no existe. Si se hace una referencia a un elemento i antes de ser usado, el motor de PL/SQL devolverá el error "ORA-1403: datos no encontrados".

Tablas indexadas de registros.

El siguiente ejemplo ilustra una tabla indexada de registros:

```
DECLARE
  TYPE StudentTab IS TABLE OF students%ROWTYPE
    INDEX BY BINARY_INTEGER;
  -- Cada elemento de v_studentse es un registro
  v_Students StudentTab;
BEGIN
  /* Recupera el registro con id = 10,001 y lo almacena en
     v_Students(10001). */
  SELECT *
    INTO v_Students(10001)
    FROM students
    WHERE id = 10001;

  -- Asigna directamente valores a  v_Students(1)
  v_Students(1).first_name := 'Larry';
  v_Students(1).last_name := 'Lemon';
END;
```

Dado que cada elemento de la tabla es un registro, se puede hacer referencia a los campos contenidos en dicho registro con la siguiente sintaxis:
Tabla(índice).campo

Tablas anidadas

Introducidas con Oracle8. La funcionalidad básica de una tabla anidada es la misma que la de una tabla indexada. Sin embargo, las tablas anidadas deben crearse con claves secuenciales, y las claves no pueden ser negativas. Además las tablas anidadas pueden almacenarse en la base de datos, mientras que las tablas indexadas no. El rango de valores para el índice en una tabla anidada es 1 .. 2147483647. La sintaxis para crear una tabla anidada es:

```
TYPE tipo_tabla  TABLE OF tipo_elemento[NOT NULL];
```

Donde tipo tabla es el nombre del nuevo tipo y tipo_elemento es el tipo de cada elemento de la tabla anidada. La única diferencia sintáctica entre las tablas indexadas y las tablas anidadas es la presencia de la cláusula INDEX BY BINARY_INTEGER. Si ésta cláusula no existe, entonces se trata de una tabla anidada.

Inicialización de una tabla anidada

Cuando se crea una tabla indexada, pero todavía no contiene ningún elemento, simplemente está vacía. Cuando se declara una tabla anidada y todavía no contiene ningún elemento, esta se inicializa para ser atómicamente nula. Debido a esto podemos aplicar el operador IS NULL a una tabla anidada pero no a una indexada. Si se intenta añadir un elemento a una tabla anidada nula, se producirá el error ORA-6531 referencia a una colección sin inicializar.
Para inicializar una tabla anidada utilizaremos el constructor que tiene el mismo nombre que la tabla.

Ejemplo:

```
DECLARE
  TYPE NumbersTab IS TABLE OF NUMBER;

  -- Crear una tabla con un elemento
  v_Tab1 NumbersTab := NumbersTab(-1);

  -- Crear una tabla con 5 elementos
  v_Primes NumbersTab := NumbersTab(1, 2, 3, 5, 7);

  -- Crear una tabla sin elementos
  v_Tab2 NumbersTab := NumbersTab();
BEGIN
  -- Asignar a v_Tab1(1). Se reemplaza el antiguo valor.
  v_Tab1(1) := 12345;

  -- Imprimir el contenido de la tabla
  FOR v_Count IN 1..5 LOOP
    DBMS_OUTPUT.PUT(v_Primes(v_Count) || ' ');
  END LOOP;
  DBMS_OUTPUT.NEW_LINE;
END;
```

Aunque una tabla no esté restringida, no se puede asignar un valor a un elemento que no exista, y que por tanto diera lugar a que la tabla aumentara de tamaño. Si se intenta obtendremos el error ORA-6533: índice fuera de cuenta.
Para aumentar el tamaño de una tabla anidada utilizaremos el método EXTEND que veremos posteriormente.

Varrays

A partir de Oracle8 y versiones posteriores. Un varray (matriz de longitud variable) es un tipo de dato similar a una matriz en C o Java. Sintácticamente, a un varray se accede de forma muy similar a como se hace con una tabla indexada o anidada. Sin embargo, un varray tiene un límite superior fijo para su tamaño, que se especifica como parte de la declaración de tipo. Los elementos un varray se almacenan de forma contigua en memoria y los elementos se numeran a partir de la posición 1.

Declaración de un varray

```
TYPE nombre_tipo IS {VARRAY| VARYING ARRAY}(tamaño_máximo) OF
tipo_elemento [NOT NULL];
```

Donde nombre_tipo es el nuevo tipo de array, tamaño_máximo es un entero que especifica el número máximo de elementos del array y tipo elemento es un escalar, registro o tipo de objeto PL/SQL. El tipo tipo_elemento puede especificarse también utilizando %TYPE.
Antes de Oracle9i, tipo_elemento tampoco podía ser de tipo TABLE ni VARRAY, esta restricción se ha levantado en Oracle9i.

Ejemplos de declaración:

```
DECLARE
  -- Algunos tipos de varray válidos.
  -- Lista de números no nulos.
  TYPE NumberList IS VARRAY(10) OF NUMBER(3) NOT NULL;

  -- Una lista de registros PL/SQL.
  TYPE StudentList IS VARRAY(100) OF students%ROWTYPE;

  -- Lista de objetos.
  TYPE ObjectList is VARRAY(25) OF MyObject;
BEGIN
  NULL;
END;
```

Inicialización de un varray

De forma similar a las tablas, los varrays se inicializan utilizando un constructor. El número de argumentos pasados al constructor se convierte en la longitud inicial del varray y debe ser menor o igual que la longitud máxima especificada en el tipo de varray.

Ejemplo:

```
DECLARE
  -- Define un tipo VARRAY.
  TYPE Numbers IS VARRAY(20) OF NUMBER(3);
  -- Declaración de un varray NULL.
  v_NullList Numbers;
-- Este varray tiene dos elementos
  v_List1 Numbers := Numbers(1, 2);

-- Este varray tiene un elemento, que es NULL.
  v_List2 Numbers := Numbers(NULL);
BEGIN
  IF v_NullList IS NULL THEN
    DBMS_OUTPUT.PUT_LINE('v_NullList is NULL');
  END IF;

  IF v_List2(1) IS NULL THEN
    DBMS_OUTPUT.PUT_LINE('v_List2(1) is NULL');
  END IF;
END;
```

Manipulación de los elementos de un varray.

Como en las tablas anidadas, el tamaño inicial de un varray se establece mediante el número de elementos usados en el constructor utilizado para declararlo. Si se hacen asignaciones a los elementos que quedan fuera de este rango se producirá el error ORA- -6553: Subíndice fuera de rango. El siguiente ejemplo muestra esta situación:

```
DECLARE
```

```
  TYPE Strings IS VARRAY(5) OF VARCHAR2(10);

-- Declara un varray con 3 elementos
  v_List Strings := Strings('One', 'Two', 'Three');
BEGIN
  -- El subíndice varia entre 1 y 3 por tanto esta asignación es
legal.
  v_List(2) := 'TWO';

-- Subíndice fuera de rango, genera el error ORA-6533
  v_List(4) := '!!!';
END;
```

Los intentos de ampliar el varray sobrepasando el tamaño máximo, generarán el error ORA-6532 subíndice fuera del límite

```
DECLARE
  TYPE Strings IS VARRAY(5) OF VARCHAR2(10);
  -- Declara un varray con 4 elementos
  v_List Strings :=
    Strings('One', 'Two', 'Three', 'Four');
BEGIN
  -- Subíndice entre 1 y 4, por tanto asignación legal.
  v_List(2) := 'TWO';

  -- Ampliamos el varray a 5 elementos y asignamos valor.
  v_List.EXTEND;
  v_List(5) := 'Five';

  -- Intento de ampliación a 6 elementos.
  -- Se generará  el error ORA-6532.
  v_list.EXTEND;
END;
```

Colecciones multinivel

Solo para Oracle9i y versiones posteriores.

Podemos utilizar, a partir de Oracle9i, colecciones de más de una dimensión, es decir, una colección de colecciones, también llamadas colecciones multinivel.
Se declaran del mismo modo que las colecciones de una dimensión, con la diferencia de que el tipo de la colección es una colección en si misma.

Ejemplos:

```
DECLARE
  -- Primero declaramos una tabla de números indexada.
  TYPE t_Numbers IS TABLE OF NUMBER
    INDEX BY BINARY_INTEGER;

  -- Ahora se declara un tipo que es una tabla indexada de t_Numbers.
  --  Esto es una colección multinivel
  TYPE t_MultiNumbers IS TABLE OF t_Numbers
    INDEX BY BINARY_INTEGER;

  -- También podemos tener un varray de una tabla indexada
  TYPE t_MultiVarray IS VARRAY(10) OF t_Numbers;

-- O una tabla anidada
  TYPE t_MultiNested IS TABLE OF t_Numbers;

  v_MultiNumbers t_MultiNumbers;

-- Para acceder a un elemento de una colección multinivel se emplean
dos índices.
BEGIN
  v_MultiNumbers(1)(1) := 12345;
END;
```

Comparación entre los tipos de colecciones.

Varrays y tablas anidadas

Similitudes:

- Ambos tipos (más las tablas indexadas) permiten el acceso a elementos individuales utilizando notación de subíndices PL/SQL.
- Ambos tipos pueden almacenarse en tablas de la base de datos (cuando se declaran fuera de un bloque PL/SQL).

- Se pueden aplicar a ambas estructuras los métodos de colecciones que veremos más adelante.

Diferencias

- Los varrays tienen un tamaño máximo, mientras que las tablas anidadas no tienen un tamaño máximo explícito.
- Cuando se almacenan en la base de datos, los varrays conservan el orden y los valores de los subíndices de los elementos, mientras que las tablas anidadas no.

Tablas anidadas y tablas indexadas

Similitudes:

- Ambos tipos de datos de tabla tienen la misma estructura.
- Se accede a los elementos individuales de ambas utilizando notación de subíndice.
- Los métodos disponibles para las tablas anidadas incluyen todos los atributos de las tablas indexadas.

Diferencias:

- Las tablas anidadas pueden manipularse utilizando SQL y se pueden almacenar en la base de datos, mientras que las tablas indexadas no.
- Las tablas anidadas tienen un rango legal de subíndices de 1 a 2147834647, mientras que las indexadas tienen un rango de –2147834647 a 2147834647.
- Las tablas anidadas pueden ser atómicamente nulas, circunstancia que se puede comprobar con el operador IS NULL.
- Las tablas anidadas tienen métodos adicionales disponibles, tales como EXTEND y TRIM, serán descritos más adelante.

Colecciones en la base de datos

Tanto los varrays como las tablas anidadas pueden almacenarse como columnas de una tabla de la base de datos.
Desde nuestro punto de vista, aunque esta posibilidad pueda ser útil en algún caso, atenta contra la primera forma normal, puesto que tendríamos un atributo multivaluado. Por esta razón no veremos las técnicas necesarias para su implementación. No obstante, si se quiere obtener más información, se le remite al capítulo 8 del libro Oracle9i Programación PL/SQL de la editorial McGraw-Hill o bien a la documentación especifica de Oracle en el libro PL/SQL guía y referencia en el capítulo colecciones y registros.

Métodos de colecciones.

Las tablas anidadas y los varrays son tipos de objeto y, como tales, tienen definidos una serie de métodos. Por su parte, las tablas indexadas tienen atributos. Tanto los métodos como los atributos se invocan utilizando la misma sintaxis:

Instancia_colección.método_o_atributo

Donde instancia_colección es una variable de colección y método_o_atributo es uno de los métodos o atributos que vamos a describir a continuación. Estos métodos solo pueden ser llamados desde instrucciones procedimentales y no desde instrucciones SQL.

EXISTS(n)

Se utiliza para determinar si existe en la colección el elemento indicado. La sintaxis es EXISTS(n) donde n es una expresión entera. Devuelve TRUE si el elemento especificado por n existe, incluso aunque tenga el valor NULL. Si n esta fuera de rango, EXISTS devuelve FALSE.

Ejemplo de uso de exists:

```
DECLARE
TYPE NumTab IS TABLE OF NUMBER;
TYPE NumVar IS VARRAY(25) OF NUMBER;
TYPE NumTabIndexBy IS TABLE OF NUMBER INDEX BY BINARY_INTEGER;

  v_NestedTable NumTab := NumTab(-7, 14.3, 3.14159, NULL, 0);
  v_Count BINARY_INTEGER := 1;
  v_IndexByTable NumTabIndexBy;
BEGIN
-- Ejecuta un bucle sobre v_NestedTable y presenta los elementos
usando
--  EXISTS para indicar el final de bucle.
LOOP
    IF v_NestedTable.EXISTS(v_Count) THEN
      DBMS_OUTPUT.PUT_LINE(
        'v_NestedTable('|| v_Count || '): ' ||v_NestedTable(v_Count));
      v_Count := v_Count + 1;
    ELSE
      EXIT;
    END IF;
  END LOOP
;
 -- Asigna los mismos elementos a la tabla indexada.
v_IndexByTable(1) := -7;
  v_IndexByTable(2) := 14.3;
  v_IndexByTable(3) := 3.14159;
  v_IndexByTable(4) := NULL;
  v_IndexByTable(5) := 0;

  -- Ejecuta un bucle similar
  v_Count := 1;
  LOOP
```

```
    IF v_IndexByTable.EXISTS(v_Count) THEN
      DBMS_OUTPUT.PUT_LINE(
        'v_IndexByTable(' || v_Count || '): ' ||
        v_IndexByTable(v_Count));
      v_Count := v_Count + 1;
    ELSE
      EXIT;
    END IF;
  END LOOP;
END;
/
La salida producida por este código sería:
v_NestedTable(1): -7
v_NestedTable(2): 14,3
v_NestedTable(3): 3,14159
v_NestedTable(4):
v_NestedTable(5): 0
v_IndexByTable(1): -7
v_IndexByTable(2): 14,3
v_IndexByTable(3): 3,14159
v_IndexByTable(4):
v_IndexByTable(5): 0
```

COUNT

Devuelve el número de elementos que tiene actualmente una colección, como un valor entero.

Ejemplo de uso:

```
DECLARE
  TYPE NumTab IS TABLE OF NUMBER;
  TYPE NumVar IS VARRAY(25) OF NUMBER;
  TYPE NumTabIndexBy IS TABLE OF NUMBER INDEX BY BINARY_INTEGER;
  v_NestedTable NumTab := NumTab(1, 2, 3);
  v_Varray NumVar := NumVar(-1, -2, -3, -4);
  v_IndexByTable NumTabIndexBy;
BEGIN
-- Primero añadimos algunos elementos a la tabla indexada.
-- Obsérvese que los valores del índice no son secuenciales
  v_IndexByTable(1) := 1;
  v_IndexByTable(8) := 8;
  v_IndexByTable(-1) := -1;
  v_IndexByTable(100) := 100;

  -- Se muestran el número de elementos de cada colección.
  DBMS_OUTPUT.PUT_LINE(
    'Nested Table Count: ' || v_NestedTable.COUNT);
  DBMS_OUTPUT.PUT_LINE(
    'Varray Count: ' || v_Varray.COUNT);
  DBMS_OUTPUT.PUT_LINE(
```

```
    'Index-By Table Count: ' || v_IndexByTable.COUNT);
END;
```

Salida producida:

Nested Table Count: 3
Varray Count: 4
Index-By Table Count: 4

LIMIT

Devuelve el número máximo actual de elementos de una colección.
Puesto que las tablas anidadas no tienen tamaño máximo, LIMIT siempre devuelve NULL cuando se aplica a una tabla anidada. LIMIT no es válido para tablas indexadas. Ejemplo de uso:

```
DECLARE
  TYPE NumTab IS TABLE OF NUMBER;
  TYPE NumVar IS VARRAY(25) OF NUMBER;
  v_Table NumTab := NumTab(1, 2, 3);
  v_Varray NumVar := NumVar(1234, 4321);
BEGIN
  -- Mostramos el número de valores y el limite para las colecciones.
  DBMS_OUTPUT.PUT_LINE('Varray limit: ' || v_Varray.LIMIT);
  DBMS_OUTPUT.PUT_LINE('Varray count: ' || v_Varray.COUNT);
  IF v_Table.LIMIT IS NULL THEN
    DBMS_OUTPUT.PUT_LINE('Table limit is NULL');
  ELSE
    DBMS_OUTPUT.PUT_LINE('Table limit: ' || v_Table. LIMIT);
  END IF;
  DBMS_OUTPUT.PUT_LINE('Table count: ' || v_Table.COUNT);
END;
```

Salida producida:

Varray limit: 25
Varray count: 2
Table limit is NULL
Table count: 3

FIRST Y LAST.

FIRST devuelve el índice del primer elemento de una colección y LAST devuelve el índice del último elemento.
En un varray FIRST siempre devuelve 1 y LAST siempre devuelve el valor de COUNT, ya que un varray es denso y los elementos no pueden borrarse.
FIRST y LAST pueden usarse junto con NEXT y PRIOR para recorrer mediante un bucle una colección tal y como se muestra a continuación:

NEXT Y PRIOR

Se usan para incrementar y decrementar la clave (índice) de una colección.
La sintaxis es:

- NEXT(n)
- PRIOR(n)

Donde n es una expresión entera. NEXT(n)devuelve la clave del elemento inmediatamente posterior al elemento n y PRIOR(n) devuelve la clave del elemento inmediatamente anterior a n. Si no existen ele elemento anterior o posterior, PRIOR y NEXT devolverán NULL.

Ejemplo de uso de NEXT, PRIOR, junto con FIRST Y LAST:

```
DECLARE
  TYPE CharTab IS TABLE OF CHAR(1);
  v_Characters CharTab :=
    CharTab('M', 'a', 'd', 'a', 'm', ',', ' ',
            'I', '''', 'm', ' ', 'A', 'd', 'a', 'm');

  v_Index INTEGER;
BEGIN
  --  Bucle hacia delante a través de la tabla.
  v_Index := v_Characters.FIRST;
  WHILE v_Index <= v_Characters.LAST LOOP
    DBMS_OUTPUT.PUT(v_Characters(v_Index));
    v_Index := v_Characters.NEXT(v_Index);
  END LOOP;
  DBMS_OUTPUT.NEW_LINE;

  -- Bucle hacia atrás a través de la tabla
  v_Index := v_Characters.LAST;
  WHILE v_Index >= v_Characters.FIRST LOOP
4.     DBMS_OUTPUT.PUT(v_Characters(v_Index));
    v_Index := v_Characters.PRIOR(v_Index);
  END LOOP;
  DBMS_OUTPUT.NEW_LINE;
END;
```

Salida del programa:

Madam, I'm Adam

madA m'I ,madaM

EXTEND

Se usa para añadir elementos al final de una tabla anidada. No es válido para tablas indexadas. EXTEND tiene tres formatos:

- EXTEND
- EXTEND(n)
- EXTEND(n,i)

EXTEND sin argumentos, simplemente añade un elemento de valor NULL al final de la colección, con el índice LAST+1.

EXTEND(n) añade n elementos con valor NULL al final de la tabla.

EXTEND(n,i) añade n copias del elemento i al final de la tabla.

Si la colección se ha creado con una restricción NOT NULL, entonces sólo se puede usar este último formato, dado que no añade un elemento nulo.

Debido a que una tabla anidada no tiene un tamaño máximo explícito, puede llamarse a EXTEND con el argumento n según se necesite (el tamaño máximo es 2gb, sujeto a restricciones de memoria). Sin embargo, un varray solo puede ampliarse hasta el tamaño máximo declarado, por lo que n puede ser (LIMIT-COUNT), como máximo.

Ejemplo de uso de EXTEND:

```
DECLARE
 TYPE NumTab IS TABLE OF NUMBER;
 TYPE NumVar IS VARRAY(25) OF NUMBER;

  v_NumbersTab NumTab := NumTab(1, 2, 3, 4, 5);
  v_NumbersList NumVar := NumVar(1, 2, 3, 4, 5);
BEGIN
  BEGIN
-- Esta asignación generara la excepción: SUBSCRIPT_BEYOND_COUNT,
-- Ya que v_NumbersTab tiene solo 5 elementos.
    v_NumbersTab(26) := -7;
  EXCEPTION
    WHEN SUBSCRIPT_BEYOND_COUNT THEN
      DBMS_OUTPUT.PUT_LINE(
        'ORA-6533 raised for assignment to v_NumbersTab(26)');
  END;
  -- Podemos solucionarlo añadiendo 30 elementos adicionales a
v_NumbersTab.
v_NumbersTab.EXTEND(30);

  -- Y ahora hacer la asignación.
  v_NumbersTab(26) := -7;

-- Un varray solo se puede ampliar hasta el tamaño máximo dado también
por LIMIT.
-- Lo siguiente generará la excepción :SUBSCRIPT_OUTSIDE_LIMIT:
  BEGIN
    v_NumbersList.EXTEND(30);
```

```
  EXCEPTION
    WHEN SUBSCRIPT_OUTSIDE_LIMIT THEN
      DBMS_OUTPUT.PUT_LINE(
        'ORA-6532 raised for v_NumbersList.EXTEND(30)');
  END;

  -- Esto es legal
  v_NumbersList.EXTEND(20);

  -- Ahora podemos asignar un valor al elemento de mayor subíndice del
varray.
v_NumbersList(25) := 25;
END;
```

Salida del programa:

ORA-6533 raised for assignment to v_NumbersTab(26)
ORA-6532 raised for v_NumbersList.EXTEND(30)

EXTEND opera sobre el tamaño interno de una colección, lo que incluye cualquier elemento borrado para una tabla anidada.
Cuando se borra un elemento (utilizando el método DELETE, descrito a continuación), se eliminan los datos correspondientes a este elemento, aunque se conserva la clave.

El siguiente ejemplo ilustra la interacción entre EXTEND y DELETE:

```
DECLARE
  TYPE NumTab IS TABLE OF NUMBER;
  -- Inicializamos una tabla anidada de 5 elementos
   v_Numbers NumTab := NumTab(-2, -1, 0, 1, 2);

  -- Procedimiento local para imprimir la tabla.
  -- Observe el uso de  FIRST, LAST, and NEXT.
  PROCEDURE Print(p_Table IN NumTab) IS
    v_Index INTEGER;
  BEGIN
    v_Index := p_Table.FIRST;
    WHILE v_Index <= p_Table.LAST LOOP
      DBMS_OUTPUT.PUT('Element ' || v_Index || ': ');
      DBMS_OUTPUT.PUT_LINE(p_Table(v_Index));
      v_Index := p_Table.NEXT(v_Index);
    END LOOP;
  END Print;

BEGIN
  DBMS_OUTPUT.PUT_LINE('Al comienzo, v_Numbers contiene');
  Print(v_Numbers);

  -- Borra el elemento 3.  Esto borra el  '0', pero deja un hueco
donde estaba.
  v_Numbers.DELETE(3);
```

```
  DBMS_OUTPUT.PUT_LINE('Después de la orden delete, v_Numbers
contiene');
  Print(v_Numbers);

  -- Añade 2 copias del elemento 1 a la tabla. Añadirá los elementos 6  7.
  v_Numbers.EXTEND(2, 1);

  DBMS_OUTPUT.PUT_LINE('Después de extend, v_Numbers contiene');
  Print(v_Numbers);

  DBMS_OUTPUT.PUT_LINE('v_Numbers.COUNT = ' || v_Numbers.COUNT);
  DBMS_OUTPUT.PUT_LINE('v_Numbers.LAST = ' || v_Numbers.LAST);
END;
/
```

La salida sería:

Al comienzo, v_Numbers contiene

Element 1: -2

Element 2: -1

Element 3: 0

Element 4: 1

Element 5: 2

Después de la orden delete, v_Numbers contiene

Element 1: -2

Element 2: -1

Element 4: 1

Element 5: 2

Después de extend, v_Numbers contiene

Element 1: -2

Element 2: -1

Element 4: 1

Element 5: 2

Element 6: -2

Element 7: -2

v_Numbers.COUNT = 6

v_Numbers.LAST = 7

TRIM

Se utiliza para eliminar elementos del final de una tabla anidada o de un varray. Tiene dos formatos:

- TRIM
- TRIM(n)

Sin argumentos, TRIM elimina un elemento del final de la colección.

En el otro caso, se eliminan N ELEMENTOS. Si n es mayor que COUNT, se genera la excepción "SUBSCRIPT_BEYOND_COUNT.

Después de aplicar TRIM, COUNT será menor, ya que se han eliminado elementos.

De forma similar a EXTEND, TRIM opera sobre el tamaño interno de una colección, incluyendo cualquier elemento eliminado con DELETE.

Lo vemos en el siguiente ejemplo:

```
DECLARE
  TYPE NumTab IS TABLE OF NUMBER;
  -- Inicializa una tabla con 7 elementos.
  v_Numbers NumTab := NumTab(-3, -2, -1, 0, 1, 2, 3);

-- Procedimiento local para imprimir la tabla.
  PROCEDURE Print(p_Table IN NumTab) IS
    v_Index INTEGER;
  BEGIN
    v_Index := p_Table.FIRST;
    WHILE v_Index <= p_Table.LAST LOOP
      DBMS_OUTPUT.PUT('Element ' || v_Index || ': ');
      DBMS_OUTPUT.PUT_LINE(p_Table(v_Index));
      v_Index := p_Table.NEXT(v_Index);
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('COUNT = ' || p_Table.COUNT);
    DBMS_OUTPUT.PUT_LINE('LAST = ' || p_Table.LAST);
  END Print;

BEGIN
  DBMS_OUTPUT.PUT_LINE('Al comienzo, v_Numbers contiene');
  Print(v_Numbers);

  -- Borramos el elemento 6.
  v_Numbers.DELETE(6);
  DBMS_OUTPUT.PUT_LINE('Después de DELETE, v_Numbers contiene');
  Print(v_Numbers);
-- Borra los 3 últimos elementos. Esto borra el 1 y el 3 y también el
(ahora vacío)
-- donde estaba el 2.
v_Numbers.TRIM(3);
  DBMS_OUTPUT.PUT_LINE('Después de TRIM, v_Numbers contiene');
  Print(v_Numbers);
END;
/
```

La salida sería:

Al comienzo, v_Numbers contiene

Element 1: -3
Element 2: -2
Element 3: -1
Element 4: 0
Element 5: 1
Element 6: 2
Element 7: 3
COUNT = 7
LAST = 7

Después de DELETE, v_Numbers contiene

Element 1: -3
Element 2: -2
Element 3: -1
Element 4: 0
Element 5: 1
Element 7: 3
COUNT = 6
LAST = 7

Después de TRIM, v_Numbers contiene

Element 1: -3
Element 2: -2
Element 3: -1
Element 4: 0
COUNT = 4
LAST = 4

DELETE

Eliminará uno o más elementos de una tabla indexada o una tabla anidada.
DELETE no tiene ningún efecto sobre un varray, ya que su tamaño es fijo, de hecho es ilegal llamar a DELETE para un varray.

DELETE tiene 3 formatos:

- DELETE
- DELETE(n)
- DELETE (n,m)

DELETE sin argumentos eliminará la tabla completa.

DELETE(n) eliminará el elemento con el índice n.

DELETE(m,n) eliminará todos los elementos entre los índices m y n.

Después de DELETE, COUNT devolverá un valor menor, que refleja el nuevo tamaño de la tabla anidada. Si el elemento que se va a borrar no existe, DELETE no generará un error, sino que, simplemente se saltará dicho elemento.

Ejemplo de uso de DELETE:

```
DECLARE
   TYPE NumTab IS TABLE OF NUMBER;
  -- Inicializa una tabla con 10 elementos.
  v_Numbers NumTab := NumTab(10, 20, 30, 40, 50, 60, 70, 80, 90, 100);
-- Procedimiento local para imprimir la tabla.
  PROCEDURE Print(p_Table IN NumTab) IS
    v_Index INTEGER;
  BEGIN
    v_Index := p_Table.FIRST;
    WHILE v_Index <= p_Table.LAST LOOP
      DBMS_OUTPUT.PUT('Element ' || v_Index || ': ');
      DBMS_OUTPUT.PUT_LINE(p_Table(v_Index));
      v_Index := p_Table.NEXT(v_Index);
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('COUNT = ' || p_Table.COUNT);
    DBMS_OUTPUT.PUT_LINE('LAST = ' || p_Table.LAST);
  END Print;
BEGIN
  DBMS_OUTPUT.PUT_LINE('Al comienzo, v_Numbers contiene');
  Print(v_Numbers);
  -- Borramos el elemento 6.
  DBMS_OUTPUT.PUT_LINE('Después de delete(6), v_Numbers contiene');
  v_Numbers.DELETE(6);
  Print(v_Numbers);
  -- Borramos los elementos del 7 al  9.
  DBMS_OUTPUT.PUT_LINE('Después de delete(7,9), v_Numbers contiene');
  v_Numbers.DELETE(7,9);
  Print(v_Numbers);
END;
```

La salida sería:
Al comienzo, v_Numbers contiene

Element 1: 10

Element 2: 20

Element 3: 30

Element 4: 40

Element 5: 50

Element 6: 60

Element 7: 70

Element 8: 80

Element 9: 90

Element 10: 100

COUNT = 10

LAST = 10

Depués de delete(6), v_Numbers contiene

Element 1: 10

Element 2: 20

Element 3: 30

Element 4: 40

Element 5: 50

Element 7: 70

Element 8: 80

Element 9: 90

Element 10: 100

COUNT = 9

LAST = 10

Después de delete(7,9), v_Numbers contiene

Element 1: 10

Element 2: 20

Element 3: 30

Element 4: 40

Element 5: 50

Element 10: 100

COUNT = 6

LAST = 10

4. SQL EN PL/SQL

Estudiaremos las instrucciones SQL permitidas en PL/SQL y las instrucciones de control de transacciones que garantizan la consistencia de los datos.

Uso de SQL en PL/SQL.

Las únicas instrucciones SQL permitidas en un programa PL/SQL son las del lenguaje de manipulación de datos y las de control de transacciones. En particular, las instrucciones de definición de datos son ilegales. La razón tiene que ver con la forma en que se ha diseñado PL/SQL.
En general, un lenguaje de programación puede acoplar las variables de dos maneras: acoplamiento **temprano** o **tardío**.
El acoplamiento de una variable es el proceso de determinar la zona de almacenamiento asociada con un identificador de programa. En PL/SQL el acoplamiento también requiere comprobar en la base de datos los permisos de acceso para el objeto de esquema referenciado.
Un lenguaje con **acoplamiento temprano**, realiza el acoplamiento de variables durante la fase de compilación, mientras que un lenguaje con **acoplamiento tardío** pospone esta tarea hasta el momento de la ejecución. El acoplamiento temprano hace que la fase de compilación tarde más tiempo, pero la ejecución será más rápida**. PL/SQL fue diseñado para utilizar acoplamiento temprano.**
Como consecuencia de esto no pueden emplearse instrucciones de definición de datos: puesto que una instrucción de definición de datos modifica los objetos de la base de datos, los permisos deberían ser verificados de nuevo, dicha verificación requeriría un acoplamiento de los identificadores, lo que ya ha sido realizado en la fase de compilación.
Veamos esto a través del siguiente ejemplo:

```
BEGIN
  CREATE TABLE temp_table(
      Num_value NUMBER,
      Char_value CHAR(10));
INSERT INTO temp_table values(10,'Hola');
END;
```

Para poder compilar este segmento, se necesita acoplar el identificador temp_table. Este proceso comprobará si la tabla existe. Sin embargo, la tabla no existirá hasta que el bloque sea ejecutado, y como el bloque no puede llegar a compilarse, no hay manera de que se pueda ejecutar.

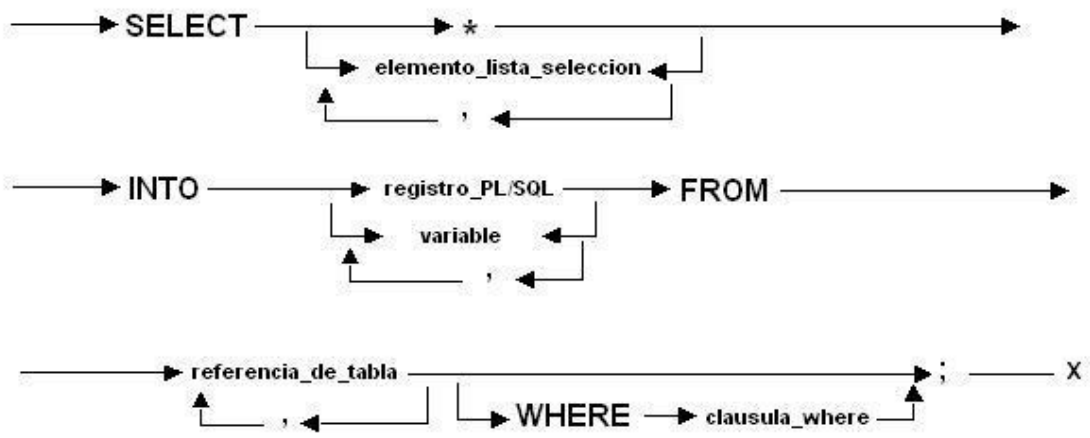
No obstante existe una técnica que permite que se ejecuten todas las instrucciones SQL válidas, incluidas las definición de datos, desde PL/SQL: **SQL dinámico**. Esta técnica la veremos más adelante.

Dml en PL/SQL

Las instrucciones de manipulación de datos (DML) permitidas son SELECT, INSERT, UPDATE y DELETE.

SELECT

Extrae datos de la base de datos y los almacena en variables PL/SQL. El formato general de la instrucción es el siguiente:



Elemento_lista_selección: Columna o columnas que se desean seleccionar. Con * recuperaríamos todas las columnas.

Variable: Variable PL/SQL en la que se almacenará un elemento de la lista de selección. Cada variable debe ser compatible con su elemento asociado, y debe haber el mismo número de elementos que de variables de salida.

Registro PL/SQL: Puede utilizarse en lugar de una lista de variables. El registro debe estar compuesto por campos que se correspondan con la lista de selección. Si la lista de selección es simplemente *, y la selección procede de una sola tabla, entonces podría definirse este registro mediante referencia_tabla%ROWTYPE.

Referencia_tabla: Identifica la tabla o tablas de las que se extraerán los datos.

Cláusula_where: Criterios empleados para la consulta.

Se pueden también emplear las cláusulas vistas para la orden select en el tema SQL tales como ORDER BY, GROUP BY, etc.

La forma aquí descrita de la instrucción SELECT, **no debería devolver más de una fila**. La cláusula WHERE será comparada con cada fila de la tabla y, si encaja con más de una fila PL/SQL devolverá el siguiente mensaje de error:

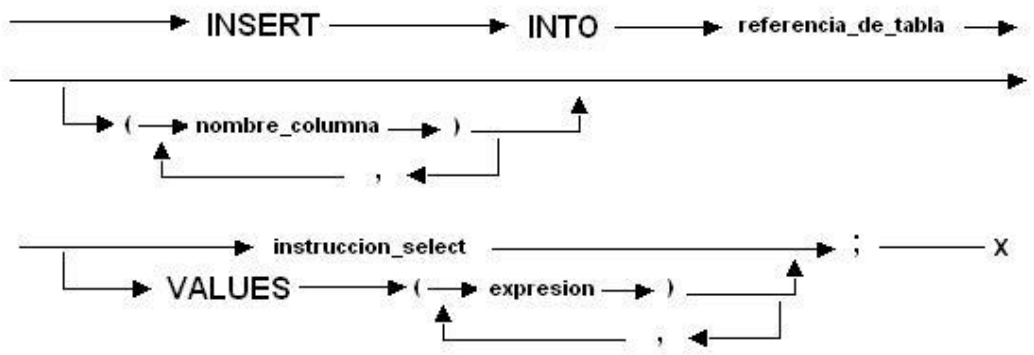
ORA-1427 Consulta de fila única devuelve más de una fila.

Ejemplo de uso:

```
DECLARE
  v_StudentRecord  students%ROWTYPE;
  v_Department     classes.department%TYPE;
  v_Course         classes.course%TYPE;
BEGIN
  SELECT *
    INTO v_StudentRecord
    FROM students
    WHERE id = 10000;
  SELECT department, course
    INTO v_Department, v_Course
    FROM classes
    WHERE room_id = 20003;
END;
```

INSERT

Consta de la siguiente sintaxis:



La cláusula referencia_de_tabla se refiere a una tabla Oracle, nombre_columna a una columna de dicha tabla y expresión es una expresión SQL o PL/SQL .

Ejemplo:

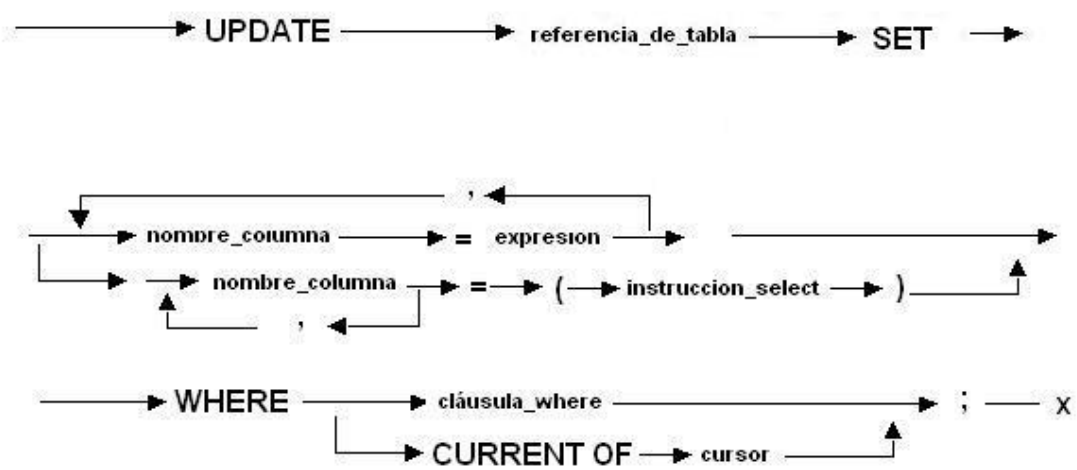
```
DECLARE
  v_StudentID  students.id%TYPE;
BEGIN
  -- Extrae un nuevo número de identificación de estudiante
  SELECT student_sequence.NEXTVAL
    INTO v_StudentID
    FROM dual;

  -- Añade una fila a la tabla students
  INSERT INTO students (id, first_name, last_name)
    VALUES (v_StudentID, 'Timothy', 'Taller');
```

```
-- Añade una segunda fila, pero usa el numero de secuencia
directamente
-- en la instrucción insert.
 INSERT INTO students (id, first_name, last_name)
    VALUES (student_sequence.NEXTVAL, 'Patrick', 'Poll');
END;
```

UPDATE

La sintaxis de la instrucción UPDATE es:



La cláusula referencia_de_tabla hace referencia a la tabla (o a alguna vista actualizable) que se va a actualizar, nombre_columna es una columna cuyo valor se va a cambiar, y expresión es una expresión SQL. Si la instrucción contiene una instrucción_select, la lista de selección debe corresponderse con las columnas de la cláusula SET.
La sintaxis especial CURRENT OF cursor se utiliza en definiciones de cursores que veremos en el siguiente apartado.

Ejemplo de uso:

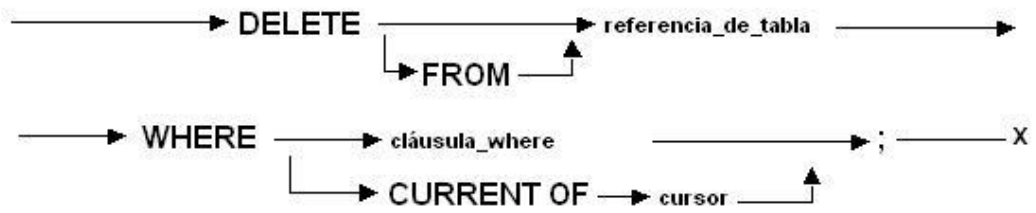
```
DECLARE
  v_Major           students.major%TYPE;
  v_CreditIncrease  NUMBER := 3;
BEGIN
-- Esta instrucción UPDATE añadirá 3 al campo  current_credits de
-- todos los estudiantes de la especialidad de historia.
v_Major := 'History';
  UPDATE students
    SET current_credits = current_credits + v_CreditIncrease
    WHERE major = v_Major;

-- Esta instrucción UPDATE actualizará las dos columnas de temp_table
para todas
```

```
-- las filas.
  UPDATE temp_table
    SET num_col = 1, char_col = 'abcd';
END;
```

DELETE

La instrucción DELETE elimina filas de una tabla de la base de datos, indicando la cláusula WHERE de la instrucción qué filas hay que eliminar. La sintaxis es la siguiente:



La cláusula referencia_de_tabla hace referencia a la tabla (o a ciertas vistas) y la cláusula_where define el conjunto de filas a borrar. La sintaxis especial CURRENT OF cursor se utiliza junto a una definición de cursor, los cursores serán tratados en los próximos apartados.

Ejemplo de uso:

```
DECLARE
  v_StudentCutoff  NUMBER;
BEGIN
  v_StudentCutoff := 10;
-- Borrar todas las clases que no tengan registrados suficientes alumnos.
  DELETE FROM classes
    WHERE current_students < v_StudentCutoff;
--  Borrar cualquier estudiante de Económicas que no tenga créditos.
  DELETE FROM students
    WHERE current_credits = 0
    AND   major = 'Economics';
END;
```

La cláusula WHERE.

Las instrucciones SELECT, UPDATE y DELETE incluyen la cláusula WHERE como parte de sus operaciones. Esta cláusula define los datos que forman el conjunto activo, es decir, el conjunto de filas devueltas por una consulta (SELECT) o sobre las que actúa una instrucción UPDATE o DELETE.
Una cláusula WHERE está compuesta de condiciones, unidas mediante los operadores booleanos, AND y OR. Las condiciones suelen adoptar la forma de comparaciones, como muestra el siguiente ejemplo de instrucción DELETE:

```
DECLARE
  v_Department  CHAR(3);
BEGIN
  v_Department := 'CS';
  -- Borrar todos los cursos de informática.
  DELETE FROM classes
    WHERE department = v_Department;
END;
```

El bloque anterior borra todas las filas de la tabla classes para las que la condición sea verdadera (aquellas en las que la columna department sea 'CS').
Con respecto a este tipo de comparaciones, hay que hacer varias observaciones de gran importancia:

Nombres de variables

Supongamos que cambiamos el nombre de variable v_Department del bloque anterior por Department:

```
DECLARE
 Department  CHAR(3);
BEGIN
  Department := 'CS';
-- Borrar todos los cursos de informática (Computer Science)
  DELETE FROM classes
    WHERE department = Department;
END;
```

El cambio anterior tiene un efecto catastrófico.
El bloque modificado eliminará todas las filas de la tabla classes, no sólo aquellas en las que el departamento sea 'CS'.
La razón se debe a la forma en que se analizan los identificadores en una instrucción PL/SQL. Cuando el motor de PL/SQL encuentra una condición tal como expr1=expr2
Comprueba **primero** si expr1 y expr2 se corresponden con columnas de la tabla con la que se está operando, y después comprueba si se trata de variables del bloque PL/SQL.
Puesto que PL/SQL no diferencia entre mayúsculas y minúsculas, tanto departament como Department en el bloque anterior son asociadas con la columna de la tabla classes, en lugar de con la variable. Esta condición será evaluada como verdadera para todas las filas de la tabla por lo que todas serán borradas. ¡ Muchísimo cuidado con esto !

Comparación de caracteres

Cuando se comparan dos valores de carácter, como sucede en el ejemplo anterior, Oracle puede emplear dos semánticas de comparación distintas: **con relleno de blancos o sin él**. Estas semánticas de comparación difieren en cuanto al modo en que se comparan cadenas de caracteres de diferente longitud. Suponga que estamos dos cadenas de caracteres, cadena1 y cadena2.
Para la semántica con **relleno de blancos**, se empleará el siguiente algoritmo:

- Si cadena1 y cadena2 tienen longitudes distintas, se rellena con blancos el valor de menor longitud, para que la longitud de ambos sea la misma.

- Se comparan ambas cadenas, carácter a carácter, empezando por la izquierda. Supongamos que el carácter de cadena1 es car1, y que el carácter de cadena2 es car2.
- Si ASCII(car1) < ASCII(car2), entonces cadena1 < cadena2. Si ASCII(car1) > ASCII(car2), entonces cadena1 > cadena2. Si ASCII(car1) = ASCII(car2), se continua con el siguiente carácter de cadena1 y cadena2.
- Si se llega al final de cadena1 y cadena2, entonces las dos cadenas son iguales.

Utilizando la semántica de relleno de blancos, todas las condiciones siguientes devolverán el valor TRUE:

'abc'='abc'
'abc '='abc'
'ab'<'abc'
'abcd' >'abcc'

El algoritmo de comparación **sin relleno de blancos** es ligeramente distinto:

- Se comparan ambas cadenas, carácter a carácter, empezando por la izquierda. Supongamos que el carácter de cadena1 es car1, y que el carácter de cadena2 es car2.
- Si ASCII(car1) < ASCII(car2), entonces cadena1 < cadena2. Si ASCII(car1) > ASCII(car2), entonces cadena1 > cadena2. Si ASCII(car1) = ASCII(car2), se continúa con el siguiente carácter de cadena1 y cadena2.
- Si cadena1 acaba antes que cadena2, entonces cadena1<cadena2. Si cadena2 acaba antes que cadena1, entonces cadena1>cadena2.

Utilizando la semántica de comparación de caracteres sin relleno de blancos todas las siguientes comparaciones devolverán TRUE:

'abc'='abc'
'ab'<'abc'
'abcd'>'abcc'

Sin embargo, la siguiente comparación devolvería FALSE, puesto que las cadenas son de diferente longitud.

'abc '='abc'

¿Cuándo se emplea una semántica u otra?

PL/SQL utiliza semántica con relleno de blancos sólo cuando los dos valores que se están comparando son de longitud fila. Si cualquiera de los dos es de longitud variable, se utiliza la semántica sin relleno de blancos. El tipo CHAR es de longitud fija, mientras que VARCHAR2 es de longitud variable. Los literales de cadena se consideran siempre de longitud fija. A continuación, se muestra un ejemplo de los problemas que pueden suceder debido a esto:

```
DECLARE
 v_Department  VARCHAR2(3);
BEGIN
  v_Department := 'CS';
  -- Eliminar todas las clases de Ciencias de la computación
  DELETE FROM classes
    WHERE department = v_Department;
END;
```

Este bloque no borraría ninguna fila, dado que la variable v_Department es de tipo VARCHAR2, en lugar de CHAR.

Para asegurarse que las cláusulas WHERE tienen el efecto deseado, asegurémonos de que las variables en el bloque PL/SQL son del mismo tipo de datos que las columnas de la base de datos con las cuales se compara. El utilizar %TYPE permite garantizar esta condición.

Control de transacciones

Una transacción es una serie de instrucciones SQL que se completan o fallan como una unidad. Las transacciones son un componente estándar de las bases de datos, y sirven para evitar la inconsistencia de los datos.
El ejemplo clásico de este concepto, es el de las transacciones bancarias.
Consideremos las dos siguientes instrucciones SQL, que implementan una transferencia por un importe de **importe_transaccion** euros entre dos cuentas bancarias, denominadas **cuenta_origen** y **cuenta_destino**.

```
UPDATE cuenta
SET saldo=saldo-importe_transaccion
Where numerocuenta=cuenta_origen
UPDATE cuenta
SET saldo=saldo+importe_transaccion
WHERE numerocuenta=cuenta_destino
```

Supongamos que la primera instrucción UPDATE se lleva a cabo con éxito, pero que la segunda falla debido a algún tipo de error (por ejemplo, una caída de la red).

Los datos serían entonces **inconsistentes**: se habría cargado el importe de la transferencia en cuenta_origen, pero dicho importe no se habría abonado en cuenta_destino.

Para prevenir este tipo de situación, lo que se hace es combinar ambas instrucciones en una única transacción, lo que hace que o bien las dos instrucciones se ejecuten, o bien fracasen las dos. Con esto evitamos la aparición de inconsistencia de los datos.

Una transacción comienza con la primera instrucción SQL ejecutada después de terminar la transacción anterior, o con la primera instrucción SQL después de conectarse con la base de datos. La transacción termina con las instrucciones COMMIT o ROLLBACK.

Cuando se envía una instrucción COMMIT a la base de datos, termina la transacción y:

- Se hace permanente todo el trabajo realizado por la transacción.
- Otras sesiones pueden ver los cambios realizados por esta transacción.
- Se liberan los bloqueos establecidos por la transacción.

La sintaxis de la instrucción COMMIT es

COMMIT [WORK].

Hasta que la transacción se confirme mediante COMMIT, sólo la sesión que esté ejecutando la transacción podrá ver los cambios hechos por ella.

Cuando se da a la base de datos una instrucción ROLLBACK, la transacción termina y:

- Todo el trabajo hecho por la transacción se deshace, como si no se hubieran ejecutado las instrucciones correspondientes.
- Se liberan los bloqueos establecidos por la transacción.

La instrucción ROLLBACK tiene la siguiente sintaxis:

ROLLBACK [WORK].

Puntos de salvaguarda.

La instrucción ROLLBACK deshace toda la transacción. Con la instrucción SAVEPOINT, podemos conseguir que la instrucción rollback solo deshaga parte de la transacción.
La sintaxis de SAVEPOINT es:

SAVEPOINT nombre;

Donde nombre es el nombre del punto de salvaguarda. Una vez definido el punto de salvaguarda el programa puede deshacer la transacción hasta el punto de salvaguarda, mediante la sintaxis:

ROLLBACK TO SAVEPOINT nombre;

Cuando se ejecuta esta instrucción ocurre lo siguiente;

- Cualquier trabajo realizado desde el punto de salvaguarda se deshace. El punto de salvaguarda permanece activo, sin embargo, y se puede volver a él de nuevo, si se desea.
- Se libera cualquier bloqueo establecido y recursos adquiridos por las instrucciones SQL desde el punto de salvaguarda.
- La transacción no finaliza, dado que hay instrucciones SQL todavía pendientes:

Ejemplo:

```
BEGIN
      INSERT INTO temp_table(char_col)VALUES('UNO');
      SAVEPOINT A;
      INSERT INTO temp_table(char_col)VALUES('DOS');
      SAVEPOINT B;
      INSERT INTO temp_table(char_col)VALUES('TRES');
      SAVEPOINT C;
      -- OTRAS INSTRUCCIONES
      COMMIT;
END;
```

Si donde pone “otras instrucciones” escribimos:

ROLLBACK to B;

Se desharán el punto de salvaguarda C y la tercera instrucción INSERT. Pero las dos primeras serán procesadas.
Si sustituimos “otras instrucciones” por ROLLBACK TO A se desharán la segunda y la tercera instrucción INSERT dejando sólo la primera.

5. CURSORES

En el punto anterior hemos visto como pueden emplearse instrucciones SQL en PL/SQL. Con el uso de cursores podemos mejorar esta funcionalidad, permitiendo al programa tomar explícitamente el control del procesamiento de las instrucciones SQL.
En este apartado, veremos como emplear los cursores para efectuar consultas sobre múltiples filas, y para otras instrucciones SQL.
Abordaremos las variables de cursor, las cuales permiten un su más dinámico de los cursores.

¿Qué es un cursor?

Para poder procesar una instrucción SQL, Oracle asigna un área de memoria que recibe el nombre de **área de contexto**. Esta área contiene informaciones necesarias para completar el procesamiento, incluyendo el número de filas procesadas por la instrucción, un puntero a la versión analizada de la instrucción y, en el caso de las consultas, **el conjunto activo**, que es el conjunto de filas resultado de la consulta. Un cursor es un puntero al área de contexto. Mediante el cursor, un programa PL/SQL puede controlar el área de contexto y lo que en ella sucede a medida que se procesa una instrucción.

```
DECLARE
 -- Variables de salida para almacenar los resultados de la consulta
v_StudentID    students.id%TYPE;
  v_FirstName    students.first_name%TYPE;
  v_LastName     students.last_name%TYPE;

-- Variable de acoplamiento utilizada en la consulta

v_Major        students.major%TYPE := 'Computer Science';

  /* Cursor declaration */
  CURSOR c_Students IS
    SELECT id, first_name, last_name
      FROM students
      WHERE major = v_Major;
BEGIN
 -- Identificar las filas del conjunto activo y preparar el
 -- procesamiento posterior de datos
OPEN c_Students;
LOOP
   -- Recuperar cada fila del conjunto activo y almacenarlas en
   -- variables PL/SQL
      FETCH c_Students INTO v_StudentID, v_FirstName, v_LastName;

    -- Si no hay más filas salir de bucle
    EXIT WHEN c_Students%NOTFOUND;
 END LOOP;

  -- Liberar los recursos usados por la consulta
  CLOSE c_Students;
END;
```

Este ejemplo muestra el concepto de cursor explícito, donde se asigna explícitamente el nombre del cursor a una instrucción SELECT.
El procesamiento de un cursor explícito consta de cuatro pasos que describimos a continuación:

Procesamiento de cursores explícitos:

Los cuatro pasos necesarios para el procesamiento son:

- Declaración del cursor.
- Apertura del cursor para una consulta.
- Extracción de los resultados en variables PL/SQL.
- Cierre del cursor.

Declaración del cursor.

Define su nombre y asocia el cursor con una instrucción SELECT. La sintaxis es la siguiente:

```
CURSOR nombre_cursor IS instrucción_select;
```

Donde **nombre_cursor** es el nombre del cursor e **instrucción_select** es la consulta que deberá ser procesada por este cursor.
Una declaración de cursor puede hacer referencia a variables PL/SQL también.
Estas variables se consideran **variables de acoplamiento**. Como se aplican las reglas de ámbito usuales, estas variables deben ser visibles en el punto donde se declara el cursor, **esto implica tener declaradas las variables de acoplamiento antes que la declaración del cursor**, si en la declaración del cursor aparecen variables de acoplamiento.

Ejemplo:

```
DECLARE
  V_Departmento classes.department%TYPE;
  V_Curso classes.course%TYPE;

CURSOR c_Classes IS
SELECT * from classes
WHERE department=v_departamento
AND course=v_curso;
```

Esta declaración es correcta ya que v_departamento y v_curso son visibles en el momento de declaración del cursor.
Sin embargo, esta otra sección declarativa sería ilegal

```
DECLARE

CURSOR c_Classes IS
SELECT * from classes
WHERE department=v_departamento
AND course=v_curso;
V_Departmento classes.department%TYPE;
V_Curso classes.course%TYPE;
```

Apertura del cursor

La sintaxis es **OPEN** nombre_cursor

Cuando se abre un cursor, suceden tres cosas:

- Se examinan los valores de las variables de acoplamiento
- Se determina el conjunto activo, basándose en los valores de dichas variables y el contenido de la tabla o tablas a las que se hace referencia en la consulta.
- Se hace apuntar el puntero del conjunto activo a la primera fila.

Las variables de acoplamiento se examinan en el momento de abrir el cursor, y sólo en ese momento. Si quisiéramos cambiar de conjunto activo por cambio de valores en las variables de acoplamiento, habría que cerrar el cursor y volver a abrirlo.

Extracción de datos desde un cursor.

Mediante la orden FETCH. Tiene dos formas posibles:

- `FETCH nombre_cursor INTO lista_variables;`
- `FETCH nombre_cursor INTO registro_PL/SQL;`

Donde:

- **nombre_cursor** identifica un cursor previamente declarado y abierto
- **lista_variables** es una lista de variables PL/SQL ya declaradas y separadas por comas.
- **registro_ PL/SQL** es un registro PL/SQL previamente declarado.

En ambos casos, la variable o variables de la cláusula INTO deben ser compatibles en cuanto a tipo con la lista de selección de la consulta.

Cierre de un cursor.

Cuando se ha terminado de recuperar el conjunto activo, debe cerrarse el cursor. Esta acción informa a PL/SQL que el programa ha terminado de utilizarlo y se pueden liberar los recursos asociados a él. La sintaxis es:

`CLOSE nombre_cursor.`

Atributos de los cursores.

Existen cuatro atributos en PL/SQL que pueden aplicarse a los cursores. Los atributos de los cursores se añaden, en un bloque PL/SQL, al nombre del cursor, de forma similar a %TYPE y %ROWTYPE. Los atributos devuelven un valor que puede emplearse como parte de una expresión.

%FOUND

Es un atributo booleano. Devuelve true si la última instrucción FETCH devolvió una fila, y FALSE en caso contrario. Si se trata de comprobar el valor de %FOUND mientras el cursor no está abierto, se devuelve el error ORA-1001 (Cursor no válido).

%NOTFOUND

Se comporta de manera opuesta a %FOUND; si la extracción anterior devuelve una fila entonces %NOTFOUND devuelve FALSE; devuelve TRUE si la extracción anterior no devuelve una fila.

%ISOPEN

Atributo booleano que se utiliza para determinar si el cursor asociado está o no abierto.

%ROWCOUNT

Atributo numérico que devuelve el número de filas extraídas por el cursor hasta el momento.

Cursores parametrizados.

Existe otra manera de emplear variables de acoplamiento en un cursor. Existe un tipo de cursor, el cursor parametrizado, que admite argumentos, de la misma forma que los procedimientos.
Observemos el cursor c_Classes visto anteriormente:

```
DECLARE
  V_Departmento classes.department%TYPE;
  V_Curso classes.course%TYPE;

CURSOR c_Classes IS
SELECT * FROM classes
WHERE department=v_departamento
AND course=v_curso;
```

Contiene dos variables de acomplamiento v_departamento y v_curso. Podemos modificar c_Classes de la siguiente forma, para obtener un cursor parametrizado equivalente

```
DECLARE
  CURSOR c_Classes(p_Department classes.department%TYPE,
                   p_Course classes.course%TYPE) IS
  SELECT *
    FROM classes
    WHERE department = p_Department
    AND course = p_Course;
BEGIN
  OPEN c_Classes('HIS', 101);
END;
```

En el caso de los cursores parametrizados, se utiliza la instrucción OPEN para pasar los valores reales al cursor, tal y como aparece en el código anterior en esta instrucción:

```
OPEN c_Classes('HIS', 101);
```

Procesamiento de cursores implícitos.

Los cursores explícitos sirven para procesar instrucciones SELECT que devuelven más de una fila. Sin embargo, todas las instrucciones SQL se ejecutan dentro de un área de contexto y tienen, por tanto, un cursor que apunta a dicha área. Este cursor se conoce con el nombre de **cursor SQL**.

A diferencia de los cursores explícitos, el programa no abre ni cierra el cursor SQL, sino que PL/SQL lo abre de modo implícito, procesa la instrucción SQL en él contenida, y cierra el cursor después.
El cursor implícito sirve para procesar las instrucciones INSERT, UPDATE, DELETE, y las instrucciones SELECT .. INTO de una sola fila. A un cursor implícito se le pueden aplicar los atributos de cursor vistos anteriormente.
Veámoslo a través de un ejemplo:

```
BEGIN
  UPDATE rooms
    SET number_seats = 100
    WHERE room_id = 99980;
    -- Si la anterior instrucción UPDATE no afecta a ninguna fila,
    -- insertamos una nueva fila.
 IF SQL%NOTFOUND THEN
    INSERT INTO rooms (room_id, number_seats)
      VALUES (99980, 100);
  END IF;
END;
```

Esto podría haberse escrito también así utilizando el atributo SQL%ROWCOUNT

```
BEGIN
  UPDATE rooms
    SET number_seats = 100
    WHERE room_id = 99980;
-- Si la anterior instrucción UPDATE no afecta a ninguna
-- fila, insertamos una nueva fila.
  IF SQL%ROWCOUNT = 0 THEN
    INSERT INTO rooms (room_id, number_seats)
      VALUES (99980, 100);
  END IF;
END;
```

Aunque se puede emplear SQL%NOTFOUND con las instrucciones SELECT..INTO, no resulta útil, porque la instrucción SELECT..INTO que no encuentra una fila provoca el error Oracle ORA-1403: Datos no encontrados. Este error hace que el código se bifurque a la sección de manejo de excepciones.
El siguiente ejemplo muestra está situación:

```
DECLARE
  -- Registro para almacenar la información de una habitación.
  v_RoomData   rooms%ROWTYPE;
BEGIN
  -- Obtener la información de la habitación con ID igual a -1.
  SELECT *
    INTO v_RoomData
    FROM rooms
    WHERE room_id = -1;
-- La siguiente sentencia nunca se ejecutará, ya que el control pasa
-- inmediatamente al manejador de exepciones.
IF SQL%NOTFOUND THEN
    DBMS_OUTPUT.PUT_LINE('SQL%NOTFOUND is true!');
  END IF;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE('NO_DATA_FOUND raised!');
END;
```

Bucles de extracción mediante cursor.

La operación más común que se realiza con un cursor consiste en extraer todas las filas del conjunto activo. Para ello se utiliza un bucle de extracción, que no es más que un bucle que procesa una a una las filas del conjunto activo.
A continuación, examinamos diferentes tipos de bucles de extracción mediante cursor y su uso.

Bucles simples

En este primer estilo de bucle se utiliza la sintaxis de bucle simple LOOP ... END LOOP para el procesamiento, controlándose el número de veces que se ejecuta el bucle mediante atributos explícitos de cursor.

Ejemplo:

```
DECLARE
  -- Declaración de variables para almacenar información acerca de los
  -- estudiantes que cursan la especialidad de Historia
  v_StudentID  students.id%TYPE;
  v_FirstName   students.first_name%TYPE;
  v_LastName    students.last_name%TYPE;
 -- Cursor para obtener la información sobre los estudiantes de
 -- Historia.
CURSOR c_HistoryStudents IS
    SELECT id, first_name, last_name
      FROM students
      WHERE major = 'History';
BEGIN
  -- Abrimos el cursor y se inicia el conjunto activo.
  OPEN c_HistoryStudents;
  LOOP
```

```
    -- Obtener la información del siguiente estudiante.
    FETCH c_HistoryStudents INTO v_StudentID, v_FirstName, v_LastName;
    -- Salir si no hay más filas que recuperar.
    EXIT WHEN c_HistoryStudents%NOTFOUND;

    -- Procesamiento de las filas recuperadas.
    -- En este caso matricula a cada estudiante en Historia 301,
    -- insertándolo en la tabla registered_students.
    -- Registra también el nombre y el apellido en temp_table
INSERT INTO registered_students (student_id, department, course)
      VALUES (v_StudentID, 'HIS', 301);

    INSERT INTO temp_table (num_col, char_col)
      VALUES (v_StudentID, v_FirstName || ' ' || v_LastName);

  END LOOP;

  -- Liberar los recursos usados por el cursor.
  CLOSE c_HistoryStudents;
END;
```

Bucles WHILE.

Podemos construir un bucle de extracción mediante cursor utilizando la sintaxis WHILE..LOOP.

Ejemplo:

```
DECLARE
   -- Cursor para recuperar información acerca de los estudiantes de
   -- Historia
  CURSOR c_HistoryStudents IS
    SELECT id, first_name, last_name
      FROM students
      WHERE major = 'History';
  -- Declaramos un registro para almacenar la información extraída.
  v_StudentData  c_HistoryStudents%ROWTYPE;
BEGIN
-- Abrimos el cursor y se inicia el conjunto activo.
  OPEN c_HistoryStudents;
 -- Recuperar la primera fila, para configurar el bucle WHILE.
FETCH c_HistoryStudents INTO v_StudentData;

  -- El bucle continua mientras haya filas por recuperar.
  WHILE c_HistoryStudents%FOUND LOOP
    -- Procesamiento de las filas recuperadas.
    -- En este caso matricula a cada estudiante en Historia 301,
    -- insertándolo en la tabla registered_students.
    -- Registra también el nombre y el apellido en temp_table
    INSERT INTO registered_students (student_id, department, course)
      VALUES (v_StudentData.ID, 'HIS', 301);

    INSERT INTO temp_table (num_col, char_col)
```

```
      VALUES (v_StudentData.ID,
              v_StudentData.first_name || ' ' ||
v_StudentData.last_name);
-- Obtener la siguiente fila. La condición %FOUND se comprobará antes
-- de que el bucle continúe.

FETCH c_HistoryStudents INTO v_StudentData;
  END LOOP;

-- Liberar los recursos usados por el cursor.
  CLOSE c_HistoryStudents;
END;
/
```

Bucles FOR de cursor.

Los dos tipos de bucle de extracción descritos requieren un procesamiento explícito del cursor, mediante las instrucciones OPEN, FETCH, y CLOSE.
PL/SQL proporciona un tipo de bucle más simple, que realiza de modo implícito el procesamiento del cursor. Este bucle recibe el nombre de bucle FOR de cursor.
El siguiente ejemplo es equivalente a los anteriores:

```
DECLARE

CURSOR c_HistoryStudents IS
    SELECT id, first_name, last_name
      FROM students
      WHERE major = 'History';

BEGIN
  -- Comienza el bucle. Se ejecuta una instrucción OPEN implícita.
  FOR v_StudentData IN c_HistoryStudents LOOP

-- También podríamos haberlo escrito sin necesidad declarar cursor
-- alguno
/*
FOR v_StudentData IN (SELECT id, first_name, last_name
                        FROM students
                        WHERE major = 'History') LOOP */
-- Se ejecuta una instrucción FETCH implícita y se comprueba
-- %NOTFOUND

--  Procesamiento de las filas recuperadas.
--  En este caso matricula a cada estudiante en Historia 301,
--  insertándolo en la tabla registered_students.
--  Registra también el nombre y el apellido en temp_table.

    INSERT INTO registered_students (student_id, department, course)
    VALUES (v_StudentData.ID, 'HIS', 301);

    INSERT INTO temp_table (num_col, char_col)
      VALUES (v_StudentData.ID,
```

```
                v_StudentData.first_name || ' ' ||
v_StudentData.last_name);
  END LOOP;
-- Cuando el bucle acaba, se ejecuta una instrucción CLOSE implícita
END;
```

Cursores SELECT FOR UPDATE

A menudo, el procesamiento que se lleva a cabo en un bucle de extracción modifica las filas extraídas por el cursor.
PL/SQL proporciona una sintaxis conveniente para estas situaciones. El método consta de dos partes: la cláusula **FOR UPDATE** en la declaración del cursor, y la cláusula **WHERE CURRENT OF** en una instrucción UPDATE o DELETE.

For update.

Está cláusula es la última cláusula de una instrucción SELECT.
La sintaxis es:

```
SELECT.. FROM .. FOR UPDATE [OF referencia_columna][NOWAIT]
```

Donde referencia_columna es una columna, o una lista de columnas, de la tabla sobre la que se realiza la consulta.
Dos ejemplos distintos:

```
DECLARE
CURSOR c_AllStudents IS
SELECT *
FROM students
FOR UPDATE OF first_name, last_name;

CURSOR c_LargeClasses IS
SELECT department, course
FROM classes
WHERE max_students >50
FOR UPDATE;
```

Normalmente, una operación SELECT no efectúa ningún bloqueo sobre las filas a las que está accediendo, lo que permite que otras sesiones que estén conectadas a la base de datos cambien los datos seleccionados. A pesar de ello, el resultado será consistente. **Oracle toma una instantánea de la tabla en el momento de ejecutar la instrucción OPEN**.
Cualquier cambio que haya sido confirmado antes de este punto se verá reflejado en el conjunto activo, mientras que cualquier cambio posterior, incluso si es confirmado, no se verá reflejado, a menos que se reabra el cursor.
Sin embargo, si está presente la cláusula FOR UPDATE, **se ejecutan bloqueos exclusivos sobre las filas del conjunto activo antes de que termine la ejecución de la instrucción OPEN**.
Estos bloqueos evitan que otras sesiones cambien las filas del conjunto activo antes de que la transacción se confirme o se anule.
Si otra sesión ha establecido con anterioridad bloqueos sobre las filas del conjunto activo, entonces la operación SELECT..FOR UPDATE esperará a que dichos bloqueos sean levantados

por la otra sesión. No existe un tiempo máximo de espera, sino que la instrucción SELECT..FOR UPDATE quedará congelada hasta que la otra sesión levante el bloqueo.

Para tratar este tipo de situaciones existe la cláusula **NOWAIT**, que hace que, si las filas están bloqueadas por otra sesión, la instrucción OPEN termine inmediatamente devolviendo el error: ORA-54: Recurso ocupado y adquirido con NOWAIT especificado.

A partir de Oracle9i, se puede utilizar la sintaxis:

```
SELECT..FROM FOR UPDATE [OF columna][WAIT n]
```

donde n es el número de segundos de espera. Si las filas no son desbloqueadas en n segundos, se devolverá el error ORA –54.
Cuando la consulta afecta a varias tablas, podemos usar la cláusua **FOR UPDATE OF** para restringir el bloqueo de filas a alguna tabla en particular. Se bloquearán las filas de una tabla en particular si la cláusula FOR UPDATE OF hace referencia a una columna de dicha tabla.
Por ejemplo, la siguiente consulta bloquea filas en la tabla emp, pero no en la tabla dept (la tabla emp es la que tiene la columna sal en su definición):

```
DECLARE
   CURSOR c1 IS
   SELECT ename, dname
   FROM emp, dept
   WHERE emp.deptno = dept.deptno AND job = 'MANAGER'
   FOR UPDATE OF sal;
```

WHERE CURRENT OF

Si se declara el cursor con la cláusula FOR UPDATE, puede emplearse la cláusula WHERE CURRENT OF en una instrucción UPDATE o DELETE.
La sintaxis de esta cláusula es:

```
WHERE CURRENT OF cursor
```

Donde cursor es el nombre de un cursor declarado con una cláusula FOR UPDATE.
La cláusula WHERE CURRENT OF hace referencia a la fila recién extraída por el cursor.
El siguiente ejemplo muestra un bloque que actualiza los créditos actuales de todos los alumnos matriculados en el curso HIS 101:

```
DECLARE
  -- Número de créditos a añadir al total de cada estudiante
  -- Este cursor solo selecciona a aquellos estudiantes que
  -- actualmente están matriculados en HIS 101
  CURSOR c_RegisteredStudents IS
    SELECT *
      FROM students
      WHERE id IN (SELECT student_id
                     FROM registered_students
                     WHERE department= 'HIS'
                     AND course = 101)
      FOR UPDATE OF current_credits;

BEGIN
  --  Preparamos el bucle de extracción del cursor.
  FOR v_StudentInfo IN c_RegisteredStudents LOOP
```

```
  -- Determinar del número de créditos para HIS 101 .
  SELECT num_credits
    INTO v_NumCredits
    FROM classes
    WHERE department = 'HIS'
    AND course = 101;

  -- Actualizamos la fila que acabamos de recuperar con el cursor.
  UPDATE students
    SET current_credits = current_credits + v_NumCredits
    WHERE CURRENT OF c_RegisteredStudents;
  END LOOP;

  -- Confirmamos el trabajo y liberamos los bloqueos.
  COMMIT;
END;
```

Extracciones entre instrucciones COMMIT.

Observemos que la instrucción COMMIT del ejemplo anterior se ejecuta después de completar el bucle de extracción. Se hace así porque una instrucción COMMIT libera los bloqueos establecidos por la sesión, y en particular los establecidos como consecuencia de la cláusula FOR UPDATE. Cuando esto sucede, el cursor queda invalidado, y cualquier extracción posterior devolverá el error Oracle **ORA-1002: Extracción fuera de secuencia.**

Variables de cursor

Todos los cursores explícitos vistos hasta ahora eran ejemplos de **cursores estáticos**: el cursor está asociado con una instrucción SQL, y ésta es conocida en el momento de compilar el bloque.
Una variable de cursor, puede asociarse a diferentes consultas en tiempo de ejecución.
Para poder utilizar una variable de cursor, primero es necesario declararla.

Declaración de una variable de cursor.

Una variable de cursor es un tipo de referencia. Un tipo de referencia es lo mismo que un puntero de C, que puede apuntar a diferentes posiciones de almacenamiento. Los tipos de referencia en PL/SQL se declaran mediante la sintaxis:

```
REF tipo
```

Donde tipo es un tipo previamente definido. La sintaxis completa para definir una variable de cursor es:

```
TYPE nombre_tipo IS REF CURSOR [RETURN tipo_retorno];
```

Donde nombre_tipo es el nuevo tipo de referencia y tipo_retorno es un tipo de registro, que indica los tipos de la lista de selección que será finalmente devuelta por la variable de cursor, en el caso de que la cláusula RETURN exista.

Ejemplos de declaración:

```
DECLARE
  -- Definición usando  %ROWTYPE.
  TYPE t_StudentsRef IS REF CURSOR
    RETURN students%ROWTYPE;

  -- Definimos un nuevo tipo registro,
  TYPE t_NameRecord IS RECORD (
    first_name  students.first_name%TYPE,
    last_name   students.last_name%TYPE);

  -- una variable de este tipo,
  v_NameRecord  t_NameRecord;

  -- Y una variable de cursor usando el tipo registro.
  TYPE t_NamesRef IS REF CURSOR
    RETURN t_NameRecord;
```

Las variables de cursor vistas en el ejemplo anterior están restringidas, en el sentido que han sido declaradas para un único tipo de retorno. PL/SQL permite declarar variables de cursor no restringidas, caracterizadas por no tener cláusula RETURN.

Ejemplo:

```
  -- Define un tipo de referencia no restringido.
  TYPE t_FlexibleRef IS REF CURSOR;

  -- y una variable de este tipo.
  v_CursorVar t_FlexibleRef;

BEGIN
  NULL;
END;
```

Asignación de espacio de almacenamiento para las variables de cursor.

Puesto que una variable de cursor es un puntero, no se produce asignación de almacenamiento cuando se declara. Antes de usarla hay que hacer que apunte a un área de memoria válida.

Reserva de espacio en el lado del cliente:

Dependerá del lenguaje en el que se haya escrito el programa del lado del cliente. Para cada caso estudiaremos la oportuna documentación. Nosotros, en el caso que utilicemos variables de cursor las utilizaremos en el lado del servidor.

Reserva de espacio en el lado del servidor

El espacio de memoria se reserva automáticamente cuando es necesario. Cuando la variable cae fuera de ámbito y deja de referenciar dicho espacio de almacenamiento, éste se libera.

Apertura de una variable de cursor para una consulta.

Para poder asociar una variable de cursor con una instrucción SELECT en particular, se amplía la sintaxis de la instrucción OPEN de la siguiente manera:

```
OPEN variable_cursor FOR instrucción_selección.
```

Cierre de las variables de cursor.

En el servidor, se cierran de la misma manera que los cursores estáticos con la instrucción CLOSE seguida del nombre de la variable de cursor.

Restricciones en el uso de variables de cursor:

- No se pueden definir en un paquete.
- Las colecciones no pueden almacenar variables de cursor.
- La consulta asociada con una variable de cursor en la instrucción OPEN..FOR no puede ser de actualización (no puede ser FOR UPDATE).

6. Tratamiento de errores

Excepción

Las excepciones y los manejadores de excepciones son el método a través del cual el programa reacciona a los errores de ejecución y realiza su tratamiento. Cuando se produce un error se genera una excepción y el control pasa al gestor de excepciones que es una sección independiente del programa.

Declaración de excepciones.

Las excepciones se declaran en la sección declarativa de un bloque, se generan en la sección ejecutable y se tratan en la sección de manejo de excepciones.
Existen dos tipos de excepciones, **definidas por el usuario** y **predefinidas por oracle**.

Excepciones definidas por el usuario

Es un error cuya definición se realiza en el programa. La sintaxis es la siguiente:

```
DECLARE
....
      nombre_excepción EXCEPTION;
...
```

Excepciones predefinidas.

Oracle ha definido diversas excepciones que se corresponden con los errores Oracle más comunes. A continuación, se muestra la lista en la que aparece el nombre de la excepción y el número de error oracle asociado.

Excepción	Error Oracle	SQLCODE Valor
ACCESS_INTO_NULL	ORA-06530	-6530
COLLECTION_IS_NULL	ORA-06531	-6531
CURSOR_ALREADY_OPEN	ORA-06511	-6511
DUP_VAL_ON_INDEX	ORA-00001	-1
INVALID_CURSOR	ORA-01001	-1001
INVALID_NUMBER	ORA-01722	-1722
LOGIN_DENIED	ORA-01017	-1017
NO_DATA_FOUND	ORA-01403	+100
NOT_LOGGED_ON	ORA-01012	-1012
PROGRAM_ERROR	ORA-06501	-6501
ROWTYPE_MISMATCH	ORA-06504	-6504
SELF_IS_NULL	ORA-30625	-30625
STORAGE_ERROR	ORA-06500	-6500
SUBSCRIPT_BEYOND_COUNT	ORA-06533	-6533
SUBSCRIPT_OUTSIDE_LIMIT	ORA-06532	-6532
SYS_INVALID_ROWID	ORA-01410	-1410
TIMEOUT_ON_RESOURCE	ORA-00051	-51
TOO_MANY_ROWS	ORA-01422	-1422
VALUE_ERROR	ORA-06502	-6502
ZERO_DIVIDE	ORA-01476	-1476

Breve descripción de algunas de las excepciones:

Excepcion	Generada cuando...
ACCESS_INTO_NULL	Se ha intentado asignar valores a atributos de un objeto que tiene valor NULL. (Recordad colecciones)
COLLECTION_IS_NULL	Intento de aplicar métodos de colección distintos de EXISTS a una tabla o varray con valor NULL.
CURSOR_ALREADY_OPEN	Intento de abrir un cursor ya abierto.
DUP_VAL_ON_INDEX	Violación de una restricción de unicidad.
INVALID_CURSOR	Operación ilegal con un cursor, tal como tratar de cerrar un cursor ya cerrado.
INVALID_NUMBER	En una instrucción SQL cuando falla la conversión de una cadena de caracteres a número. En una instrucción procedimental se generaría en su lugar la excepción VALUE_ERROR.
LOGIN_DENIED	Nombre de usuario o contraseña inválidos.
NO_DATA_FOUND	Una sentencia SELECT INTO que no devuelve filas, o tratar de acceder a un elemento eliminado en una tabla anidada a acceso a un elemento no inicializado de una tabla indexada
NOT_LOGGED_ON	No hay conexión con Oracle.
PROGRAM_ERROR	PL/SQL tiene un problema interno.
ROWTYPE_MISMATCH	Incompatibilidad de tipos en las variables afectadas.
SELF_IS_NULL	Intento de llamar a un método en una instancia de objeto que tiene el valor NULL.
STORAGE_ERROR	Error interno PL/SQL provocado por falta de memoria.
SUBSCRIPT_BEYOND_COUNT	Intento de acceder a una tabla anidada o índice de varray mayor que el número de elementos de la colección.
SUBSCRIPT_OUTSIDE_LIMiT	Intento de acceder a una tabla anidada o índice de varray fuera del rango declarado.
SYS_INVALID_ROWID	La conversión de un string a un rowid universal falla porque el string no representa un rowid válido.
TIMEOUT_ON_RESOURCE	Tiempo excedido en la espera de un recurso
TOO_MANY_ROWS	Una sentencia SELECT INTO devuelve más de una fila.
VALUE_ERROR	Provocado por un error aritmético, de conversión, de truncamiento o de restricciones.
ZERO_DIVIDE	Intento de dividir por cero

Generación de excepciones.

Cuando se produce el error asociado con una excepción, dicha excepción es generada.
Las excepciones definidas por el usuario se generan explícitamente mediante la instrucción RAISE de la siguiente manera:

```
RAISE nombre_excepción;
```

Las excepciones predefinidas (o las excepciones definidas por el usuario asociadas a un error Oracle mediante el pragma EXCEPTION_INIT) se generan implícitamente cuando ocurre su error. Si se produce un error Oracle que no está asociado a una excepción, también se genera una excepción, que se puede detectar en la rutina de tratamiento OTHERS.

Las excepciones predefinidas pueden generarse también de manera explícita si así se desea, mediante la instrucción RAISE.

Tratamiento de excepciones.

Cuando se genera una excepción el control pasa a la sección de excepciones del bloque.
La sección de excepciones está compuesta por rutinas de tratamiento para las distintas excepciones llamadas también **gestor de excepciones**.
Una rutina de tratamiento contiene el código que se ejecutará cuando ocurra el error asociado con la excepción y ésta sea generada. Si la rutina de tratamiento para la excepción no existe, la excepción se propaga al bloque del nivel superior. Una vez que pasa el control al gestor de excepciones no hay forma de volver a la sección ejecutable del bloque.

La sección de excepciones está compuesta por rutinas de tratamiento para las distintas excepciones. Cada rutina está compuesta por la cláusula WHEN y las instrucciones que se ejecutarán cuando la excepción sea generada. La cláusula WHEN identifica la excepción correspondiente a cada rutina de tratamiento.
La sintaxis de la sección de excepciones es como sigue:

```
EXCEPTION
      WHEN nombre_excepción THEN
            Secuencia_de_órdenes1;
      WHEN nombre_excepción THEN
            Secuencia_de_órdenes2;
      WHEN OTHERS THEN
            Secuencia_de_órdenes3;
END;
```

Un mismo gestor puede utilizarse para más de una excepción

Ejemplo:

```
EXCEPCION
      WHEN NOT_DATA_FOUND OR TOO_MANY_ROWS THEN
```

La rutina de tratamiento de excepciones OTHERS.

El gestor OTHERS se ejecutará para todas las excepciones generadas y no controladas con otros gestores. Debe ser siempre el último gestor del bloque. Es una buena práctica de programación el definir un gestor OTHERS en el nivel superior del programa, para asegurarse de que ningún error se quede sin detectar. Para determinar qué error generó la excepción que esté siendo tratada por el gestor OTHERS, podemos usar las funciones predefinidas SQLCODE y SQLERRM, descritas a continuación.

SQLCODE y SQLERRM

Utilizadas para averiguar dentro de una rutina de tratamiento OTHERS, qué error Oracle dio lugar a la excepción, independientemente de que haya o no una excepción predefinida para el error.

- SQLCODE: devuelve el **código de error actual**
- SQLERRM: el **mensaje de texto del error**.

En el caso de excepciones definidas por el usuario, SQLCODE devuelve 1 y SQLERRM devuelve User_defined Exception.
También se puede llamar a SQLERRM con un argumento numérico. En este caso, la función devuelve el texto asociado con el número de error. El argumento debe ser siempre negativo salvo 0 o +100. Con 0 devuelve ORA-0000: normal, successful completion y con +100 No data found.
El siguiente ejemplo resume lo visto hasta ahora:

```
DECLARE
-- Excepción que indica una condición de error.
 e_TooManyStudents EXCEPTION;
  -- Número actual de estudiantes matriculados en HIS-101
  v_CurrentStudents NUMBER(3);
  -- Número máximo de estudiantes permitidos en HIS-101
  v_MaxStudents NUMBER(3);

  -- Código y texto de otros errores de tiempo de ejecución.
  v_ErrorCode log_table.code%TYPE;
  v_ErrorText log_table.message%TYPE;
BEGIN
  /* Hallar el número de estudiantes registrados y el número máximo de
alumnos permitidos. */
  SELECT current_students, max_students
    INTO v_CurrentStudents, v_MaxStudents
    FROM classes
    WHERE department = 'HIS' AND course = 101;

  /* Comprobar el número de alumnos en ese curso. */
  IF v_CurrentStudents > v_MaxStudents THEN
    /* Demasiados alumnos matriculados  -- generar excepción. */
    RAISE e_TooManyStudents;
  END IF;
EXCEPTION
  WHEN e_TooManyStudents THEN
      /* Rutina de tratamiento que se ejecuta cuando hay demasiados
      alumnos matriculados en HIS-101.
      Insertará un mensaje que registre lo sucedido. */
    INSERT INTO log_table (info)
      VALUES ('History 101 has ' || v_CurrentStudents ||
              'students: max allowed is ' || v_MaxStudents);
  WHEN OTHERS THEN
    -- Rutina de tratamiento que se ejecuta para el resto de errores.
    v_ErrorCode := SQLCODE;
    -- Observar el uso de SUBSTR.
    v_ErrorText := SUBSTR(SQLERRM, 1, 200);
    INSERT INTO log_table (code, message, info) VALUES
      (v_ErrorCode, v_ErrorText, 'Oracle error occurred');
END;
```

El pragma EXCEPTION_INIT.

Es posible asociar una excepción nominada con un error Oracle determinado, lo que nos permite interceptar de forma específica dicho error, en lugar de hacerlo a través de una rutina de tratamiento OTHERS.
Para ello se utiliza el pragma **EXEPTION_INIT**. Se emplea de la siguiente manera:

```
PRAGMA EXCEPTION_INIT(nombre_excepcion, número_error_Oracle)
```

```
DECLARE
  e_NuloNoPermitido EXCEPTION;

  PRAGMA EXCEPTION_INIT(e_NuloNoPermitido, -1400);
BEGIN
  INSERT INTO students (id) VALUES (NULL);
EXCEPTION
  WHEN  e_NuloNoPermitido then
    INSERT INTO log_table (info) VALUES ('ORA-1400 occurred');
END;
```

Utilización de RAISE_APPLICATION_ERROR.

Puede emplear la función predefinida RAISE_APLICATION_ERROR para crear sus propios mensajes de error, que pueden ser más descriptivos que las excepciones nominadas. Los errores definidos por el usuario se propagan fuera del bloque, al entorno que realizó la llamada, de la misma forma que los errores Oracle. La sintaxis de RAISE_APLICATION_ERROR es la siguiente:

```
RAISE_APLICATION_ERROR (número_error,mensaje_error,[preservar_errores]);
```

Donde:

- **número_error** es un parámetro comprendido entre –20000 y –20999.
- **mensaje_error** es el texto es el texto asociado con este error . Debe ser menor de 512 caracteres
- **preservar_errores** es una valor booleano. Si dicho parámetro toma el valor TRUE, el nuevo error se añade a la lista de errores que han sido generados (si es que existe); si toma el valor FALSE, cosa que hace por defecto, el nuevo error reemplazará la lista actual de errores.

Ejemplo de uso:

```
CREATE OR REPLACE FUNCTION FACTORIAL(N NUMBER)RETURN NUMBER AS
BEGIN
      IF N<0 THEN
            RAISE_APPLICATION_ERROR(-20000,'NO EXISTE EL FACTORIAL DE
                                          UN NUMERO NEGATIVO');
```

```
        END IF;
        IF N!=TRUNC(N) THEN
              RAISE_APPLICATION_ERROR(-20000,'NO EXISTE EL FACTORIAL DE
                                              UN NUMERO DECIMAL');
        END IF;
        IF N=0 THEN
              RETURN 1;
        ELSE
              RETURN N*FACTORIAL(N-1);
        END IF;
END;
```

Propagación de excepciones.

Excepciones generadas en la sección ejecutable:

Cuando se genera una excepción en la sección ejecutable de un bloque PL/SQL, se utiliza el siguiente algoritmo para determinar qué gestor de excepciones invocar:

1) Si el bloque actual tiene un gestor para la excepción, lo ejecuta y termina el bloque. El control pasa entonces al bloque de nivel superior.

2) Si no hay gestor para la excepción actual, se propaga la excepción, generándola en el bloque de nivel superior.

   Entonces se ejecuta el paso 1) en el bloque de nivel superior. Si no hay un bloque de nivel superior, la excepción se propagará al entorno que realiza la llamada, como podría ser SQL*Plus.

Excepciones generadas en la sección declarativa

Si una asignación en la sección declarativa, genera una excepción, la excepción se propaga inmediatamente al bloque de nivel superior.

Excepciones generadas en la sección de excepciones

También pueden generarse excepciones dentro de un gestor de excepciones, bien explícitamente mediante raise, o bien implícitamente por un error de ejecución. En cualquiera de los casos, la excepción se propaga inmediatamente al bloque de nivel superior.

Consejos sobre las excepciones.

- **Ámbito de las excepciones:** El ámbito de las excepciones es igual que el de las variables. Si una excepción definida por el usuario se propaga fuera de su ámbito, no podrá ser referenciada.

En general, si hay que propagar un error definido por el usuario fuera de un bloque, es mejor definir la excepción en un paquete, para continúe siendo visible fuera del bloque, o utilizar en su lugar RAISE_APLICATION_ERROR.

- Utilización de WHEN OTHERS para evitar las excepciones no tratadas.

Determinación de la localización del error.

Puesto que se utiliza una misma sección de excepciones para todo el bloque, puede resultar difícil determinar que instrucción SQL provocó un determinado error. Consideremos el siguiente ejemplo:

```
BEGIN
      SELECT ....
      SELECT ....
      SELECT ....
EXCEPTION
WHEN NO_DATA_FOUND THEN

¿Qué select generó la excepción?
```

Hay dos formas de resolver este problema: El primero consiste en incrementar un contador que indique la instrucción SQL.

```
DECLARE
-- Variable para almacenar el número de instrucción SELECT.
   V_ContadorSelect NUMBER:=1;
   SELECT ....
   V_ContadorSelect:=2;
   SELECT...
   V_ContadorSelect:=3;
   SELECT....
EXCEPTION
      WHEN NO_DATA_FOUND THEN
            IF V_Contador_Select=1 then ....
            ELSIF V_Contador_Select=2  THEN....
            ELSE ...
 END;
```

El segundo método consiste en incluir cada instrucción dentro de su propio subloque:

```
BEGIN
      BEGIN
            SELECT ....
      EXCEPTION
            WHEN NO_DATA_FOUND THEN
                  ....
      END;
      BEGIN
            SELECT ....
      EXCEPTION
            WHEN NO_DATA_FOUND THEN
                  ....
      END;
END;
```

Este formato nos permite ejecutar las tres SELECT aunque alguna de ellas falle.

7. Procedimientos, funciones y paquetes.

Procedimientos y funciones.

Se comportan de manera muy parecida a los procedimientos y funciones de otros lenguajes de tercera generación.
A modo de ejemplo, el siguiente código crea un procedimiento en la base de datos:

```
CREATE OR REPLACE PROCEDURE AddNewStudent (
  p_FirstName  students.first_name%TYPE,
  p_LastName   students.last_name%TYPE,
  p_Major      students.major%TYPE) AS
BEGIN
-- Inserta una nueva fila en la tabla students. Se usa
student_sequence
-- para generar un nuevo identificador, se introduce 0 en el campo
current_credits.
  INSERT INTO students (ID, first_name, last_name,
                        major, current_credits)
    VALUES (student_sequence.nextval, p_FirstName, p_LastName,
            p_Major, 0);
END AddNewStudent;
```

Una vez creado el procedimiento, puede realizarse una llamada al mismo desde otro bloque PL/SQL:

```
BEGIN
  AddNewStudent('Zelda', 'Zudnik', 'Computer Science');
END;
```

Este ejemplo muestra varios detalles a tener en cuenta:

El procedimiento AddNewStudent se crea mediante la instrucción CREATE OR REPLACE PROCEDURE. Cuando se crea un procedimiento, primero se compila y luego se almacena en la base de datos en formato compilado. Este código ya compilado puede ejecutarse después desde otro bloque PL/SQL.
Cuando se realiza una llamada al procedimiento, se pueden pasar parámetros.
Una llamada a un procedimiento es una instrucción PL/SQL en si misma. Cuando se realiza una llamada a un procedimiento, se transfiere el control a la primera instrucción ejecutable del procedimiento. Cuando el procedimiento finaliza su ejecución, se devuelve el control a la instrucción siguiente a la llamada al procedimiento.
Un procedimiento es un bloque PL/SQL, con una sección declarativa, una sección ejecutable y una sección de manejo de excepciones.

Creación de un procedimiento.

La sintaxis básica de la instrucción CREATE [OR REPLACE] PROCEDURE es:

```
CREATE [OR REPLACE] PROCEDURE nombre_procedimiento
[(argumento[{IN|OUT|IN OUT}] tipo,
....
....
argumento [{IN|OUT|IN OUT}] tipo)]{IS|AS}

cuerpo_del_procedimiento
```

donde:

- **nombre_procedimiento**: es el nombre del procedimiento que se quiere crear.
- **argumento**: es el nombre de un parámetro del procedimiento.
- **tipo** es el tipo del parámetro.
- **cuerpo_del_procedimiento** es un bloque PL/SQL que constituye el código del procedimiento.

Más adelante estudiaremos en profundidad el tema de los parámetros.

Para modificar el código de un procedimiento, éste debe eliminarse y volverse a crear.
Como ésta es la operación habitual en la fase de desarrollo, las palabras clave **OR REPLACE** permiten que esto se haga mediante una sola operación (sino habría que eliminarlo previamente con la orden DROP PROCEDURE nombre_procedimiento).

El cuerpo del procedimiento.

Es un bloque PL/SQL con secciones declarativa, ejecutable y tratamiento de excepciones. La sección declarativa aparece entre la palabra clave IS o AS y la palabra clave BEGIN.
La sección ejecutable (la única obligatoria) aparece entre las palabras clave BEGIN y EXCEPTION, o entre BEGIN y END si no existe la sección de tratamiento de excepciones.
La sección de tratamiento de excepciones se sitúa entre las palabras clave EXCEPTION y END.

La estructura por tanto de un procedimiento tiene este aspecto:

```
CREATE OR REPLACE PROCEDURE nombre_procedimiento[listaparametros] AS
/* Aquí vendría la sección declarativa */
BEGIN
 /* Sección ejecutable */
EXCEPTION
   /* Sección de manejo de excepciones */
END[nombre_procedimiento];
```

Creación de una función.

Una función es muy parecida a un procedimiento. Sin embargo, una llamada a un procedimiento es, en sí misma, una instrucción PL/SQL, mientras que una llamada a una función se produce como parte de una expresión; el valor asociado a la función en la expresión es el que devuelve a través se la instrucción RETURN.

Veamos a continuación un ejemplo de una función y un bloque de código usándola:

```
CREATE OR REPLACE FUNCTION AlmostFull (
  p_Department classes.department%TYPE,
  p_Course     classes.course%TYPE)
  RETURN BOOLEAN IS

  v_CurrentStudents NUMBER;
  v_MaxStudents     NUMBER;
  v_ReturnValue     BOOLEAN;
  v_FullPercent     CONSTANT NUMBER := 80;
BEGIN
  -- Obtiene el número actual y el máximo de alumnos para el curso
  -- solicitado
  SELECT current_students, max_students
    INTO v_CurrentStudents, v_MaxStudents
    FROM classes
    WHERE department = p_Department
    AND course = p_Course;

-- Si el curso tiene más alumnos que el porcentaje dado por
v_FullPercent
-- devuelve TRUE. En caso contrario devuelve FALSE.
  IF (v_CurrentStudents / v_MaxStudents * 100) >= v_FullPercent THEN
    v_ReturnValue := TRUE;
  ELSE
    v_ReturnValue := FALSE;
  END IF;

  RETURN v_ReturnValue;
END AlmostFull;
```

La función devuelve un valor booleano y puede llamarse, por ejemplo, desde el siguiente código PL/SQL. La llamada a la función no es una instrucción en sí misma, sino que se utiliza como parte de la instrucción IF.

```
DECLARE
  CURSOR c_Classes IS
    SELECT department, course
    FROM classes;
BEGIN
  FOR v_ClassRecord IN c_Classes LOOP
    -- Mostrar los cursos que no tienen mucho espacio libre.
    IF AlmostFull(v_ClassRecord.department,
                  v_ClassRecord.course) THEN
      DBMS_OUTPUT.PUT_LINE(
        v_ClassRecord.department || ' ' ||
        v_ClassRecord.course || ' está casi lleno!');
    END IF;
  END LOOP;
END;
```

Sintaxis de las funciones.

Es muy parecida a la de los procedimientos:

```
CREATE OR REPLACE FUNCTION nombre_funcion
[(argumento[{IN|OUT|IN OUT}] tipo,
....
....
argumento [{IN|OUT|IN OUT}] tipo)]
RETURN tipo_retorno {IS|AS}
cuerpo_de_la_ función
```

donde:

- **nombre_función**: es el nombre de la función que se quiere crear.
- **argumento**: es el nombre de un parámetro del procedimiento.
- **tipo** es el tipo del parámetro.
- **Tipo_retorno:** es el tipo del valor que devuelve la función
- **cuerpo_de_la_ función** : es un bloque PL/SQL que constituye el código de la función.

Dentro del cuerpo de la función, la instrucción RETURN, que es obligatoria, se utiliza para devolver el control, junto con un valor, al entorno que realizó la llamada.

La sintaxis general de la instrucción RETURN es

```
RETURN expresión
```

Donde expresión es el valor que se devuelve. Cuando se ejecuta RETURN, expresión se convierte al tipo especificado en la cláusula RETURN de la definición de a función, si no son del mismo tipo. Después, el control se devuelve inmediatamente al entorno que realizó la llamada.

Eliminación de procedimientos y funciones.

Para eliminar un procedimiento la sintaxis a emplear es:

```
DROP PROCEDURE nombre_procedimiento;
```

Y la sintaxis para eliminar una función es:

```
DROP FUNCTION nombre_función.
```

Parámetros de los subprogramas.

Examinaremos en las siguientes secciones el paso de parámetros en PL/SQL.

Modos de los parámetros.

Dado el procedimiento AddNewStudent visto anteriormente, podemos llamarlo desde el siguiente bloque anónimo:

```
DECLARE
-- Variables que describen al nuevo alumno
  v_NewFirstName  students.first_name%TYPE := 'Cynthia';
  v_NewLastName   students.last_name%TYPE := 'Camino';
  v_NewMajor      students.major%TYPE := 'History';
BEGIN
  -- Añadir a Cynthia Camino a la base de datos.
  AddNewStudent(v_NewFirstName, v_NewLastName, v_NewMajor);
END;
```

Las variables declaradas en el bloque anterior (v_NewFirstName ,v_NewLastName, v_NewMajor) se pasan como **argumentos** a AddNewStudent.
En este contexto, se denominan **parámetros reales**, mientras que los parámetros en la declaración del procedimiento (p_FirstName , p_LastName y p_Major) se denominan **parámetros formales**.
Los parámetros reales son los que contienen los valores que se pasan en la llamada al procedimiento y son los que reciben los resultados del procedimiento cuando éste retorna (dependiendo del modo).
Cuando se llama a un procedimiento, el valor de los parámetros reales se asigna a los parámetros formales. Cuando el procedimiento retorna (dependiendo del modo) el valor de los parámetros formales se asigna a los parámetros reales.
Los parámetros formales pueden tener tres modos: **IN, OUT, IN OUT**, (a partir de Oracle 8i se puede añadir el modificador **NOCOPY**).
Si no se especifica nada el modo de un parámetro formal es IN.
A continuación, se describen los tres modos:

- **IN:** El valor del parámetro real se pasa al procedimiento cuando se produce la llamada. Dentro del procedimiento, el parámetro formal se comporta como una constante PL/SQL (se considera de solo lectura y no se puede cambiar, generando el compilador el correspondiente mensaje de error). Cuando el procedimiento finaliza y devuelve el control al entorno que realizó la llamada, el parámetro real no se modifica.

- **OUT:** Se ignora cualquier valor que el parámetro real pueda tener cuando se produce la llamada al procedimiento. Dentro del procedimiento, el parámetro formal se comporta como una variable sin inicializar de PL/SQL por lo que tiene el valor NULL. Puede leerse (versiones posteriores a 8.0.3) dicha variable y escribirse en ella. Cuando el procedimiento finaliza y devuelve el control al entorno de llamada, se asigna el valor del parámetro formal al parámetro real (a partir de Oracle 8i este comportamiento se puede alterar mediante el modificador NOCOPY)

- **IN OUT:** Este modo es una combinación de los modos IN y OUT. El valor del parámetro real se pasa al procedimiento en la llamada. Dentro del procedimiento, el parámetro formal se comporta como una variable inicializada, pudiendo leerse su valor y escribirse en ella. Cuando el procedimiento finaliza y devuelve el control al entorno de llamada, se asigna el valor del parámetro formal al parámetro real (a partir de Oracle 8i este comportamiento se puede alterar mediante el modificador NOCOPY).

Si el procedimiento genera una excepción, los valores de los parámetros formales con modos IN OUT y OUT no se copian a los parámetros reales correspondientes (esto también sujeto al modificador NOCOPY que veremos posteriormente).

El siguiente ejemplo muestra el uso de lo visto en esta sección:

```
CREATE OR REPLACE PROCEDURE ModeTest (
  p_InParameter    IN NUMBER,
  p_OutParameter   OUT NUMBER,
  p_InOutParameter IN OUT NUMBER) IS

  v_LocalVariable  NUMBER := 0;

BEGIN
  DBMS_OUTPUT.PUT_LINE('Dentro de ModeTest:');
  IF (p_InParameter IS NULL) THEN
    DBMS_OUTPUT.PUT('p_InParameter es NULO');
  ELSE
    DBMS_OUTPUT.PUT('p_InParameter = ' || p_InParameter);
  END IF;

  IF (p_OutParameter IS NULL) THEN
    DBMS_OUTPUT.PUT('  p_OutParameter is NULL');
  ELSE
    DBMS_OUTPUT.PUT('  p_OutParameter = ' || p_OutParameter);
  END IF;

  IF (p_InOutParameter IS NULL) THEN
    DBMS_OUTPUT.PUT_LINE('  p_InOutParameter is NULL');
  ELSE
    DBMS_OUTPUT.PUT_LINE('  p_InOutParameter = ' ||
                         p_InOutParameter);
  END IF;

  /* Asignación de p_InParameter a v_LocalVariable. Esto es legal,
     ya que se lee un parámetro IN parameter y no escribimos en él */
 v_LocalVariable := p_InParameter;  -- Legal

  /* Asignamos 7 a p_InParameter. Esto es  ILEGAL, ya que escribimos
    en un parámetro IN. */
-- p_InParameter := 7;  -- Ilegal

  /* Asignamos 7 a p_OutParameter. Esto es legal, ya que escribimos en
     un  parámetro OUT. */
  p_OutParameter := 7;  -- Legal

  /* Asignamos p_OutParameter a v_LocalVariable. En versiones de
Oracle anteriores a  7.3.4, es ilegal leer de un parametro de tipo OUT
*/
  v_LocalVariable := p_OutParameter;  -- Possibly illegal

  /* Asignamos  p_InOutParameter a v_LocalVariable. Es legal,
     ya que leemos un parámetro de tipo IN OUT. */
  v_LocalVariable := p_InOutParameter;  -- Legal

  /* Asignamos 8 a p_InOutParameter. Legal puesto que escribimos en un
      parámetro IN OUT. */
  p_InOutParameter := 8;  -- Legal
```

```
  DBMS_OUTPUT.PUT_LINE('Al final de ModeTest:');
  IF (p_InParameter IS NULL) THEN
    DBMS_OUTPUT.PUT('p_InParameter es nulo);
  ELSE
    DBMS_OUTPUT.PUT('p_InParameter = ' || p_InParameter);
  END IF;

  IF (p_OutParameter IS NULL) THEN
    DBMS_OUTPUT.PUT('  p_OutParameter es nulo');
  ELSE
    DBMS_OUTPUT.PUT('  p_OutParameter = ' || p_OutParameter);
  END IF;

  IF (p_InOutParameter IS NULL) THEN
    DBMS_OUTPUT.PUT_LINE('  p_InOutParameter es nulo');
  ELSE
    DBMS_OUTPUT.PUT_LINE('  p_InOutParameter = ' ||
                         p_InOutParameter);
  END IF;

END ModeTest;
```

Llamamos a este procedimiento con el siguiente bloque:

```
DECLARE
  v_In NUMBER := 1;
  v_Out NUMBER := 2;
  v_InOut NUMBER := 3;
BEGIN
  DBMS_OUTPUT.PUT_LINE('Antes de llamar a ModeTest:');
  DBMS_OUTPUT.PUT_LINE('v_In = ' || v_In ||
                       '  v_Out = ' || v_Out ||
                       '  v_InOut = ' || v_InOut);

  ModeTest(v_In, v_Out, v_InOut);

  DBMS_OUTPUT.PUT_LINE('Después de llamar a ModeTest:');
  DBMS_OUTPUT.PUT_LINE('  v_In = ' || v_In ||
                       '  v_Out = ' || v_Out ||
                       '  v_InOut = ' || v_InOut);
END;
```

La salida sería:

```
Antes de llamar a ModeTest
v_In = 1  v_Out = 2  v_InOut = 3
Dentro de ModeTest:
p_InParameter = 1
p_OutParameter es Nulo
p_InOutParameter = 3
```

```
Al final de ModeTest
p_InParameter = 1
p_OutParameter = 7
p_InOutParameter = 8
Después de llamar a ModeTest
v_In = 1  v_Out = 7  v_InOut = 8
```

Literales y constantes como parámetros reales.

Debido al proceso de copia todo parámetro real que se corresponda con un parámetro OUT o IN OUT debe ser una variable, y no puede ser una constante o una expresión: debe existir un lugar para almacenar el valor que se devuelve.

Restricciones sobre los parámetros formales.

Cuando se llama a un procedimiento, se pasan los valores de los parámetros reales y, dentro del procedimiento, se hace referencia a los mismos utilizando los parámetros formales.
Las restricciones sobre las variables también se pasan como parte del mecanismo de paso de parámetros.
En la declaración de un procedimiento no se puede restringir la longitud de los parámetros de tipo CHAR o VARCHAR2, ni la precisión y/o escala de los de tipo NUMBER, ya que las restricciones se tomarán de los parámetros reales.

A modo de ejemplo el siguiente procedimiento no compilaría por presentar errores en la declaración de los parámetros:

```
CREATE OR REPLACE PROCEDURE ParameterLength (
  p_Parameter1 IN OUT VARCHAR2(10), -- Declaración errónea
  p_Parameter2 IN OUT NUMBER(3,1)) AS  --Declaración errónea.
BEGIN
  p_Parameter1 := 'abcdefghijklm';
  p_Parameter2 := 12.3;
END ParameterLength;
```

La declaración correcta habría sido ésta:

```
CREATE OR REPLACE PROCEDURE ParameterLength (
  p_Parameter1 IN OUT VARCHAR2,
  p_Parameter2 IN OUT NUMBER) AS
BEGIN
  p_Parameter1 := 'abcdefghijklmno';
  p_Parameter2 := 12.3;
END ParameterLength;
```

¿Cuáles son las restricciones sobre p_Parameter1 y p_Parameter2? Las que tuviesen los parámetros reales.
Si llamamos a ParameterLength así:

```
DECLARE
  v_Variable1 VARCHAR2(40);
  v_Variable2 NUMBER(7,3);
BEGIN
  ParameterLength(v_Variable1, v_Variable2);
END;
```

p_Parameter1 tendrá una longitud máxima de 40(que proviene del parámetro real v_variable1) y p_parameter2 una precisión de 7 y una escala de 3 (que proviene del parámetro real v_Variable2).

Consideremos ahora la siguiente llamada:

```
DECLARE
  v_Variable1 VARCHAR2(10);
  v_Variable2 NUMBER(7,3);
BEGIN
  ParameterLength(v_Variable1, v_Variable2);
END;
```

Al estar ahora p_Parameter1 restringido a 10 caracteres máximo (por v_Variable1), la asignación **p_Parameter1 := 'abcdefghijklmno'**; provoca el siguiente error en tiempo de ejecución ya que no habría suficiente espacio para alojar la cadena:

ERROR en línea 1:
ORA-06502: PL/SQL: error : **buffer de cadenas de caracteres demasiado pequeño**
numérico o de valor
ORA-06512: en "PL.PARAMETERLENGTH", línea 5
ORA-06512: en línea 5

%TYPE y parámetros de procedimientos

Aunque no se pueden declarar parámetros formales con restricciones, éstas pueden imponerse utilizando %TYPE. Si se declara un parámetro formal con %TYPE y el tipo subyacente está restringido, la restricción actuará sobre el parámetro formal, y no sobre el parámetro real.

Excepciones generadas en los subprogramas

Si ocurre un error dentro de un subprograma, se genera una excepción, pudiendo ser esta predefinida o definida por el usuario. Si el procedimiento no tiene una rutina para el tratamiento de excepciones, se pasa inmediatamente el control al entorno de llamada. Los valores de los parámetros formales con modo out o in out no se devuelven a los parámetros reales. Los parámetros reales tendrán los mismos valores que tenían en el momento de la llamada.

Paso de parámetros por referencia y por valor.

- **Paso por referencia**: Se pasa un puntero al parámetro real.
- **Paso por valor**: Se copia el valor del parámetro real al parámetro formal.

El paso por referencia es más rápido porque no hay copia.

De manera predeterminada, PL/SQL pasa los parámetros con modo IN por referencia y los parámetros con modo IN OUT y OUT por valor, para mantener la semántica de excepciones vista en la sección anterior.
Este comportamiento se puede modificar en versiones posteriores a Oracle8i mediante NOCOPY.

NOCOPY

Es una directiva de compilación. La sintaxis de declaración de los parámetros con esta indicación seria:

```
nombre_parametro[modo]NOCOPY tipodedato
```

Si NOCOPY está presente el compilador de PL/SQL intenta pasar el parámetro por referencia en lugar de por valor, tiene por tanto tiene sentido en los modos: in-out y out.

Semántica de las excepciones con NOCOPY.

Si se produce una excepción no tratada dentro del procedimiento después de modificar el parámetro formal, el valor original del parámetro real se habrá perdido.

Restricciones de NOCOPY.

En algunos casos, NOCOPY se ignora y el parámetro se pasa por valor. Los casos son:

- Si el paso del parámetro real necesita una conversion implícita al tipo de dato del parámetro formal.
- El parámetro real es un miembro de una colección. Sin embargo, si el parámetro real es colección entera, NOCOY si que tiene efecto.
- El parámetro real tiene una restricción de precisión, escala o NOT NULL.
- Cuando los parámetros formales y reales son ambos registros, uno de ellos o ambos se declaró usando %ROWTYPE o %TYPE y las restricciones de los respectivos campos no coinciden.
- Los parámetros reales y formales son ambos registros, el parámetro real se ha declarado de forma implícita (FOR v_reg in CURSOR) y las restricciones en los campos difieren
- Cuando el subprograma es invocado a través de un enlace de base de datos o como un procedimiento remotoo (RPC).

La ventaja principal de NOCOPY es la mejora en el rendimiento que puede lograrse en algunas ocasiones, siendo especialmente importante en paso de tablas PL/SQL de gran tamaño.

Notación posicional y notación nominal

Posicional: Los parámetros reales se asocian con los formales por la posición.

Nominal: Los parámetros reales se asocian con los formales de manera explícita en la llamada a la función

Veamos un ejemplo para aclarar ambos modos:

```
CREATE OR REPLACE PROCEDURE CallMe(
  p_ParameterA VARCHAR2,
  p_ParameterB NUMBER,
  p_ParameterC BOOLEAN,
  p_ParameterD DATE) AS
BEGIN
  NULL;
END CallMe;

DECLARE
  v_Variable1 VARCHAR2(10);
  v_Variable2 NUMBER(7,6);
  v_Variable3 BOOLEAN;
  v_Variable4 DATE;
BEGIN
  -- Aquí los parámetros se asocian por posición v_Variable1 con
p_ParameterA ...
  CallMe(v_Variable1, v_Variable2, v_Variable3, v_Variable4);
END;

DECLARE
  v_Variable1 VARCHAR2(10);
  v_Variable2 NUMBER(7,6);
  v_Variable3 BOOLEAN;
  v_Variable4 DATE;
BEGIN
 -- Aquí los parámetros se asocian por nombre

  CallMe(p_ParameterA => v_Variable1,
         p_ParameterB => v_Variable2,
         p_ParameterC => v_Variable3,
         p_ParameterD => v_Variable4);
END;


DECLARE
  v_Variable1 VARCHAR2(10);
  v_Variable2 NUMBER(7,6);
  v_Variable3 BOOLEAN;
  v_Variable4 DATE;
BEGIN
-- Aquí los parámetros se asocian por nombre, pudiéndose alterar el
orden definido
-- en la declaración del procedimiento
```

```
  CallMe(p_ParameterB => v_Variable2,
         p_ParameterC => v_Variable3,
         p_ParameterD => v_Variable4,
         p_ParameterA => v_Variable1);
END;

DECLARE
  v_Variable1 VARCHAR2(10);
  v_Variable2 NUMBER(7,6);
  v_Variable3 BOOLEAN;
  v_Variable4 DATE;
BEGIN
-- Mezcla de ambos modos. Los dos primeros parámetros por posición y
los dos
-- Siguientes por nombre
CallMe(v_Variable1, v_Variable2,
         p_ParameterC => v_Variable3,
         p_ParameterD => v_Variable4);
END;
```

Valores predeterminados en los parámetros

Los parámetros formales de los procedimientos y funciones pueden tener valores predeterminados. Si un parámetro tiene un valor predeterminado, no es obligatorio pasarlo desde el entorno de llamada. Si se pasa, el valor del parámetro real sustituirá al predeterminado.
La sintaxis a utilizar es la siguiente:

```
Nombre_parametro [modo] tipo [{:=| DEFAULT} valor inicial]
```

Ejemplos de uso:

```
CREATE OR REPLACE PROCEDURE DefaultTest (
  p_ParameterA NUMBER DEFAULT 10,
  p_ParameterB VARCHAR2 DEFAULT 'abcdef',
  p_ParameterC DATE DEFAULT SYSDATE) AS
BEGIN
  DBMS_OUTPUT.PUT_LINE(
    'A: ' || p_ParameterA ||
    '  B: ' || p_ParameterB ||
    '  C: ' || TO_CHAR(p_ParameterC, 'DD-MON-YYYY'));
END DefaultTest;

BEGIN
  DefaultTest(p_ParameterA => 7, p_ParameterC => '30-DEC-95');
END;

BEGIN
  -- Usamos el valor por defecto tanto para el parámetro  p_ParameterB
como para p_ParameterC.
  DefaultTest(7);
END;
```

8. PAQUETES

Introducción

Es una estructura PL/SQL que permite almacenar juntos una secuencia de objetos relacionados.
Tiene dos partes: **La especificación y el cuerpo**.
Cada una se almacena por separado en el diccionario de datos. Un paquete no puede ser local; sólo puede almacenarse. Los paquetes son menos restrictivos con respecto a las dependencias y presenta una serie de ventajas en cuanto al rendimiento.
En un paquete puede haber procedimientos, funciones, cursores, tipos, excepciones y variables y de esta forma, estos objetos, pueden referenciarse desde otros bloque PL/SQL, con lo que los paquetes permiten disponer de variables globales en PL/SQL.

Especificación de un paquete.

La especificación o cabecera de un paquete contiene información acerca del contenido del paquete. Sin embargo, no contiene el código de los subprogramas, sino las declaraciones formales de estos.
La sintaxis general para la creación de la cabecera de paquete es:

```
CREATE [OR REPLACE] PACKAGE nombre_paquete {[IS/AS]}
Especificación_procedimiento /
Especificación_función /
Declaración_variable /
Definición_tipo /
Declaración_excepción /
Declaración_cursor
END [nombre_paquete]
```

donde nombre_paquete es el nombre que le daremos al paquete.
Los elementos dentro del paquete (especificaciones de procedimientos y funciones, variables, etc.) son idénticos a como serían, si estuviesen en la sección declarativa de un bloque anónimo.
Las reglas sintácticas para la cabecera de un paquete son las mismas que para la sección declarativa, excepto en el caso de declaraciones de procedimientos y funciones.
Dichas reglas son:

- Los elementos del paquete pueden aparecer en cualquier orden. Sin embargo, un objeto debe ser declarado antes de poderlo referenciar.
- No es necesario que estén presentes todos los tipos de elementos.
- Las declaraciones de procedimientos y funciones deben ser declaraciones formales.

Ejemplo:

```
CREATE OR REPLACE PACKAGE ClassPackage AS
  -- Añadir un nuevo alumno al curso especificado.
  PROCEDURE AddStudent(p_StudentID  IN students.id%TYPE,
                       p_Department IN classes.department%TYPE,
                       p_Course     IN classes.course%TYPE);

  -- Eliminar al alumno especificado del curso especificado
  PROCEDURE RemoveStudent(p_StudentID  IN students.id%TYPE,
                          p_Department IN classes.department%TYPE,
                          p_Course     IN classes.course%TYPE);

  -- Excepción generada por RemoveStudent.
  e_StudentNotRegistered EXCEPTION;

  -- Tipo de tabla utilizada para el almacenamiento de la información
  -- de los alumnos.
  TYPE t_StudentIDTable IS TABLE OF students.id%TYPE
    INDEX BY BINARY_INTEGER;

-- Devuelve una tabla PL/SQL (parámetro p_IDS) que contiene a los
-- alumnos actualmente matriculados
PROCEDURE ClassList(p_Department  IN  classes.department%TYPE,
                    p_Course      IN  classes.course%TYPE,
                    p_IDs         OUT t_StudentIDTable,
                    p_NumStudents IN OUT BINARY_INTEGER);
END ClassPackage;
```

Cuerpo del paquete.

El cuerpo no puede ser compilado a menos que haya previamente compilado la cabecera correspondiente. El cuerpo contiene el código para las declaraciones formales de los subprogramas incluidos en la cabecera.
El cuerpo del paquete es opcional, si la cabecera no contiene ningún subprograma, el cuerpo no es necesario. Esta técnica es de utilidad para declarar variables globales.
Cualquier declaración formal de la cabecera del paquete debe ser sustanciada en el cuerpo. La especificación del procedimiento o función en el cuerpo debe ser idéntica al de la declaración formal de la cabecera, incluidos el nombre, los parámetros, tipos y modos.

```
CREATE [OR REPLACE] PACKAGE BODY nombre_paquete {[IS/AS]}
Declaraciones formales y código de los procedimientos y funciones de
la cabecera;
END [nombre_paquete];
```

Ejemplo: Para la cabecera vista anteriormente tendríamos el siguiente cuerpo.

```
CREATE OR REPLACE PACKAGE BODY ClassPackage AS
-- Añadir un nuevo alumno a l curso especificado.
  PROCEDURE AddStudent(p_StudentID  IN students.id%TYPE,
                       p_Department IN classes.department%TYPE,
                       p_Course     IN classes.course%TYPE) IS
  BEGIN
```

```
    INSERT INTO registered_students (student_id, department, course)
      VALUES (p_StudentID, p_Department, p_Course);
  END AddStudent;

-- Eliminar al alumno especificado del curso especificado
  PROCEDURE RemoveStudent(p_StudentID  IN students.id%TYPE,
                          p_Department IN classes.department%TYPE,
                          p_Course     IN classes.course%TYPE) IS
  BEGIN
    DELETE FROM registered_students
      WHERE student_id = p_StudentID
      AND department = p_Department
      AND course = p_Course;

-- Comprobar si la operación DELETE fue satisfactoria. Si no eliminó
-- Ninguna fila generar un error.
IF SQL%NOTFOUND THEN
      RAISE e_StudentNotRegistered;
    END IF;
  END RemoveStudent;

-- Devuelve una tabla  PL/SQL (parámetro p_IDS) que contiene a los
-- alumnos actualmente matriculados
  PROCEDURE ClassList(p_Department  IN  classes.department%TYPE,
                      p_Course      IN  classes.course%TYPE,
                      p_IDs         OUT t_StudentIDTable,
                      p_NumStudents IN OUT BINARY_INTEGER) IS

    v_StudentID  registered_students.student_id%TYPE;
    -- Cursor local para recuperar los alumnos matriculados.
    CURSOR c_RegisteredStudents IS
      SELECT student_id
        FROM registered_students
        WHERE department = p_Department
        AND course = p_Course;
  BEGIN
/* p_NumStudents será el índice de la tabla. Comenzará en 0, y será
incrementado en cada pasada del bucle de extracción. Al final del
bucle, contendrá el número de filas extraídas y, por tanto, el número
de filas que se devuelven en p_IDS.
*/
    p_NumStudents := 0;

    OPEN c_RegisteredStudents;
    LOOP
      FETCH c_RegisteredStudents INTO v_StudentID;
      EXIT WHEN c_RegisteredStudents%NOTFOUND;

      p_NumStudents := p_NumStudents + 1;
      p_IDs(p_NumStudents) := v_StudentID;
    END LOOP;
  END ClassList;
END ClassPackage;
```

Paquetes y ámbito.

Cualquier objeto declarado en la cabecera de un paquete está dentro de ámbito y es visible fuera del paquete, sin más que cualificar el objeto con el nombre del paquete (incluir como prefijo el nombre del paquete).

Nombrepaquete.objeto[(...)]

Por ejemplo, podemos llamar a ClassPackage.RemoveStudent desde el siguiente bloque PL/SQL:

```
BEGIN
      ClassPackage.RemoveStudent(10006,'HIS',101);
END;
```

Los procedimientos empaquetados pueden tener parámetros predeterminados, y pueden ser llamados utilizando notación posicional o nominal.
Dentro del cuerpo del paquete se puede hacer referencia a los objetos definidos en la cabecera sin necesidad de utilizar el nombre del paquete.

Ámbito de los objetos en el cuerpo de un paquete.

Tal y como están actualmente escritos, ClassPackage.AddStudent y ClassPackage.RemoveStudent actualizan la tabla registered_students.
Deberían actualizarse también las tablas students y classes.
Esta operación se hará añadiendo un procedimiento local al cuerpo del paquete, este procedimiento será llamado tanto por ClassPackage.AddStudent como ClassPackage.RemoveStudent. El paquete con este cambio quedaría así:

```
CREATE OR REPLACE PACKAGE ClassPackage AS
-- Añadir un nuevo alumno al curso especificado.
  PROCEDURE AddStudent(p_StudentID  IN students.id%TYPE,
                       p_Department IN classes.department%TYPE,
                       p_Course     IN classes.course%TYPE);

-- Eliminar al alumno especificado del curso especificado
  PROCEDURE RemoveStudent(p_StudentID  IN students.id%TYPE,
                          p_Department IN classes.department%TYPE,
                          p_Course     IN classes.course%TYPE);

-- Excepción generada por RemoveStudent.
  e_StudentNotRegistered EXCEPTION;

  -- Tipo tabla para almacenar la información de los estudiantes
matriculados.
  TYPE t_StudentIDTable IS TABLE OF students.id%TYPE
  INDEX BY BINARY_INTEGER;

-- Devuelve una tabla  PL/SQL (parámetro p_IDS) que contiene a los
-- alumnos actualmente matriculados
  PROCEDURE ClassList(p_Department  IN  classes.department%TYPE,
                      p_Course      IN  classes.course%TYPE,
```

```
                        p_IDs         OUT t_StudentIDTable,
                        p_NumStudents IN OUT BINARY_INTEGER);
END ClassPackage;
/

CREATE OR REPLACE PACKAGE BODY ClassPackage AS
-- Procedimiento que actualiza las tablas students and classes para
reflejar los
-- cambios. Si p_ADD   es TRUE, las tablas se actualizarán añadiendo
el alumno
-- al curso. Si es FALSE, se actualizan eliminando al alumno.
-- Este procedimiento es local al paquete y no es visible fuera de él.
PROCEDURE UpdateStudentsAndClasses(
    p_Add       IN BOOLEAN,
    p_StudentID IN students.id%TYPE,
    p_Department IN classes.department%TYPE,
    p_Course     IN classes.course%TYPE) IS

-- Número de créditos del curso solicitado.
 v_NumCredits  classes.num_credits%TYPE;
  BEGIN
    -- Primero determinar num_credits.
    SELECT num_credits
      INTO v_NumCredits
      FROM classes
      WHERE department = p_Department
      AND course = p_Course;

    IF (p_Add) THEN
      -- Añadir  NumCredits a la carga lectiva del alumno.
      UPDATE STUDENTS
        SET current_credits = current_credits + v_NumCredits
        WHERE ID = p_StudentID;

      -- Incrementar  current_students
      UPDATE classes
        SET current_students = current_students + 1
        WHERE department = p_Department
        AND course = p_Course;
     ELSE
       -- Borrar  NumCredits de la carga lectiva del alumno.
       UPDATE STUDENTS
         SET current_credits = current_credits - v_NumCredits
         WHERE ID = p_StudentID;

       -- Y decrementar current_students
       UPDATE classes
         SET current_students = current_students - 1
         WHERE department = p_Department
         AND course = p_Course;
     END IF;
  END UpdateStudentsAndClasses;

-- Añadir un nuevo alumno a l curso especificado.
```

```
  PROCEDURE AddStudent(p_StudentID  IN students.id%TYPE,
                       p_Department IN classes.department%TYPE,
                       p_Course     IN classes.course%TYPE) IS
  BEGIN
    INSERT INTO registered_students (student_id, department, course)
      VALUES (p_StudentID, p_Department, p_Course);

    UpdateStudentsAndClasses(TRUE, p_StudentID, p_Department,
                                   p_Course);
  END AddStudent;

-- Eliminar al alumno especificado del curso especificado
  PROCEDURE RemoveStudent(p_StudentID  IN students.id%TYPE,
                          p_Department IN classes.department%TYPE,
                          p_Course     IN classes.course%TYPE) IS
  BEGIN
    DELETE FROM registered_students
      WHERE student_id = p_StudentID
      AND department = p_Department
      AND course = p_Course;

-- Comprobar si la operación DELETE fue satisfactoria. Si no eliminó
-- Ninguna fila generar un error.
    IF SQL%NOTFOUND THEN
      RAISE e_StudentNotRegistered;
    END IF;

    UpdateStudentsAndClasses(FALSE, p_StudentID, p_Department,
                                   p_Course);

  END RemoveStudent;

-- Devuelve una tabla  PL/SQL (parámetro p_IDS) que contiene a los
-- alumnos actualmente matriculados
  PROCEDURE ClassList(p_Department  IN  classes.department%TYPE,
                      p_Course      IN  classes.course%TYPE,
                      p_IDs         OUT t_StudentIDTable,
                      p_NumStudents IN OUT BINARY_INTEGER) IS

    v_StudentID  registered_students.student_id%TYPE;

-- Cursor local para recuperar los alumnos matriculados.
    CURSOR c_RegisteredStudents IS
      SELECT student_id
        FROM registered_students
        WHERE department = p_Department
        AND course = p_Course;
  BEGIN
/* p_NumStudents será el índice de la tabla. Comenzará en 0, y será
incrementado en cada pasada del bucle de extracción. Al final del
bucle, contendrá el número de filas extraídas y, por tanto, el número
de filas que se devuelven en p_IDS.
*/
    p_NumStudents := 0;
```

```
    OPEN c_RegisteredStudents;
    LOOP
      FETCH c_RegisteredStudents INTO v_StudentID;
      EXIT WHEN c_RegisteredStudents%NOTFOUND;
      p_NumStudents := p_NumStudents + 1;
      p_IDs(p_NumStudents) := v_StudentID;
    END LOOP;
  END ClassList;
END ClassPackage;
```

Sobrecarga de subprogramas empaquetados

Dentro de un paquete, pueden sobrecargarse los procedimientos y funciones, es decir, puede haber más de un procedimiento o función con el mismo nombre, pero con distintos parámetros.
Por ejemplo, supongamos que queremos realizar dos procedimientos casi idénticos, para añadir un nuevo estudiante a una clase.
Se llamarán igual, pero el primero especificaremos el ID_estudiante, y el segundo, especificaremos el nombre y el apellido del estudiante.
Para el primero será sencillo, ya que es el propio id el que almacenaremos en la tabla clase, mientras que para el segundo, primero buscaremos su id a través del nombre y el apellido antes de la inserción.
En definitiva, serán dos procedimientos iguales con diferentes argumentos.

Veamos el código del paquete sobrecargado:

```
CREATE OR REPLACE PACKAGE ClassPackage AS
-- Añadir un nuevo alumno al curso especificado.
  PROCEDURE AddStudent(p_StudentID  IN students.id%TYPE,
                       p_Department IN classes.department%TYPE,
                       p_Course     IN classes.course%TYPE);

-- Añade un nuevo alumno pero especificando nombre y
-- apellidos en lugar del identificador.

  PROCEDURE AddStudent(p_FirstName IN students.first_name%TYPE,
                       p_LastName  IN students.last_name%TYPE,
                       p_Department IN classes.department%TYPE,
                       p_Course     IN classes.course%TYPE);

END ClassPackage;
/

CREATE OR REPLACE PACKAGE BODY ClassPackage AS
-- Añadir un nuevo alumno al curso especificado.
  PROCEDURE AddStudent(p_StudentID  IN students.id%TYPE,
                       p_Department IN classes.department%TYPE,
                       p_Course     IN classes.course%TYPE) IS
  BEGIN
    INSERT INTO registered_students (student_id, department, course)
```

```
      VALUES (p_StudentID, p_Department, p_Course);
  END AddStudent;

-- Añade un nuevo alumno pero especificando nombre y
-- apellidos en lugar del identificador.
  PROCEDURE AddStudent(p_FirstName IN students.first_name%TYPE,
                       p_LastName  IN students.last_name%TYPE,
                       p_Department IN classes.department%TYPE,
                       p_Course     IN classes.course%TYPE) IS
    v_StudentID students.ID%TYPE;
  BEGIN
    /* Obtener primero el identificador de la tabla students. */
    SELECT ID
      INTO v_StudentID
      FROM students
      WHERE first_name = p_FirstName
      AND last_name = p_LastName;

    -- Ahora podemos añadir el alumno por su identificador.
    INSERT INTO registered_students (student_id, department, course)
      VALUES (v_StudentID, p_Department, p_Course);
  END AddStudent;

END ClassPackage;
```

La sobrecarga está sujeta a diversas restricciones.

- No se pueden sobrecargar dos subprogramas si sus parámetros sólo difieren en el nombre o en el modo.
- No pueden sobrecargarse dos funciones basándose sólo en su tipo de retorno.
- Los parámetros de las funciones sobrecargadas deben diferir también en cuanto a familia de tipos. Un procedimiento no puede diferir en tan sólo un parámetro que en uno es char y en otro varchar2, pues pertenecen a la misma familia de tipos.

Inicialización del paquete.

La primera vez que se llama a un paquete, este es instanciado, lo que quiere decir que el paquete se lee de disco y se lleva a memoria, y se ejecuta el código-p (código objeto de oracle).
En este punto se asigna memoria a todas las variables definidas en el paquete. **Cada sesión tendrá su propia copia de las variables empaquetadas, asegurando que dos sesiones que ejecuten el mismo subprograma del mismo paquete utilicen posiciones de memoria distintas.**
En muchos casos, un cierto código de inicialización necesita ser ejecutado la primera vez que se instancia el paquete.
Para ello se añade una sección de inicialización al final del cuerpo del paquete.

```
CREATE [OR REPLACE] PACKAGE BODY nombre_paquete {[IS/AS]}
Declaraciones formales y código de los procedimientos y funciones de la
cabecera.
BEGIN
      Código de inicialización;
END [nombre_paquete];
```

El ejemplo clásico de uso de esta posibilidad es un programa para generar números aleatorios. El algoritmo cambia la "semilla" usada en el algoritmo para generar la secuencia de números, la semilla asegura que las secuencias no van a ser siempre las mismas.

```
CREATE OR REPLACE PACKAGE Random AS
  -- Generador de números aleatorios.  Usa el mismo algoritmo que
  -- la funcion rand() de C.

  -- Se usa para cambiar la semilla.  Para una semilla dada, se
  -- obtiene siempre la misma secuencia de números aleatorios.
  PROCEDURE ChangeSeed(p_NewSeed IN NUMBER);

  -- Devuelve un entero alatorio entre 1 y 32767
  FUNCTION Rand RETURN NUMBER;

  -- Lo mismo que Rand pero con interfaz procedural.
  PROCEDURE GetRand(p_RandomNumber OUT NUMBER);

  -- Devuelve un entero aleatorio entre 1 and p_MaxVal.
  FUNCTION RandMax(p_MaxVal IN NUMBER) RETURN NUMBER;

  -- Igual que RandMax, pero con interfaz procedural.
  PROCEDURE GetRandMax(p_RandomNumber OUT NUMBER,
                       p_MaxVal IN NUMBER);
END Random;
/

CREATE OR REPLACE PACKAGE BODY Random AS

  /* Usados para calcular el siguiente número. */
  v_Multiplier  CONSTANT NUMBER := 22695477;
  v_Increment   CONSTANT NUMBER := 1;

  /* Semilla para generar la secuencia aleatoria. */
  v_Seed        number := 1;

  PROCEDURE ChangeSeed(p_NewSeed IN NUMBER) IS
  BEGIN
    v_Seed := p_NewSeed;
  END ChangeSeed;

  FUNCTION Rand RETURN NUMBER IS
  BEGIN
    v_Seed := MOD(v_Multiplier * v_Seed + v_Increment,
                  (2 ** 32));
    RETURN BITAND(v_Seed/(2 ** 16), 32767);
  END Rand;

  PROCEDURE GetRand(p_RandomNumber OUT NUMBER) IS
  BEGIN
    -- Sencillamente llamar a Rand y devolver el valor.
    p_RandomNumber := Rand;
```

```
  END GetRand;

  FUNCTION RandMax(p_MaxVal IN NUMBER) RETURN NUMBER IS
  BEGIN
    RETURN MOD(Rand, p_MaxVal) + 1;
  END RandMax;

  PROCEDURE GetRandMax(p_RandomNumber OUT NUMBER,
                       p_MaxVal IN NUMBER) IS
  BEGIN
    -- Simplemente llamar a RandMax y devolver el valor.
    p_RandomNumber := RandMax(p_MaxVal);
  END GetRandMax;

BEGIN
  /* Inicialización del paquete. Iniciar la semilla con la hora actual
en segundos */
  ChangeSeed(TO_NUMBER(TO_CHAR(SYSDATE, 'SSSSS')));
END Random;
```

Privilegios y subprogramas almacenados

Los paquetes y subprogramas almacenados son objetos del diccionario de datos y, como tales, pertenecen a un usuario, o esquema, de la base de datos en particular. El resto de usuarios pueden acceder a estos objetos si se les otorgan los privilegios adecuados sobre los mismos. Los privilegios y roles también entran en juego cuando se crea un objeto almacenado, afectando el acceso disponible dentro del subprograma.

Para conceder privilegios de ejecución sobre nuestros subprogramas utilizaremos el privilegio EXECUTE.

Supongamos las dos siguientes funciones:

```
CREATE OR REPLACE FUNCTION AlmostFull (
  p_Department classes.department%TYPE,
  p_Course     classes.course%TYPE)
  RETURN BOOLEAN IS

  v_CurrentStudents NUMBER;
  v_MaxStudents     NUMBER;
  v_ReturnValue     BOOLEAN;
  v_FullPercent     CONSTANT NUMBER := 80;
BEGIN
  -- Obtiene el número actual y el número máximo de alumnos para el
  -- curso solicitado
    SELECT current_students, max_students
    INTO v_CurrentStudents, v_MaxStudents
    FROM classes
    WHERE department = p_Department
    AND course = p_Course;
```

```
  -- Si el curso contiene más alumnos que el porcentaje dado por
  -- v_FullPercent devuelve TRUE. En caso contrario devuelve FALSE.
  IF (v_CurrentStudents / v_MaxStudents * 100) >= v_FullPercent THEN
    v_ReturnValue := TRUE;
  ELSE
    v_ReturnValue := FALSE;
  END IF;

  RETURN v_ReturnValue;
END AlmostFull;
/

-- Ahora el procedimiento RecordFullClasses que utiliza la función
anterior:
CREATE OR REPLACE PROCEDURE RecordFullClasses AS
  CURSOR c_Classes IS
    SELECT department, course
      FROM UserA.classes;
BEGIN
  FOR v_ClassRecord IN c_Classes LOOP
    -- Almacenar en temp_table todas las clases que no tienen mucho
espacio libre.
    IF AlmostFull(v_ClassRecord.department,
                        v_ClassRecord.course) THEN
      INSERT INTO temp_table (char_col) VALUES
        (v_ClassRecord.department || ' ' || v_ClassRecord.course ||
         ' is almost full!');
    END IF;
  END LOOP;
END RecordFullClasses;
```

Suponga que tanto las tablas classes y temp_table como el procedimiento RecordFullClasses y la función AlmostFull pertenecen a un usuario llamado USERA.

Si otorgamos el privilegio EXECUTE sobre el procedimiento RecordFullClasses a otro usuario de la base de datos, como pueda ser USERB de la manera:

```
GRANT EXECUTE ON RECORDFULLCLASSES TO USERB;
```

Entonces userb podría utilizar el procedimiento como se muestra a continuación:

```
BEGIN
   UserA.RecordFullClasses;
END;
```

RecordFullClasses hace una inserción en la tabla temp_table. Supongamos que el usuario userB es propietario de una tabla llamada también temp_table.
¿En cúal de las dos tablas se haría la inserción?

Se hará en la tabla UserA. Esto lo podemos expresar de la siguiente forma:

De forma predeterminada, un subprograma se ejecuta bajo el conjunto de privilegios de su propietario.

Aunque sea UserB quien llama a RecordFullClasses, RecordFullClasses pertenece a UserA. Por tanto, el identificador temp_table hace referencia a la tabla de UserA y no a la del usuario UserB.

Subprogramas almacenados y roles

Supongamos que cambiamos la situación anterior de la siguiente manera:
UserA es propietario de la tabla classes y de la función almostfull.
UserB es propietario de temp_table y del procedimiento RecordFullClasses.

El procedimiento RecordFullClasses tendría que ser reescrito del siguiente modo:

```
CREATE OR REPLACE PROCEDURE RecordFullClasses AS
  CURSOR c_Classes IS
    SELECT department, course
    FROM UserA.classes;
BEGIN
  FOR v_ClassRecord IN c_Classes LOOP
-- Almacenar en temp_table todas las clases que no tienen mucho
espacio libre.
    IF UserA.AlmostFull(v_ClassRecord.department,
                        v_ClassRecord.course) THEN
      INSERT INTO temp_table (char_col) VALUES
        (v_ClassRecord.department || ' ' || v_ClassRecord.course ||
         ' is almost full!');
    END IF;
  END LOOP;
END RecordFullClasses;
```

Para que este procedimiento se compile correctamente, UserA debe haber concedido a UserB los privilegios SELECT sobre classes y EXECUTE sobre AlmostFull.
Además, la concesión de los privilegios tiene que realizarse explícitamente, y no a través de un rol.
Las siguientes concesiones, ejecutadas por el usuario UserA permitirían la correcta compilación de UserB.RecordFullClasses.

```
GRANT SELECT ON classes TO UserB;
GRANT EXECUTE ON AlmostFull TO UserB;
```

Una concesión de privilegios realizada a través de un rol intermedio, como la siguiente:

```
CREATE ROLE UserA_Role;
GRANT SELECT ON classes TO UserA_Role;
GRANT EXECUTE ON AlmostFull TO UserA_Role;
GRANT UserA_Role TO UserB;
```

no funcionaría.

La regla vista anteriormente se afina de la siguiente manera:

Un subprograma se ejecuta bajo los privilegios que se han concedido explícitamente a su propietario, y no de los concedidos a través de un rol.

La razón está en el acoplamiento temprano de variables en el proceso de compilación. GRANT y REVOKE son instrucciones de definición de datos (DDL) y **su efecto es inmediato**, y los nuevos privilegios se registran en el diccionario de datos.
Todas las sesiones pueden ver el nuevo conjunto de privilegios. Sin embargo, esto no funciona igual en el caso de roles.
Un rol se concede a un usuario, y ese usuario luego puede decidir inhabilitar el rol mediante la instrucción SET ROLE. La diferencia es que SET ROLE afecta únicamente a una sesión, mientras que GRANT y REVOKE afectan a todas las sesiones.
Para permitir que los privilegios concedidos a través de un rol se utilizasen dentro de disparadores y subprogramas almacenados, tendrían que comprobarse en tiempo de ejecución.

Para mantener el acoplamiento temprano, todos los roles se encuentran inhabilitados dentro de los disparadores y procedimientos almacenados.

9. SQL dinámico.

Como hemos visto, PL/SQL utiliza acoplamiento temprano para ejecutar instrucciones SQL, y como consecuencia, sólo se pueden incluir instrucciones de manipulación de datos en los bloques PL/SQL. Para superar esta limitación utilizaremos **SQL dinámico**.

Existen dos técnicas para ejecutar SQL dinámico. La primera es a través del paquete DBMS_SQL, utilizada antes de Oracle8i y que no vamos a explicar por ser su uso mucho más engorroso.
La segunda técnica, introducida a partir de Oracle8i se denomina Sql dinámico nativo. El SQL dinámico nativo, es parte integrante del lenguaje y por tanto más cómodo de utilizar que el paquete DBMS_SQL.

Ejecución de instrucciones de no consulta y de bloques PL/SQL

La instrucción básica utilizada en instrucciones que no sean de consulta es la instrucción **EXECUTE IMMEDIATE**.
Veamos un ejemplo de su uso:

```
DECLARE
  v_SQLString  VARCHAR2(200);
  v_PLSQLBlock VARCHAR2(200);
BEGIN
  --  Primero crea una tabla temporal utilizando un literal.
  -- Obsérvese que no hay punto y coma al final de la cadena.
  EXECUTE IMMEDIATE
    'CREATE TABLE execute_table (col1 VARCHAR(10))';

-- Insertamos algunas filas utilizando una cadena. No hay
-- punto y coma al final de la cadena
 FOR v_Counter IN 1..10 LOOP
    v_SQLString :=
        'INSERT INTO execute_table
         VALUES (''Row ' || v_Counter || ''')';
    EXECUTE IMMEDIATE v_SQLString;
  END LOOP;

 -- Imprimimos el contenido de la tabla usando un bloque PL/SQL
 -- anónimo. Aquí insertamos el bloque entero en una cadena,
 -- incluyendo el punto y coma.

v_PLSQLBLock :=
    'BEGIN
       FOR v_Rec IN (SELECT * FROM execute_table) LOOP
         DBMS_OUTPUT.PUT_LINE(v_Rec.col1);
       END LOOP;
     END;';
  -- Ahora ejecutamos el bloque anónimo.
 EXECUTE IMMEDIATE v_PLSQLBlock;
```

```
  -- Finalmente, borramos la tabla.
  EXECUTE IMMEDIATE 'DROP TABLE execute_table';
END;
```

El ejemplo anterior muestra varios usos de EXECUTE IMMEDIATE:
ejecución de instrucciones DDL, DML y bloques anónimos.

El uso de SQL dinámico nos da la ventaja de poder utilizar sentencias DDL dentro de un bloque PL/SQL, circunstancia que en algunos procedimientos nos puede resultar útil.

La siguiente función permite crear usuarios de una manera cómoda asignándoles los roles connect y resource:

```
CREATE OR REPLACE PROCEDURE CREARUSUARIO(P_NOMBRE VARCHAR2,P_CLAVE
VARCHAR2) IS
V_CREARUSUARIO VARCHAR2(250);
V_ROLE VARCHAR2(150);
BEGIN
  V_CREARUSUARIO:='CREATE USER '||P_NOMBRE ||' IDENTIFIED BY
'||P_CLAVE || ' DEFAULT TABLESPACE USERS TEMPORARY TABLESPACE TEMP';
  EXECUTE IMMEDIATE V_CREARUSUARIO;
  V_ROLE:='GRANT CONNECT,RESOURCE TO '||P_NOMBRE;
  EXECUTE IMMEDIATE V_ROLE;
END;
```

El siguiente bloque de código utiliza la función anterior para crear los usuarios de nuestra clase:

```
DECLARE
V_USUARIO VARCHAR2(10);
V_CLAVE VARCHAR2(10);
BEGIN
 FOR I IN 1..15 LOOP
  V_USUARIO:='VS1DAW'||TO_CHAR(I);
  V_CLAVE:='VS1DAW'||TO_CHAR(I);
  CREARUSUARIO(V_USUARIO,V_CLAVE);
 END LOOP;
END;
```

Debido a lo estudiado en la sección anterior "privilegios y subprogramas" y que resumíamos así:

> Un subprograma se ejecuta bajo los privilegios que se han concedido explícitamente a su propietario, y no de los concedidos a través de un rol.

Para que el procedimiento crearusuario funcione adecuadamente se deben de haber concedido al propietario del procedimiento los privilegios CREATE TABLE, CREATE USER de manera explícita y no a través de un rol. No funcionaría el procedimiento incluso siendo DBA. Además para poder conceder los roles CONNECT y RESOURCE al usuario que estamos creando se nos deben haber concedido estos roles también de manera explícita con la opción **with admin option.**

10. Disparadores de base de datos (triggers)

Introducción

Los disparadores se asemejan a los procedimientos y funciones en que son bloques PL/SQL, nominados con secciones declarativa, ejecutable y de manejo de excepciones.
Deben almacenarse en la base de datos y se ejecutan de manera implícita cada vez que tiene lugar el suceso de disparo (INSERT, UPDATE Y DELETE).
Los disparadores pueden emplearse para:

- El mantenimiento de restricciones de integridad complejas.
- La auditoría de la información contenida en una tabla, registrando los cambios realizados y la identidad del que los llevó a cabo.
- El aviso automático a otros programas de que hay que llevar a cabo una determinada acción, cuando se realiza un cambio en una tabla.

La sintaxis es:

```
CREATE [OR REPLACE] TRIGGER nombre_disparador
{BEFORE | AFTER} suceso_disparo ON nombre_tabla
[FOR EACH ROW] [WHEN condición_disparo]
cuerpo_disparador;
```

Los disparadores tienen un espacio de nombres separado, por lo que un disparador puede tener igual nombre que la tabla asociada, aunque no es recomendable.

Disparadores dml

Se activan a través de instrucciones DML y el tipo de instrucción determina el tipo de disparador.
Los disparadores pueden definirse para las operaciones **INSERT**, **UPDATE** O **DELETE**, y pueden dispararse antes (**BEFORE**) o después (**AFTER**) de la operación.
Además, el nivel de los disparadores puede ser a **nivel de orden** (se activan una única vez, antes o después de la orden que los dispara) o a **nivel de fila**, identificados por la cláusula **FOR EACH ROW**, que se activan una vez por cada fila afectada por la orden que provocó el disparo.
A partir de las versión 2.1 de PL/SQL un disparador puede activarse para más de un tipo de orden de disparo (Ej: AFTER INSERT OR UPDATE OR DELETE) y también una tabla puede tener más de un disparador de cada tipo. Veremos este tipo de forma más detallada en el apartado Creación de disparadores DML.

Disparadores de sustitución.

Sólo se definen sobre vistas y a nivel de filas. Son necesarios porque si la vista se define basada en combinaciones, a veces no son actualizables.

Por ejemplo, supongamos que tenemos una vista con la siguiente información sobre un empleado: NUMEMP, NOMBRE y la descripción del ALOJAMIENTO.

La vista ha sido creada a partir de las tablas EMPLEADO y ALOJAMIENTO a través de la combinación los campos alojamiento y numaloj.

Si intentamos insertar una tupla en esta vista sólo nos dejaría insertar el numemp y el nombre, pues alojamiento es de la otra tabla relacionada.

Si quisiéramos insertar tuplas en esta vista con sus tres atributos deberíamos crear un disparador para que antes de la inserción, se buscase en la tabla alojamiento, el código del alojamiento a partir del alojamiento de la tupla a insertar en la vista, y a través de él, poder ya insertar los tres atributos en la tabla empleado.
De esta forma al visualizar de nuevo la vista se actualizará con los nuevos datos.

```
CREATE OR REPLACE TRIGGER TRVEMP
INSTEAD OF INSERT OR UPDATE OR DELETE ON VEMP
FOR EACH ROW
DECLARE
   NALOJ EMPLEADO.ALOJAMIENTO%TYPE;
BEGIN
   IF INSERTING OR UPDATING THEN
       SELECT NUMALOJ INTO NALOJ
       FROM ALOJAMIENTO
       WHERE ALOJAMIENTO=:NEW.ALOJAMIENTO;
   END IF;
   IF INSERTING THEN
       INSERT INTO EMPLEADO (NUMEMP,NOMBRE,ALOJAMIENTO)
       VALUES(:NEW.NUMEMP,:NEW.NOMBRE,NALOJ);
   ELSIF UPDATING THEN
       UPDATE EMPLEADO
       SET NOMBRE=:NEW.NOMBRE,ALOJAMIENTO=NALOJ
       WHERE NUMEMP=:OLD.NUMEMP;
   ELSE
       DELETE FROM EMPLEADO
       WHERE NUMEMP=:OLD.NUMEMP;
  END IF;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
         RAISE_APPLICATION_ERROR(-20000,'ALOJAMIENTO NO VALIDO');
END;
```

Disparadores del sistema.

Oracle8i y versiones posteriores ofrecen un tercer tipo de disparador. Un **disparador de sistema** se dispara debido a dos tipos de sucesos distintos: **DDL, o de base de datos**.
Los sucesos DDL incluyen las instrucciones **CREATE**, **ALTER** o **DROP**, mientras que los sucesos de base de datos incluyen la **conexión/desconexión del servidor**, el **inicio/finialización** de la sesión de usuario y los **errores del servidor**.

La sintaxis para la creación de un disparador de sistema es:

```
CREATE [OR REPLACE TRIGGER ] [esquema.]nombre_disparador
{BEFORE|AFTER}
{lista_sucesos_ddl |lista_sucesos_base_datos}
ON DATABASE | [esquema.]SCHEMA
[claúsula_when]
cuerpo_disparador;
```

donde lista_sucesos_ddl representa uno o más sucesos DDL (separados por la palabra clave OR), y lista_sucesos_base_datos representa uno o más sucesos de base de datos (separados por la palabra clave OR).

Lista de suscesos DDL y de base de datos

SUCESO	TEMPORIZACIONES PERMITIDAS	DESCRIPCION
STARTUP	AFTER	Se dispara cuando se inicia una instancia
SHUTDOWN	BEFORE	Cuando se cierra una instancia
SERVERERROR	AFTER	Cuando ocurre un error
LOGON	AFTER	Después de la conexión se un usuario a la bd.
LOGOFF	BEFORE	Después de la finalización de una sesión de usuario.
CREATE	BEFORE|AFTER	Antes o después de la creación de un objeto de esquema.
DROP	BEFORE|AFTER	Antes o después del borrado de un objeto de esquema.
ALTER	BEFORE|AFTER	Antes o después de la modificación de un objeto de esquema.

Dentro del cuerpo del disparador podemos obtener información del suceso de disparo mediante determinadas funciones de atributos de sucesos. Son propiedad del usuario SYS, y por tanto al llamarlas precederemos la palabra SYS delante de la función. He aquí alguna de ellas:

- **SYSEVENT:** Suceso que activó el disparador
- **INSTANCE_NUM:** Numero de instancia actual
- **DATABASE_NAME:** Nombre de la base de datos
- **SERVER_ERROR:** Acepta un argumento de tipo number. Devuelve el error situado en la pila de errores indicada por el argumento. La posición 1 es la cabeza de la pila.
- **LOGIN_USER:** Identificador de usuario que activó el disparador.
- **DICTIONARY_OBJ_TYPE:** tipo de objeto sobre el que tuvo lugar la operación ddl
- **DICTIONARY_OBJ_NAM:** nombre del objeto sobre el que tuvo lugar la operación ddl
- **DICTIONARY_OBJ_OWNER:** propietario del objeto sobre el que tuvo lugar la operación ddl

También podemos obtener información a través del la función userenv que tiene la siguiente sintaxis:

Userenv(opción), donde opción puede ser una de las siguientes:

- 'ISDBA': Si actualmente se tiene el rol dba habilitado-
- 'LANGUAGE': Lenguaje y territorio actual de la sesión.
- 'SESSIONID': Identificador de sesión.

Para poder crear un disparador de sistema es necesario poseer el privilegio **ADMINISTER DATABASE TRIGGER**. Si posteriormente a la creación del disparador se revocara este privilegio, el creador del disparador podría borrarlo, pero no modificarlo.

Ejemplo de disparador que controla el acceso a la base de datos:

```
CREATE TABLE ACCESOBD(
   SESION NUMBER,
   USUARIO VARCHAR2(30),
   FECHA DATE
);
-- Tabla donde almacenaremos quien se conecta a la base de datos,
guardamos el usuario, la sesión y la fecha.

CREATE OR REPLACE TRIGGER CONTROLACCESOBD
 AFTER LOGON ON DATABASE
 BEGIN
INSERT INTO ACCESOBD VALUES(USERENV('SESSIONID'),USER,SYSDATE);
END;
```

Restricciones de los disparadores.

El cuerpo de un disparador es un bloque PL/SQL. Cualquier orden que sea legal en un bloque PL/SQL, es legal en el cuerpo de un disparador, con las **siguientes restricciones**:

- **No se admiten órdenes de control de transacciones** (no commit, no rollback, no savepoint). Ni que las contengan funciones o procedimientos que sean llamados desde los disparadores. El disparador se activa como parte de la orden que provocó el disparo y forma parte de la misma transacción de dicha orden. **Cuando la orden que provoca el disparo es confirmada o cancelada, se confirma o cancela también el trabajo realizado por el disparador.**
- De forma análoga cualquier procedimiento o función que se llame desde el disparador no puede ejecutar ninguna instrucción de control sobre transacciones.
- En el cuerpo de un disparador no se pueden declarar variables de tipo LONG o LONG RAW. Además, :new y :old no pueden hacer referencia a columnas de la tabla sobre la que está definido el disparador que contenga datos de estos tipos.
- Restricciones sobre tablas mutantes que veremos más adelante.

Los disparadores y el diccionario de datos.

La vista **user_triggers** contiene información de los disparadores de nuestro esquema.

Eliminación y deshabilitación de disparadores.

DROP TRIGGER nombre_disparador; → Elimina el disparador.
ALTER TRIGGER nombre_disparador [DISABLE/ENABLE]; → Deshabilita o habilita el disparador.

Codigo-p de un disparador.

A partir de la versión PL/SQL 2.3 (Oracle7 versión 7.3) los disparadores se almacenan en forma compilada (código-p) y también se almacena información de sus dependencias. De esta forma, un disparador puede ser automáticamente inválido por motivos de dependencia, que volverá a compilarse la siguiente vez que se dispare.

Creación de disparadores dml.

Orden de activación de los disparadores.

Los disparadores se activan al ejecutarse la orden DML (INSERT, UPDATE O DELETE) sobre una tabla de la base de datos.

Pueden dispararse antes o después de la ejecución de la instrucción y pueden además activarse una vez por cada fila afectada por la orden (**disparadores a nivel de fila**) o bien una única vez por cada instrucción (**disparadores a nivel de orden**).

Una tabla puede tener cualquier número de disparadores definido sobre la misma, incluyendo más de un disparador DML del mismo tipo. Los disparadores del mismo tipo se activarán de forma secuencial, aunque el orden no está especificado. En el caso de disparadores de distinto tipo el orden de ejecución es el siguiente:

a) Ejecutar, si existe, el disparador de tipo **before** a nivel de **orden**.
b) Para cada fila a la que afecta la orden:
   b.1) Ejecutar, si existe, el disparador de tipo **before** a nivel de **fila**.
   b.2) Ejecutar la propia **orden** y ejecutar las restricciones de integridad.
   b.3) Ejecutar, si existe, el disparador de tipo **after** a nivel de **fila**.
c) Completar el chequeo de las restricciones de integridad diferidas (aquí por ejemplo se comprueban las restricciones de clave foránea).
d) Ejecutar, si existe, el disparador de tipo **after** a nivel de **orden**.

El orden en que se activan los disparadores de un mismo tipo no está definido. Si el orden es importante, se habrán de combinar todas las operaciones en un único disparador.

Utilización de :old y :new en los disparadores a nivel de fila.

Un disparador con nivel de fila se ejecuta una vez por cada fila procesada por la orden que provoca el disparo. Dentro del disparador, puede accederse a la fila que está siendo actualmente procesada utilizando dos pseudoregistros, **:old** y **:new**.
Son como registros del tipo tabladisparo%rowtype, pero tienen la restricción que no se pueden utilizar como un conjunto sino que sólo se pueden utilizar para acceder a las columnas individualizadas.

Significado de :old y :new en función de la orden que provoca el disparo:

Orden de disparo	:Old	:New
INSERT	No definido: todos los campos toman valor null.	Valores que serán insertados cuando se complete la orden.
UPDATE	Valores originales de la fila, antes de la actualización.	Nuevos valores que serán escritos cuando se complete la orden.
DELETE	Valores originales antes del borrado de la fila.	No definido; todos los campos toman valor NULL.

Ejemplo de uso:

```
CREATE OR REPLACE TRIGGER TRIGGER_CLI
BEFORE INSERT ON CLIENTE
FOR EACH ROW
BEGIN
      SELECT SECUENCIACLI.NEXTVAL
      INTO :NEW.ID
      FROM DUAL;
END TRIGGER_CLI;
```

Uso de los predicados de los disparadores: inserting, updating y deleting.

En un disparador de tipo: **insert or update or delete** hay tres funciones booleanas que pueden emplearse para determinar qué operación ha provocado el disparo de dicho disparador. Se trata de los predicados:

- Inserting: Devuelve TRUE si el suceso de disparo ha sido una orden insert.
- Updating: Devuelve TRUE si el suceso de disparo ha sido una orden update.
- Deleting: Devuelve TRUE si el suceso de disparo ha sido una orden delete.

Ejemplo de uso

El siguiente disparador registra los cambios que se realizan en la tabla registered_students. Se guarda en la tabla RS_AUDIT el tipo de cambio:

- Inserción → ChangeType=I
- Actualización → ChangeType=U
- Borrado → ChangeType=D

Se resgistra también la fecha y el usuario que realiza en cambio, asi como, los valores anteriores y posteriores al cambio realizado.

```
CREATE OR REPLACE TRIGGER LogRSChanges
  BEFORE INSERT OR DELETE OR UPDATE ON registered_students
  FOR EACH ROW
DECLARE
  v_ChangeType CHAR(1);
BEGIN
  /* Usaremos 'I' para INSERT, 'D' para DELETE, y 'U' para UPDATE. */
  IF INSERTING THEN
    v_ChangeType := 'I';
  ELSIF UPDATING THEN
    v_ChangeType := 'U';
  ELSE
    v_ChangeType := 'D';
  END IF;
/* Registrar todos los cambios hechos a registered_students en
   RS_audit. Usamos SYSDATE para obtener la fecha y hora a la
   que se hace el cambio y USER para obtener el identificador de
   usuario del usuario actual. */
  INSERT INTO RS_audit(change_type, changed_by,
timestamp,old_student_id, old_department, old_course, old_grade,
     new_student_id, new_department, new_course, new_grade)
  VALUES(v_ChangeType, USER, SYSDATE,:old.student_id, :old.department,
:old.course, :old.grade,
     :new.student_id, :new.department, :new.course, :new.grade);
END LogRSChanges;
```

La cláusula when.

Válida únicamente para disparadores a nivel de fila. Si aparece, el cuerpo del disparador se ejecutará únicamente sobre aquellas filas que cumplan la condición especificada en la cláusula WHEN. La sintaxis es when condición_disparo. Dentro de la condición de disparo se puede hacer referencia a :new y :old, pero sin utilizar el carácter dos puntos.

Ejemplo de uso:

```
CREATE OR REPLACE TRIGGER checkcredits
BEFORE INSERT OR UPDATE OF current_credits on students
FOR EACH ROW
WHEN(new.current_credits>20)
BEGIN
  /* CUERPO DEL DISPARADOR*/
END;
```

Solo se ejecutaría si el número de créditos que un alumno esta cursando es superior a 20.

Sería equivalente a:

```
CREATE OR REPLACE TRIGGER checkcredits
BEFORE INSERT OR UPDATE OF current_credits on students
FOR EACH ROW
BEGIN
  IF:new.current_credits > 20 THEN
       /* CUERPO DEL DISPARADOR*/
  END IF;
END;
```

Tablas mutantes.

Hay ciertas restricciones sobre las tablas y columnas a las que puede acceder el cuerpo de un disparador. Una **tabla mutante** es una tabla que está modificándose actualmente por una orden DML. **Para un disparador, esta es la tabla sobre la que está definido.**
Las tablas que puedan necesitar ser actualizadas como resultado de restricciones de integridad referencial DELETE CASCADE también son mutantes.
Las tablas se consideran mutantes únicamente para la sesión que lanza la orden que la modifica.

Las tablas nunca se consideran mutantes para los disparadores a nivel de orden a no ser que el disparador se haya activado como resultado de una orden DELETE CASCADE.

Para todos los disparadores de nivel de fila, o disparadores a nivel de orden que se han activado como resultado de una orden DELETE CASCADE, hay una restricción relacionada con las tablas mutantes.
Estas restricciones previenen al disparador de ver un conjunto de datos inconsistente:

> Las sentencias SQL del cuerpo del disparador no pueden leer o modificar una tabla mutante para dicho disparador.

Excepción a la regla anterior:

Si una operación INSERT afecta a una única fila, el disparador de tipo **BEFORE a nivel de fila** para esa fila no trata a la tabla de disparo como mutante.
Sentencias insert que afectan a más de una fila, como insert into tabla select... no se consideran inserciones de fila única, aunque el resultado de ellas solo produzca una fila.

Ejemplo: Controlar que no se inserten más de tres titulares en una cuenta bancaria.

Dado el siguiente subesquema:

```
CREATE TABLE CLIENTE(
  NUMCLI NUMBER,
  NOMBRE VARCHAR2(40),
  CONSTRAINT PK_CLIENTE
    PRIMARY KEY (NUMCLI)
)
;
CREATE TABLE CUENTA(
   NUMCUENTA NUMBER,
   SALDO NUMBER,
   CONSTRAINT PK_CUENTA
     PRIMARY KEY (NUMCUENTA)
)
;
CREATE TABLE TITULAR(
   NUMCUENTA NUMBER,
   NUMCLI NUMBER,
   CONSTRAINT PK_TITULAR
     PRIMARY KEY (NUMCLI,NUMCUENTA),
   CONSTRAINT FK_TITULAR_CLIENTE
     FOREIGN KEY (NUMCLI)REFERENCES CLIENTE(NUMCLI),
   CONSTRAINT FK_TITULAR_CUENTA
     FOREIGN KEY(NUMCUENTA)REFERENCES CUENTA(NUMCUENTA)
)
;
```

Sin el problema de la tabla mutante el problema lo resolveríamos mediante el siguiente disparador:

```
CREATE OR REPLACE TRIGGER CONTROLTITULARESFILA
BEFORE INSERT OR UPDATE ON TITULAR
FOR EACH ROW
DECLARE
  V_NUMTITULARES NUMBER;
  V_MAXTITULARES CONSTANT NUMBER:=3;
BEGIN
    SELECT COUNT(NUMCLI)
    INTO V_NUMTITULARES
    FROM TITULAR
    WHERE NUMCUENTA=:NEW.NUMCUENTA;
    IF V_NUMTITULARES>V_MAXTITULARES  THEN
       RAISE_APPLICATION_ERROR(-20000,'PUEDE HABER UN MAXIMO DE 3
TITULARES POR CUENTA');
    END IF;
END;
```

Si tratamos de ejecutarlo y ponerlo a prueba mediante una orden de actualización, obtendremos el siguiente error:

ORA-04091: la tabla CUENTA.TITULAR está mutando, puede que el disparador/la función no puedan verla

La solución consiste en **leer de la tabla titular con un disparador a nivel de orden**, para el cual la tabla no es mutante. Sin embargo, los disparadores a nivel de orden no tienen acceso a los seudoregistros :new y :old. **¿Qué hacer entonces?.**

Crearemos dos disparadores: uno a nivel de fila y uno a nivel de orden.

En el disparador a nivel de fila se registra el valor de :new.numcuenta, pero no se lee de la tabla titular. La consulta se realiza en el disparador a nivel de orden que utiliza el valor registrado en el disparador a nivel de fila.

¿Cómo y donde registrar el valor?. Utilizaremos una tabla PL/SQL definida dentro de un paquete. De esta forma si la orden afecta a más de una fila se registrarán todos los valores (pensemos en una orden update). Además, cada sesión obtiene su propia instancia de las variables empaquetadas, por lo que no hay que preocuparse de las actualizaciones simultáneas realizadas por sesiones distintas.
La solución para nuestro problema ya implementada es la siguiente:

```
/*****************************************************************/
/* PAQUETE EN CUYAS ESTRUCTURAS SE ALMACENARAN LOS DATOS DE LAS     */
/* INSERCIONES  MODIFICACIONES O BORRADOS DE LA TABLA CUENTA        */
/*****************************************************************/
CREATE OR REPLACE PACKAGE PACKCONTROLCUENTA AS
  TYPE T_CUENTA IS TABLE OF TITULAR.NUMCUENTA%TYPE
  INDEX BY BINARY_INTEGER;
  V_CUENTA T_CUENTA;
  V_FILAS NUMBER:=0;
END;
/
/*****************************************************************/
CREATE OR REPLACE TRIGGER ANOTARCUENTA
BEFORE INSERT OR UPDATE ON TITULAR
FOR EACH ROW
DECLARE
I BINARY_INTEGER;
BEGIN
  PACKCONTROLCUENTA.V_FILAS:=PACKCONTROLCUENTA.V_FILAS+1;
  I:=PACKCONTROLCUENTA.V_FILAS;
  PACKCONTROLCUENTA.V_CUENTA(I):=:NEW.NUMCUENTA;
END;
/
/*****************************************************************/
/* DISPARADOR QUE CONTROLA QUE NO SE INSERTEN MAS DE TRES TITULARES*/
/* POR CUENTA                                                    */
/*****************************************************************/
CREATE OR REPLACE TRIGGER CONTROLTITULARES
AFTER INSERT OR UPDATE ON TITULAR
DECLARE
  V_NUMTITULARES NUMBER;
  V_MAXTITULARES CONSTANT NUMBER:=3;
```

```
BEGIN
   FOR I IN 1..PACKCONTROLCUENTA.V_FILAS LOOP
    SELECT COUNT(NUMCLI)
    INTO V_NUMTITULARES
    FROM TITULAR
    WHERE NUMCUENTA=PACKCONTROLCUENTA.V_CUENTA(I);
    IF V_NUMTITULARES>V_MAXTITULARES  THEN
       PACKCONTROLCUENTA.V_FILAS:=0; -- importantísimo
       RAISE_APPLICATION_ERROR(-20000,'PUEDE HABER UN MAXIMO DE 3
TITULARES POR CUENTA');
    END IF;
  END LOOP;
  PACKCONTROLCUENTA.V_FILAS:=0;
END;
/
/*****************************************************************/
```

A partir de la versión 11g, podemos usar los llamados COMPOUND trigger que pueden dispararse en más de un punto en el tiempo.

El código que mostramos a continuación es una reescritura de los disparadores anteriores que controlaban el número de titulares de una cuenta, haciendo uso de los disparadores COMPOUND.

```
create or replace TRIGGER CHECKTITULARES
FOR INSERT OR UPDATE  ON TITULAR
COMPOUND TRIGGER

  TYPE T_CUENTA  IS
  TABLE OF TITULAR.NUMCUENTA%TYPE INDEX BY BINARY_INTEGER;
  V_CUENTA  T_CUENTA ;

  BEFORE EACH ROW IS -- Hace el papel del disparador ANOTARCUENTA
      I BINARY_INTEGER:=V_CUENTA.COUNT;
  BEGIN
    I:=I+1;
    V_CUENTA(I):=:NEW.NUMCUENTA;
    DBMS_OUTPUT.PUT_LINE('INDICE ARRAY V_CUENTA VALE: '||I);
  END BEFORE EACH ROW;

  AFTER STATEMENT IS -- Hace el papel de CONTROLTITULARES
  I BINARY_INTEGER;
  V_NUMTITULARES NUMBER;
  V_MAXTITULARES CONSTANT NUMBER:=3;
  BEGIN
    I:=V_CUENTA.FIRST;
    WHILE I<=V_CUENTA.LAST LOOP
       SELECT COUNT(*)
       INTO V_NUMTITULARES
       FROM TITULAR
       WHERE NUMCUENTA=V_CUENTA(I);
       IF V_NUMTITULARES>V_MAXTITULARES THEN
```

```
            RAISE_APPLICATION_ERROR(-20000,'PUEDE HABER UN MAXIMO DE 3
TITULARES POR CUENTA');
        END IF;
        I:=V_CUENTA.NEXT(I);
    END LOOP;
  END AFTER STATEMENT;
END CHECKTITULARES;
```

El código de este tipo de disparadores es más compacto y no necesitamos el paquete PACKCONTROLCUENTA para crear esa zona común donde guardar las cuentas que están siendo modificadas.
Las variables que se declaran en la parte declarativa del disparador, son accesibles por el código de cada una de las secciones temporales que se pueden declarar en el código del disparador. La siguiente tabla muestra las posibles secciones que se pueden declarar en el cuerpo de un disparador de tipo COMPOUND.

Punto en el tiempo en que se ejecuta	Sección
Antes de que se ejecute la sentencia	BEFORE STATEMENT
Después de que se ejecute la sentencia	AFTER STATEMENT
Antes de que cada fila sea afectada por la sentencia que la orden	BEFORE EACH ROW
Después de que cada fila sea afectada por la sentencia que la orden	AFTER EACH ROW