# Unidad Didáctica 9:

# PROGRAMACIÓN CON PL/SQL

# 1. Introducción

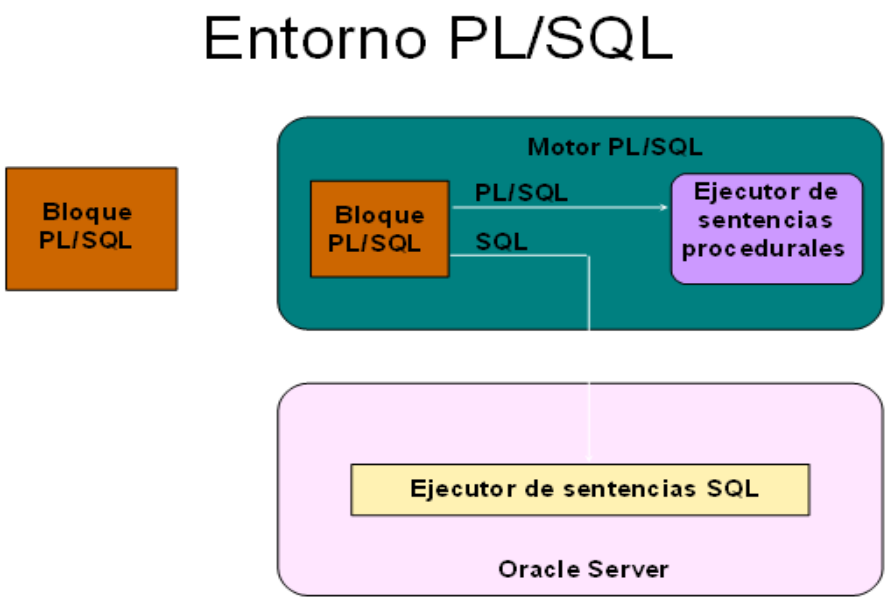
Casi todos los grandes Sistemas Gestores de Datos incorporan utilidades que permiten ampliar el lenguaje SQL para producir pequeñas utilidades que añaden al SQL mejoras de la programación estructurada (bucles, condiciones, funciones,....). La razón es que hay diversas acciones en la base de datos para las que SQL no es suficiente.

Por ello, todas las bases de datos incorporan algún lenguaje de tipo procedimental que permite manipular de forma más avanzada los datos de la base de datos.

**PL/SQL** es el lenguaje procedimental que es implementado por el precompilador de Oracle. Se trata de un lenguaje creado para dar a SQL nuevas posibilidades. Esas posibilidades permiten utilizar condiciones y bucles al estilo de los lenguajes de tercera generación (como Basic, Cobol, C++, Java, etc.).

En otros sistemas gestores de bases de datos existen otros lenguajes procedimentales: SQL Server utiliza Transact SQL, Informix usa Informix 4GL,...

PL/SQL no es un producto de Oracle propiamente dicho. Se trata de una tecnología que emplea Oracle Server y ciertas herramientas de Oracle. Los bloques PL/SQL se transfieren a un motor PL/SQL que los procesa y que puede residir en la herramienta o en Oracle Server.




El motor que se utiliza depende del lugar desde el cual se llama al bloque PL/SQL. Cuando se ejecutan bloques PL/SQL desde un precompilador de Oracle como, por ejemplo, programas Pro*C o Pro*Cobol, SQL*Plus o Server Manager, el motor PL/SQL de Oracle Server los procesa. Se encarga de separar las sentencias SQL y las envía individualmente al ejecutor de sentencias SQL.

Sólo es necesaria una transferencia para enviar el bloque desde la aplicación a Oracle Server, mejorando así el rendimiento, especialmente en las redes cliente-servidor. Además, el código PL/SQL se puede almacenar en Oracle Server como subprogramas a los que puede hacer referencia cualquier número de aplicaciones conectadas a la base de datos.

Muchas herramientas de Oracle, incluyendo **Oracle Developer**, tienen su propio motor PL/SQL, que es independiente del motor de Oracle Server. Este motor filtra las sentencias SQL, las envía individualmente al ejecutor de sentencias SQL de Oracle Server y, después, procesa las sentencias procedimentales restantes en el ejecutor de sentencias procedimentales, que se encuentra en el motor PL/SQL.

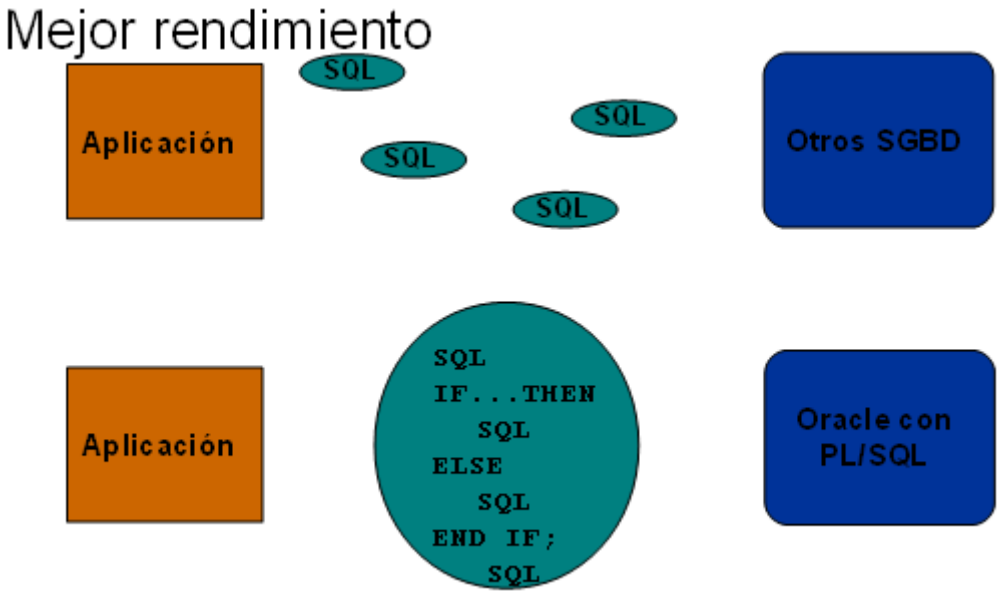
El ejecutor de sentencias procedimentales procesa los datos locales de la aplicación (es decir, los datos que ya se encuentran en el entorno cliente, en lugar de en la base de datos). De esta manera, se reduce el trabajo que se envía a Oracle Server y el número de cursores de memoria necesarios.

# 2. Ventajas de PL/SQL

## Rendimiento:

PL/SQL puede mejorar el rendimiento de una aplicación. Las ventajas varían en función de cada entorno de ejecución.

PL/SQL se puede utilizar para agrupar sentencias SQL en un solo bloque y enviar el bloque completo al servidor con una sola llamada, reduciendo así el tráfico de red. Sin PL/SQL, las sentencias SQL se enviarían a Oracle Server de una en una. Cada sentencia SQL origina una llamada a Oracle Server y, por lo tanto, la sobrecarga de rendimiento es mayor. En un entorno de red, esta sobrecarga puede ser importante. Tal y como ilustra la imagen, si la aplicación utiliza SQL de manera intensiva, se pueden usar subprogramas y bloques PL/SQL para agrupar sentencias SQL antes de enviarlas a Oracle Server para que las ejecute.




PL/SQL también funciona con las herramientas de desarrollo de aplicaciones de Oracle Server como, por ejemplo, Oracle Forms y Oracle Reports. Al aumentar la capacidad de procesamiento procedimental de estas herramientas, PL/SQL mejora el rendimiento.

Nota: Los procedimientos y las funciones que se declaran como parte de una aplicación Oracle Forms o Reports Developer son distintas de las almacenadas en la base de datos, aunque su estructura general sea la misma. Los subprogramas almacenados son objetos de base de datos y están almacenados en el diccionario de datos. Cualquier aplicación puede acceder a ellos, incluidas las aplicaciones Oracle Forms o Reports Developer.

## Portabilidad:

Dado que PL/SQL es un lenguaje nativo de Oracle Server, se pueden desplazar los programas a cualquier entorno host (sistema operativo o plataforma) que sea compatible con Oracle Server y PL/SQL. Dicho de otro modo, los programas PL/SQL funcionan allí donde funcione Oracle Server, por lo que no es necesario adaptarlos a cada nuevo entorno.

## Estructuras de Control de Lenguaje Procedimental:

Las estructuras de control de lenguaje procedimental permiten hacer lo siguiente:

- Ejecutar una secuencia de sentencias de manera condicional.
- Ejecutar una secuencia de sentencias de manera iterativa en un bucle.
- Procesar individualmente las filas que devuelve una consulta de múltiples filas con un cursor explícito.

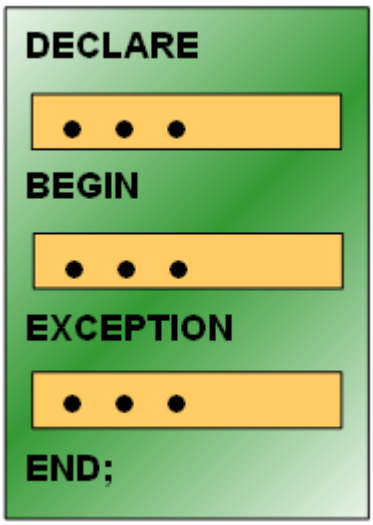
## Errores:

La funcionalidad del manejo de errores de PL/SQL permite hacer lo siguiente:

- Procesar los errores de Oracle Server con rutinas de manejo de excepciones.
- Declarar condiciones de error definidas por el usuario y procesarlas con rutinas de manejo de errores.

# 3. Estructura de Bloques PL/SQL

PL/SQL es un lenguaje estructurado en bloques, lo que significa que los programas se pueden dividir en bloques lógicos. Los bloques PL/SQL constan de hasta tres secciones: l**a sección declarativa** (opcional), **la sección ejecutable** (necesaria) y l**a sección de manejo de excepciones** (opcional).



- **Sección declarativa.** Define e inicializa las variables, constantes, excepciones de usuario y cursores utilizados en el bloque. Va precedida de la palabra **DECLARE .**

- **Sección ejecutable.** Contiene sentencias SQL para manipular datos en la base de datos y sentencias PL/SQL para manipular datos en el bloque. Todas estas sentencias van precedidas por la palabra **BEGIN**.

- **Manejo de excepciones.** Para indicar las acciones a realizar en caso de error. Van precedidas por la palabra **EXCEPTION.**

- **Final del bloque.** La palabra **END** da fin al bloque.

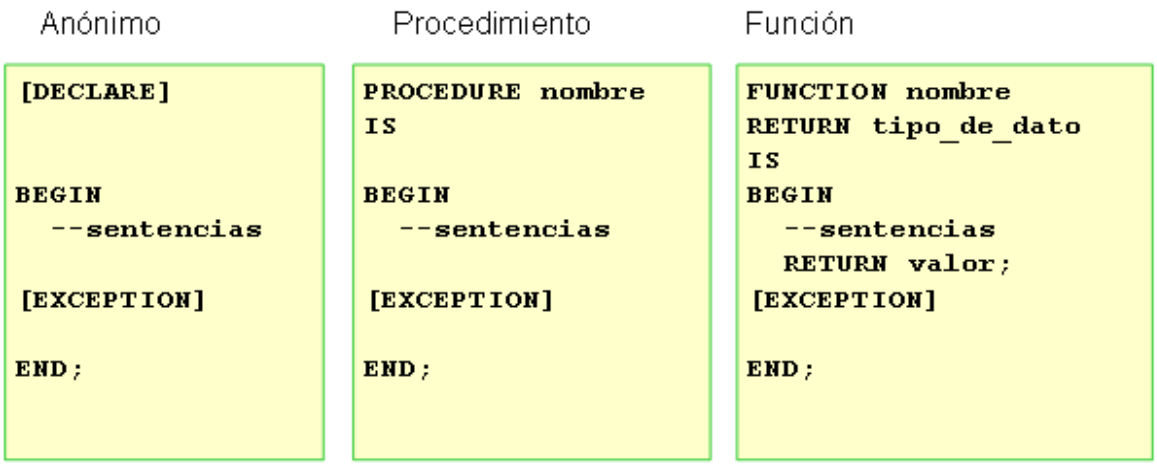
# 4. Normas de escritura de instrucciones PL/SQL

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

- Las palabras clave, nombres de tabla y columna, funciones,… no distinguen entre mayúsculas y minúsculas.
- Colocar un punto y coma (**;**) al final de las sentencias SQL o las sentencias de control PL/SQL.
- Las palabras clave de sección **DECLARE**, **BEGIN** y **EXCEPTION** no van seguidas de punto y coma.
- Las sentencias **END** y el resto de las sentencias PL/SQL necesitan un punto y coma para terminar la sentencia.
- Es posible encadenar sentencias en la misma línea, pero no se recomienda hacerlo por motivos de claridad y de edición.
- Los comentarios varias líneas comienzan con **/*** y terminan con ***/**
- Los comentarios de línea simple utilizan los signos **--** (doble guión). El texto a la derecha de los guiones se considera comentario (el de la izquierda no)

```
DECLARE
   v NUMBER := 17;
BEGIN

/* Este es un comentario que
ocupa varias líneas */

v:=v*2; -- este sólo ocupa esta línea

DBMS_OUTPUT.PUT_LINE(v) -- escribe 34
END;
```

# 5. Tipos de bloques




Los programas PL/SQL constan de uno o varios bloques. Estos bloques pueden estar completamente separados entre sí o anidados unos en otros.

Las unidades básicas (procedimientos y funciones, también denominados subprogramas y bloques anónimos) que conforman un programa PL/SQL son bloques lógicos, que pueden contener cualquier número de subbloques anidados. Por lo tanto, un bloque puede representar una pequeña parte de otro bloque que, a su vez, puede formar parte de una unidad de código completa.

**Bloques Anónimos:**

Los bloques anónimos son bloques sin nombre. Se declaran en el punto de la aplicación en el que tienen que ser ejecutados y se transfieren al motor PL/SQL para que los ejecute en tiempo de ejecución.

**Subprogramas:**

Los subprogramas son bloques PL/SQL con nombre que aceptan parámetros y se pueden llamar. Se pueden declarar como **procedimientos** o como **funciones**. Generalmente, los procedimientos se utilizan para realizar una acción y las funciones, para calcular un valor.

Los subprogramas se pueden almacenar en el nivel de aplicación o de servidor. Con los componentes de Oracle Developer (Forms, Reports y Graphics), los procedimientos y las funciones se pueden declarar como parte de la aplicación (una pantalla o un informe) y llamarlos desde otros procedimientos, funciones y disparadores dentro de la misma aplicación siempre que sea necesario.

Las funciones son similares a los procedimientos, excepto en que las funciones deben devolver un valor.

# 6. Uso de variables

Con PL/SQL, es posible declarar variables y utilizarlas en sentencias procedimentales y SQL en cualquier lugar en el que se pueda utilizar una expresión. Las variables se utilizan para lo siguiente:

- Almacenamiento temporal de los datos: Los datos se pueden almacenar temporalmente en una o varias variables para utilizarlos cuando se valide la entrada de datos y para procesarlos posteriormente durante el flujo de datos.
- Manipulación de valores almacenados: Las variables pueden se pueden utilizar para realizar cálculos y otras manipulaciones de datos sin acceder a la base de datos.
- Reutilización: Después de ser declaradas, las variables se pueden utilizar repetidas veces en una aplicación simplemente haciendo referencia a ellas en otras sentencias, incluidas otras sentencias declarativas.

## 6.1. Declaración e inicialización de variables

Se puede declarar las variables en la parte declarativa de cualquier bloque, subprograma o paquete PL/SQL. Las declaraciones asignan espacio de almacenamiento para un valor, especifican su tipo de dato y dan un nombre a la ubicación de almacenamiento para que se le pueda hacer referencia. Las declaraciones también pueden asignar un valor inicial e imponer la restricción NOT NULL (no nulo) a la variable. Las referencias por adelantado no están permitidas. Debe declarar una variable antes de hacer referencia a ella en otras sentencias, incluidas otras sentencias declarativas.

Sintaxis:

```
identificador [CONSTANT] tipo_de_dato [NOT NULL]
[:= | DEFAULT expr];
```

Donde:

| | |
|---|---|
| ***identificador*** | es el nombre de la variable. |
| ***CONSTANT*** | restringe la variable para que su valor no pueda cambiar; las constantes deben inicializarse. |
| ***tipo_de_dato*** | es un tipo de dato escalar, compuesto, de referencia o LOB. |
| ***NOT NULL*** | restringe la variable para que obligatoriamente contenga un valor (las variables `NOT NULL` deben inicializarse.) . |
| ***expr*** | es cualquier expresión PL/SQL que pueda ser una expresión literal, otra variable o una expresión que implique el uso de operadores y funciones. |

**Ejemplo:**

```
DECLARE
 v_hiredate DATE;
 v_deptno NUMBER(2) NOT NULL := 10;
 v_location VARCHAR2(13) := 'Atlanta';
 c_comm CONSTANT NUMBER := 1400;
 v_mgr NUMBER(6) DEFAULT 100;
```

## 6.2. Instrucciones para declarar variables PL/SQL

He aquí algunas de las instrucciones que debe seguir para declarar variables PL/SQL:

- Asigne un nombre al identificador utilizando las mismas reglas que se aplican a los objetos SQL.
- Puede utilizar reglas de nomenclatura como, por ejemplo, ***v_nombre*** para representar una variable y ***c_nombre*** para representar una constante.
- Si utiliza la restricción `NOT NULL`, debe asignar un valor.
- Si declara un solo identificador por línea, el código será más fácil de leer y de mantener.
- En las declaraciones de constantes, la palabra clave **`CONSTANT`** debe preceder al tipo. La siguiente declaración define una constante del subtipo `REAL` de `NUMBER` y le asigna el valor de 50000. Las constantes se deben inicializar en su declaración; de lo contrario, se producirá un error de compilación al elaborar (compilar) la declaración.

```
v_sal  CONSTANT REAL := 50000.00;
```

- Inicialice la variable a una expresión con el operador de asignación (**:=**) o con la palabra reservada **`DEFAULT`**. Si no le asigna un valor inicial, la nueva variable contendrá el valor `NULL` por defecto hasta que le asigne un valor posteriormente. Para asignar o volver a asignar un valor a una variable, hay que escribir una sentencia de asignación PL/SQL. Para que reciba un nuevo valor, debe nombrar explícitamente la variable a la izquierda del operador de asignación (**:=**). La inicialización de todas las variables es una buena práctica de programación.
- Dos objetos pueden tener el mismo nombre, siempre y cuando estén definidos en bloques diferentes. Si coexisten, sólo se puede utilizar el objeto declarado en el bloque actual.
- El nombre (identificador) de una variable no debería ser el mismo que el nombre de las columnas de la tabla utilizadas en el bloque. Si en las sentencias SQL hay

variables PL/SQL que tienen el mismo nombre que una columna, Oracle Server asume que se está haciendo referencia a la columna.

- Los nombres de las variables no deben tener más de 30 caracteres. El primer carácter debe ser una letra y el resto pueden ser letras, números o símbolos especiales. Hay otros caracteres como, por ejemplo, los guiones, las barras inclinadas y los espacios, que no son válidos.

## 6.3. Tipos de Variables


Esta imagen ilustra los siguientes tipos de dato de variables:

- TRUE representa un valor booleano.
- 25-JAN-01 representa un tipo de dato DATE.
- La fotografía representa un tipo de dato BLOB.
- El texto del discurso representa un tipo de dato LONG.
- 256120.08 representa un tipo de dato NUMBER con precisión y escala.
- La película representa un tipo de dato BFILE.
- El nombre de la ciudad, Atlanta, representa un tipo de dato VARCHAR2.

Todas las constantes, variables y parámetros tienen un tipo de dato que especifica un formato de almacenamiento, restricciones y un rango de valores válido.

El **tipo de dato escalar** guarda un solo valor y no tiene componentes internos. Los tipos de dato escalares se pueden clasificar en cuatro categorías: número, carácter, fecha y booleano. Los tipos de dato de carácter y número tienen subtipos que asocian un tipo base a una restricción. Por ejemplo, `INTEGER` y `POSITIVE` son subtipos del tipo base `NUMBER`.

| tipo de datos | descripción |
|---|---|
| CHAR(n) | Texto de anchura fija |
| VARCHAR2(n) | Texto de anchura variable |
| NUMBER[(p[,s])] | Número. Opcionalmente puede indicar el tamaño del número (*p*) y el número de decimales (s) |
| DATE | Almacena fechas |
| TIMESTAMP | Almacena fecha y hora |
| INTERVAL YEAR TO MONTH | Almacena intervalos de años y meses |
| INTERVAL DAY TO SECOND | Almacena intervalos de días, horas, minutos y segundos |
| LONG | Para textos de más de 32767 caracteres |
| LONG RAW | Para datos binarios. PL/SQL no puede mostrar estos datos directamente |
| INTEGER | Enteros de -32768 a 32767 |
| BINARY_INTEGER | Enteros largos (de -2.147.483.647 a -2.147.483.648) |
| PLS_INTEGER | Igual que el anterior pero ocupa menos espacio |
| BOOLEAN | Permite almacenar los valores TRUE (verdadero) y FALSE (falso) |
| BINARY_DOUBLE | Disponible desde la versión 10g, formato equivalente al double del lenguaje C. Representa números decimales en coma flotante. |
| BINARY_FLOAT | Otro tipo añadido en la versión 10g, equivalente al float del lenguaje C. |

**Ejemplos:**

```
DECLARE
 v_job VARCHAR2(9);

/*variable para almacenar el puesto de un empleado en la
empresa */

v_count BINARY_INTEGER := 0;

/*variable que cuenta las iteraciones de un bucle y que se
inicializa en 0*/

 v_total_sal NUMBER(9,2) := 0;

/*variable que suma el total de los sueldos de un departamento
y que se inicializa en 0 */
 v_orderdate DATE := SYSDATE + 7;

/*variable que almacena la fecha de envío de un pedido  y se
inicializa a una semana del día actual */
```

```
 c_tax_rate CONSTANT NUMBER(3,2) := 8.25;

/* una variable constante para el tipo de impuestos, que no
cambia nunca en el bloque PL/SQL */

 v_valid BOOLEAN NOT NULL := TRUE;

/*indicador que señala si un dato es válido o no y que se
inicializa en TRUE*/
```

## 6.4. Atributo %TYPE

Cuando se declaran variables PL/SQL para guardar valores de columnas, debemos asegurarnos de que la variable es del tipo de dato y precisión correctos. Si no es así, se producirá un error PL/SQL durante la ejecución.

En lugar de codificar el tipo de dato y la precisión de una variable, podemos utilizar el atributo **%TYPE** para declarar la variable de acuerdo con otra variable que se haya declarado anteriormente, o en función de una columna de base de datos. El atributo %TYPE se suele utilizar cuando el valor almacenado en la variable se va a extraer de una tabla de base de datos. Para utilizar el atributo en lugar del tipo de dato que es necesario en la declaración de la variable, tenemos que prefíjar con el nombre de la tabla y la columna de base de datos. Si estamos haciendo referencia a una variable declarada anteriormente, prefijar el nombre de la variable al atributo.

PL/SQL determina el tipo de dato y el tamaño de la variable al compilar el bloque, de manera que dichas variables sean siempre compatibles con la columna que se utiliza para rellenarla. Esta ventaja es muy importante a la hora de escribir y mantener el código, ya que no es necesario preocuparse por los cambios de tipos de dato de las columnas que se realizan en la base de datos. También puede declarar una variable de acuerdo con otra variable declarada anteriormente prefijando el atributo con el nombre de la variable.

**Ejemplos:**

```
Declare variables para almacenar el apellido de un empleado.
La variable v_nombre se define con el mismo tipo de dato que
la columna APELLIDO de la tabla EMPLE. %TYPE proporciona el
tipo de dato de una columna de base de datos:
         v_nombre EMPLE.APELLIDO%TYPE;

Declare variables para almacenar el saldo de una cuenta
bancaria y su saldo mínimo, que se inicia como 10. La variable
v_min_balance se define con el mismo tipo de dato que la
variable v_balance. %TYPE proporciona el tipo de dato de una
variable:

   v_balance NUMBER(7,2);
   v_min_balance v_balance%TYPE := 10;
```

> La restricción NOT NULL para columnas de bases de datos no se aplica a las variables que se declaran con %TYPE. Por lo tanto, si declara una variable con el atributo %TYPE, que utiliza una columna de base de datos definida como NOT NULL, es posible asignar un valor NULL a la variable.

## 6.5. DBMS_OUTPUT.PUT_LINE

Una opción para ver la información de un bloque PL/SQL es **DBMS_OUTPUT.PUT_LINE**. DBMS_OUTPUT es un paquete de Oracle y PUT_LINE es una función contenida en ese paquete.

Podemos hacer referencia a DBMS_OUTPUT.PUT_LINE dentro de un bloque PL/SQL y, entre paréntesis, especificar la cadena que deseamos imprimir en pantalla. Primero, hay que hacer visible el buffer que utiliza el paquete en la sesión de SQL*Plus. Para ello, ejecutamos el comando

**SET SERVEROUTPUT ON** de SQL*Plus.

**Ejemplo:**

```
DECLARE
   a NUMBER := 17;
BEGIN
   DBMS_OUTPUT.PUT_LINE(a);
END;
```

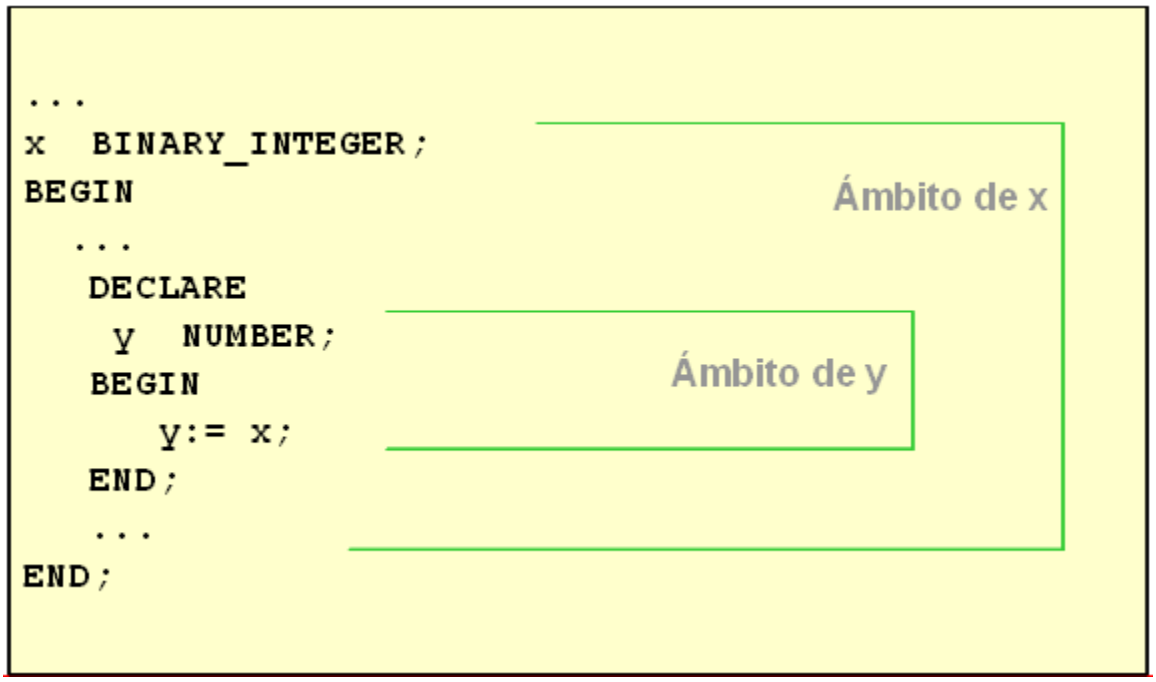
Eso escribiría el número 17 en la pantalla.

## 6.6. Ámbito de las variables

Una de las ventajas de PL/SQL con respecto a SQL es su capacidad de anidar sentencias. Es posible anidar bloques en los lugares en los que se permita una sentencia ejecutable, con lo que se convierte el bloque anidado en una sentencia. Por lo tanto, se puede dividir la parte ejecutable de un bloque en bloques de menor tamaño. La sección de excepciones también puede contener bloques anidados.

Las referencias a un identificador se resuelven de acuerdo con su ámbito y su visibilidad. El ámbito de un identificador es la región de una unidad de programa (bloque, subprograma o paquete) desde la cual se puede hacer referencia al identificador. Los identificadores sólo son visibles en las regiones desde las cuales se puede hacer referencia a ellos utilizando un nombre no cualificado. Los identificadores que se declaran en un bloque PL/SQL se consideran **locales** para ese bloque y **globales** para todos sus subbloques. Si se vuelve a declarar un identificador global en un subbloque, ambos identificadores permanecen en el ámbito. En el subbloque, no obstante, sólo es visible el identificador local, porque hay que utilizar un nombre cualificado para hacer referencia al identificador global.

Aunque no es posible declarar dos veces un identificador en el mismo bloque, se puede declarar el mismo identificador en dos bloques diferentes. Los dos elementos representados por el identificador son distintos y los cambios realizados en uno no afectan al otro. Sin embargo, un bloque no puede hacer referencia a identificadores declarados en otros bloques en el mismo nivel, porque esos identificadores no son locales ni globales para el bloque.




```
DECLARE
   v NUMBER := 2;
BEGIN
   v:=v*2;
    DECLARE
       z NUMBER := 3;
    BEGIN
        z:=v*3;
        DBMS_OUTPUT.PUT_LINE(z); --escribe 12
        DBMS_OUTPUT.PUT_LINE(v); --escribe 4
    END;
    DBMS_OUTPUT.PUT_LINE(v*2); --escribe 8
    DBMS_OUTPUT.PUT_LINE(z); --error
END;
```

# 7. Operadores de PL/SQL

En PL/SQL se permiten utilizar todos los operadores de SQL: los operadores aritméticos (+ - * /), condicionales (> < != <> >= <= OR AND NOT) y de cadena (||).

A estos operadores, PL/SQL añade el operador de potencia **.

Las operaciones de una expresión se realizan en un orden en particular que depende de su precedencia (prioridad). La siguiente tabla muestra el orden que tienen por defecto las operaciones, desde la prioridad más alta hasta la más baja.

| Operador | Operación |
|---|---|
| ** | Exponenciación |
| +, - | Identidad, negación |
| *, / | Multiplicación, división |
| +, -, \|\| | Suma, resta, concatenación |
| =, <, >, <=, >=, <>, !=, ~=, ^=, IS NULL, LIKE, BETWEEN, IN | Comparación |
| NOT | Negación lógica |
| AND | Conjunción |
| OR | Inclusión |

```
Ejemplos:

Incrementar el contador para un bucle:

   v_cont := v_count + 1;

Especificar el valor de una variable booleana:

   v_equal := (v_n1=v_n2);

Validar si el número de un empleado contiene un valor:

   v_valid := (v_empno IS NOT NULL);
```

# 8. Funciones en PL/SQL

La mayoría de las funciones disponibles en SQL también son válidas en las expresiones PL/SQL. Hay que tener en cuenta que las funciones de grupo: AVG, MIN, MAX, COUNT, SUM, STDDEV y VARIANCE se aplican a los grupos de filas de una tabla y, por lo tanto, sólo están disponibles en las sentencias SQL de un bloque PL/SQL.

# 9. Sentencias SQL en PL/SQL

Para extraer información desde la base de datos o aplicar cambios en ella, hay que utilizar el lenguaje SQL. El lenguaje PL/SQL es compatible con el lenguaje de manipulación de datos y con los comandos de control de transacciones de SQL. **Se pueden utilizar sentencias SELECT** para rellenar las variables con los valores que se han consultado en una fila de una tabla. **Se pueden utilizar comandos DML** para modificar los datos de una tabla de base de datos. Sin embargo, no se olvide de seguir estas reglas de los bloques PL/SQL al utilizar las sentencias DML **y los comandos de control de transacciones** en los bloques PL/SQL.

La palabra clave END señala el final de un bloque PL/SQL, no el final de una transacción. Al igual que un bloque puede abarcar múltiples transacciones, una transacción puede abarcar múltiples bloques.

PL/SQL no admite directamente sentencias DDL (Lenguaje de Definición de Datos) como, por ejemplo, CREATE TABLE, ALTER TABLE o DROP TABLE.

PL/SQL no admite sentencias DCL (Lenguaje de Control de Datos) como, por ejemplo, GRANT o REVOKE.

Siempre que se emite una sentencia SQL, Oracle abre un área de memoria en la que se analiza y se ejecuta el comando. Esta área se denomina cursor. Por tanto, un cursor es un área de trabajo SQL privada. Existen dos tipos de cursores:

- Cursores implícitos: Declarados para todas las sentencias DML y SELECT de PL/SQL
- Cursores explícitos: El programador los declara y les asigna un nombre. Los veremos mas adelante.

Los cursores SQL poseen una serie de atributos. Los atributos de los cursores implícitos, que pueden utilizarse dentro de los bloques PL/SQL para probar el resultado de las sentencias SQL, son:

- **SQL%ROWCOUNT** Número de filas afectadas por la sentencia SQL más reciente (un valor entero)
- **SQL%FOUND** Atributo booleano que se evalúa como TRUE si la sentencia SQL más reciente afecta a una o varias filas
- **SQL%NOTFOUND** Atributo booleano que se evalúa como TRUE si la sentencia SQL más reciente no afecta a ninguna fila

- **SQL%ISOPEN** Siempre se evalúa como FALSE porque PL/SQL cierra los cursores implícitos inmediatamente después de ejecutarlos

## 9.1. Sentencias SELECT en PL/SQL

Utilice la sentencia SELECT para recuperar datos de la base de datos. En la sintaxis:

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

- *lista_select* es una lista de al menos una columna que puede incluir expresiones SQL, funciones de filas o funciones de grupos.
- *nombre_variable* es la variable escalar que guarda el valor recuperado.
- *nombre_registro* es el registro (RECORD) PL/SQL que guarda los valores recuperados.
- *tabla* especifica el nombre de la tabla de base de datos.
- *condición* se compone de nombres de columnas, expresiones, constantes y operadores de comparación, incluidos variables y constantes PL/SQL.

Hay que tener en cuenta:

- Cada sentencia SQL termina con un punto y coma (;).
- Es necesario utilizar la cláusula **INTO** para la sentencia `SELECT` cuando está embebida en PL/SQL.
- La cláusula `WHERE` es opcional y se puede utilizar para especificar variables de entrada, constantes, literales o expresiones PL/SQL.
- Especifique el mismo número de variables en la cláusula `INTO` que columnas de base de datos de la cláusula `SELECT`. Asegúrese de que sus posiciones se corresponden y de que sus tipos de dato son compatibles.
- Utilice funciones de grupos como, por ejemplo, `SUM`, en las sentencias SQL, porque este tipo de funciones se aplica a los grupos de filas de una tabla.
- Las consultas deben devolver sólo una fila. Si las consultas devuelven más de una fila o ninguna, se generará un error.

- Para gestionar estos errores, PL/SQL emite excepciones estándar, que se pueden interrumpir en la sección de excepciones del bloque con las excepciones NO_DATA_FOUND y TOO_MANY_ROWS .

**Ejemplo:**

```
DECLARE
   v_salario empleados.salario%TYPE;
   v_nombre empleados.nombre%TYPE;
BEGIN
   SELECT salario,nombre INTO v_salario, v_nombre
   FROM empleados
   WHERE id_empleado=12344;

DBMS_OUTPUT.PUT_LINE('El nuevo salario será de ' ||
vsalario*1.2 || 'euros');
END;
```

## 9.2. Manipulación de Datos con PL/SQL

Los datos de la base de datos se manipulan con los comandos DML. En PL/SQL, se pueden emitir los comandos DML **INSERT**, **UPDATE** y **DELETE** sin ninguna restricción. Para liberar los bloqueos de filas (y de tablas), hay que incluir sentencias COMMIT o ROLLBACK en el código PL/SQL.

**Ejemplos:**

```
Agregue la información de un nuevo empleado a la tabla EMPLE.

BEGIN

INSERT INTO emple(emp_no, apellido, oficio,fecha_alt, salario)
VALUES (12345, 'Cores', 'vendedor', sysdate, 4000);

END;



Aumente el sueldo a todos los empleados que tengan el puesto
de administrativos.

DECLARE
  v_sal_incr   emple.salario%TYPE := 800;
BEGIN
  UPDATE emple
  SET salario = salario + v_sal_incr
  WHERE  oficio = 'administrativo';
```

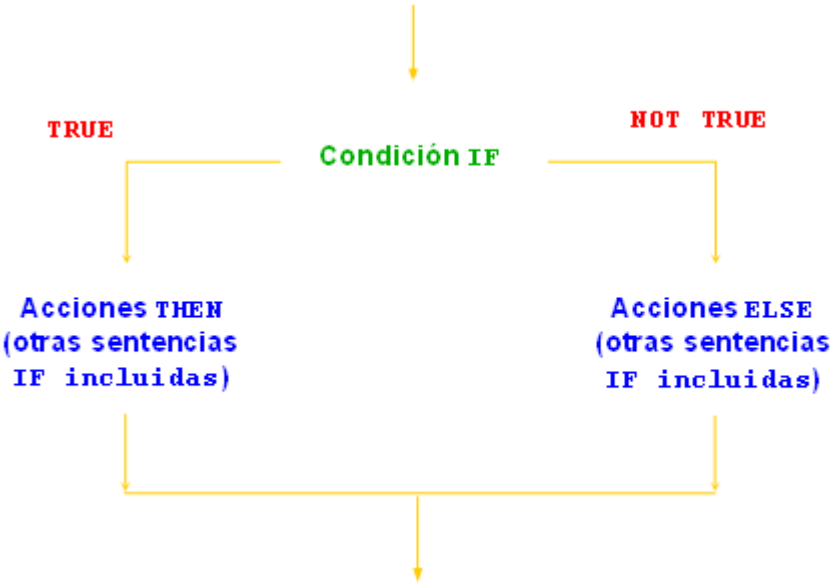
```
END;


Suprima los empleados  que pertenecen al departamento 10.

DECLARE
  v_deptno    emple.dept_no%TYPE := 10;
BEGIN
  DELETE FROM    emple
  WHERE          dept_no = v_deptno;
END;
```

# 10. Estructuras de control

Para cambiar el flujo lógico de las sentencias en el interior de los bloques PL/SQL, podemos utilizar una serie de *estructuras de control*. Esta lección explica tres tipos de estructuras de control PL/SQL: **construcciones condicionales** con la sentencia **`IF`**, **expresiones `CASE`** y **estructuras de control `LOOP`.**

## 10.1. Sentencias IF

La estructura de la sentencia `IF` de PL/SQL es similar a la estructura de las sentencias `IF` de otros lenguajes procedimentales. Permite a PL/SQL realizar acciones selectivamente basándose en condiciones.



```
IF condición THEN
        sentencias;
[ELSIF condición THEN
        sentencias;]
[ELSE
        sentencias;]
END IF;
```

En la sintaxis:

- condición es una variable o expresión booleana (`TRUE, FALSE o NULL`).
- THEN es una cláusula que asocia la expresión booleana que la precede con la secuencia de sentencias que la sigue.
- sentencias pueden ser una o varias sentencias PL/SQL o SQL. (Pueden incluir otras sentencias `IF` que contienen varias sentencias `IF, ELSE y ELSIF` anidadas.)
- ELSIF es una palabra clave que introduce una expresión booleana. (Si la primera condición devuelve `FALSE o NULL`, la palabra clave `ELSIF` introduce condiciones adicionales.)
- ELSE es una palabra clave que ejecuta la secuencia de sentencias que le sigue si el control llega hasta ella.

**Ejemplos:**

Si el apellido es Vargas defina el oficio como ANALISTA y el número del departamento como 80

```
IF apellido  = 'Vargas' THEN
   v_oficio:= 'ANALISTA';
   v_deptno:= 80;
END IF;
```

Si el apellido es Vargas y el sueldo es mayor de 6.500 definir el número del departamento como 60.

```
IF apellido = 'Vargas' AND salario > 6.500 THEN
   v_deptno := 60;
END IF;
```

Defina un indicador booleano como TRUE si la fecha de contratación es mayor de cinco años; de lo contrario, defina el indicador booleano como FALSE.

```
DECLARE
   v_fecha_alta  DATE := '12/12/90';
   v_five_years BOOLEAN;
BEGIN
. . .
IF MONTHS_BETWEEN(SYSDATE,v_fecha_alta)/12 > 5 THEN
   v_five_years := TRUE;
ELSE
   v_five_years := FALSE;
END IF;
```

Determine la bonificación de un empleado en base al departamento al que pertenece.

```
    IF v_deptno = 10 THEN
      v_bonus := 5000;
    ELSIF v_deptno = 80 THEN
      v_bonus := 7500;
    ELSE
      v_bonus := 2000;
    END IF;
```

Nota: En caso de que haya múltiples sentencias IF-ELSIF, sólo se procesará la primera sentencia verdadera. La cláusula ELSE final es opcional.

## 10.2. Expresiones CASE

La expresión **CASE** selecciona un resultado y lo devuelve. Para seleccionar el resultado, la expresión CASE utiliza un selector, que es una expresión cuyo valor sirve para seleccionar una de varias alternativas. El selector va seguido de una o varias cláusulas WHEN, que se comprueban secuencialmente. El valor del selector determina qué cláusula se ejecuta. Si el valor del selector es igual al valor de una expresión de cláusula WHEN, se ejecuta esa cláusula WHEN.

```
CASE selector
   WHEN condición1_búsqueda THEN resultado1;
   WHEN condición2_búsqueda THEN resultado2;
   ...
   WHEN condiciónN_búsqueda THEN resultadoN;
  [ELSE resultadoN+1;]
END CASE;
```

**Ejemplos:**

```
texto:= CASE actitud
          WHEN 'A' THEN 'Muy buena';
          WHEN 'B' THEN 'Buena';
          WHEN 'C' THEN 'Normal';
          WHEN 'D' THEN 'Mala';
          ELSE 'Desconocida';
        END CASE;


aprobado:= CASE
              WHEN actitud='A' AND nota>=4 THEN TRUE;
              WHEN  nota>=5  AND  (actitud='B'  OR  actitud='C')
              THEN TRUE;
              WHEN nota>=7 THEN TRUE;
              ELSE FALSE
           END CASE;
```

En este segundo ejemplo la expresión CASE de búsqueda no tiene selector. Además, las cláusulas WHEN contienen condiciones de búsqueda que devuelven un valor booleano, no expresiones que pueden devolver un valor de cualquier tipo.

## 10.3. Control Iterativo: Sentencias LOOP

PL/SQL ofrece una serie de facilidades para estructurar bucles que repiten múltiples veces una sentencia o una secuencia de sentencias.

PL/SQL proporciona los siguientes tipos de bucles:

- Un bucle básico que realiza acciones repetitivas sin condiciones globales.
- Bucles **FOR** que realizan controles iterativos de acciones basándose en un contador.
- Bucles **WHILE** que realizan controles iterativos de acciones basándose en una condición.

Utilizar la sentencia **EXIT** para finalizar los bucles.

### Bucles Básicos

El tipo más sencillo de sentencia LOOP es el bucle básico (o infinito), que encierra una secuencia de sentencias entre las palabras clave **LOOP** y **END LOOP**. Cada vez que el flujo de

ejecución llega a la sentencia `END LOOP`, el control se devuelve a la sentencia `LOOP` correspondiente por encima de ella. Un bucle básico permite la ejecución de su sentencia al menos una vez, aunque la condición ya se haya cumplido antes de entrar en el bucle. Sin la sentencia `EXIT`, el bucle sería infinito.

La Sentencia EXIT

Utilice la sentencia **EXIT** para finalizar los bucles. El control pasa a la sentencia que hay después de la sentencia `END LOOP`. Puede emitir EXIT como una acción en el interior de una sentencia `IF`, o como una sentencia autónoma en el interior del bucle. La sentencia EXIT se debe colocar en el interior de un bucle. En este caso, puede añadir una cláusula **WHEN** para permitir que el bucle termine de manera condicional. Cuando se encuentra con la sentencia `EXIT`, la condición de la cláusula `WHEN` se evalúa. Si esta condición devuelve `TRUE`, el bucle finaliza y el control pasa a la siguiente sentencia que hay después del bucle. Un bucle básico puede contener múltiples sentencias `EXIT`.

```
LOOP
   sentencia1;
. . .
   EXIT [WHEN condición];
END LOOP;
```

donde *condición* es una expresión o variable booleana (`TRUE, FALSE o NULL`);

**Ejemplo:**

Insertar tres nuevos identificadores de lugares para el código de país de CA y la ciudad de Montreal. Estos tres nuevos identificadores tienen que ser consecuativos al lugar de id más alto.

```
DECLARE
 v_pais_id   lugares.pais_id%TYPE := 'CA';
 v_lugar_id  lugares.lugar_id%TYPE;
 v_cont NUMBER(2) := 1;
 v_ciudad lugares.ciudad%TYPE := 'Montreal';

BEGIN
  SELECT MAX(lugar_id) INTO v_lugar_id FROM lugares
  WHERE ciudad_id = v_ciudad_id;

  LOOP
   INSERT INTO lugares(lugar_id, ciudad, pais_id)
   VALUES((v_lugar_id + v_cont),v_ciudad, v_pais_id);
   v_cont := v_cont + 1;
  EXIT WHEN v_cont > 3;
  END LOOP;

END;
```

Nota: Los bucles básicos permiten ejecutar sus sentencias al menos una vez, aunque la condición ya se haya cumplido al entrar en el bucle, siempre y cuando la condición se coloque en el bucle, para que no se compruebe hasta después de estas sentencias. Sin embargo, si la condición de salida se sitúa en la parte superior del bucle, antes de cualquiera de las otras sentencias ejecutables, y esa condición es verdadera, el bucle finalizará y las sentencias nunca se ejecutarán.

## Bucles WHILE

El bucle **WHILE** sirve para repetir una secuencia de sentencias hasta que la condición de control ya no sea TRUE. La condición se evalúa al principio de cada iteración. El bucle termina cuando la condición es FALSE. Si la condición es FALSE al principio del bucle, no se realiza ninguna iteración más.

```
WHILE condición LOOP
 sentencia1;
 sentencia2;
 . . .
END LOOP;
```

En la sintaxis:

- *condición* es una variable o expresión booleana (TRUE, FALSE o NULL).
- *sentencia* puede ser una o varias sentencias PL/SQL o SQL.

Si las variables que se utilizan en las condiciones no cambian a lo largo del cuerpo del bucle, la condición sigue siendo TRUE y el bucle no termina.

**Nota:** Si la condición devuelve NULL, se ignora el bucle y el control pasa a la siguiente sentencia.

**Ejemplo:**

Insertar tres nuevos identificadores de lugares para el código de país de CA y la ciudad de Montreal. Estos tres nuevos identificadores tienen que ser consecutivos al lugar de id más alto.

```
DECLARE
 v_pais_id   lugares.pais_id%TYPE := 'CA';
 v_lugar_id  lugares.lugar_id%TYPE;
 v_cont NUMBER(1) := 1;
 v_ciudad lugares.ciudad%TYPE := 'Montreal';
```

```
BEGIN
  SELECT MAX(lugar_id) INTO v_lugar_id FROM lugares
  WHERE ciudad_id = v_ciudad_id;

  WHILE v_cont<=3 LOOP
   INSERT INTO lugares(lugar_id, ciudad, pais_id)
   VALUES((v_lugar_id + v_cont),v_ciudad, v_pais_id);
   v_cont := v_cont + 1;
  END LOOP;
END;
```

## Bucles FOR

Los bucles **FOR** tienen la misma estructura general que los bucles básicos. Pero, además, poseen una sentencia de control antes de la palabra clave LOOP para determinar el número de iteraciones que realiza PL/SQL. En la sintaxis:

```
FOR contador IN [REVERSE] límite_inferior .. límite_superior LOOP
      sentencia1;
      sentencia2;
      . . .
END LOOP;
```

donde:

- contador — es un número entero declarado implícitamente cuyo valor aumenta o disminuye automáticamente (disminuye si se utiliza la palabra clave **REVERSE**) por 1 en cada iteración del bucle hasta que se llega al límite inferior o superior.
- REVERSE — hace que el contador disminuya con cada iteración desde el límite superior hasta el límite inferior. (Tenga en cuenta que el límite inferior se sigue referenciando en primer lugar).
- *límite_inferior* — especifica el límite inferior del rango de valores del contador.
- *límite_superior* — especifica el límite superior del rango de valores del contador.

No declare el contador, ya que se declara implícitamente como un entero.

**Nota:** La secuencia de sentencias se ejecuta cada vez que aumenta el contador, tal y como determinan los dos límites. El límite inferior y el límite superior del rango del bucle pueden ser literales, variables o expresiones, pero se deben evaluar como enteros.

```
DECLARE
 v_lower NUMBER := 1;
 v_upper NUMBER := 100;

BEGIN
 FOR i IN v_lower..v_upper LOOP
 ...
 END LOOP;
END;
```

El límite inferior y el límite superior están incluidos en el rango del bucle. Si el límite inferior del rango del bucle se evalúa como un entero mayor que el límite superior, la secuencia de sentencias no se ejecutará, siempre y cuando no se haya utilizado REVERSE.

Por ejemplo, la siguiente sentencia sólo se ejecuta una vez:

```
FOR i IN 3..3 LOOP
      sentencia1;
END LOOP;
```

**Ejemplo:**

Insertar tres nuevos identificadores de lugares para el código de país de CA y la ciudad de Montreal. Estos tres nuevos identificadores tienen que ser consecutivos al lugar de id más alto.

```
DECLARE
 v_pais_id   lugares.pais_id%TYPE := 'CA';
 v_lugar_id  lugares.lugar_id%TYPE;
 v_ciudad lugares.ciudad%TYPE := 'Montreal';

BEGIN
  SELECT MAX(lugar_id) INTO v_lugar_id
  FROM lugares
  WHERE ciudad_id = v_ciudad_id;

  FOR i IN 1..3 LOOP
     INSERT INTO lugares(lugar_id, ciudad, pais_id)
     VALUES((v_lugar_id + v_cont),v_ciudad, v_pais_id);
  END LOOP;
END;
```

# 11. Creación de procedimientos

Un procedimiento es un bloque PL/SQL denominado que puede aceptar parámetros (que en ocasiones se denominan argumentos) y que se puede llamar. En general, los procedimientos sirven para realizar una acción. Los procedimientos poseen una cabecera, una sección de declaración, una sección ejecutable y una sección de manejo de excepciones opcional.

Los procedimientos se pueden compilar y almacenar en la base de datos como objetos de esquema.

Los procedimientos mejoran la capacidad de reutilización y de mantenimiento. Una vez validados, se pueden utilizar en cualquier número de aplicaciones. Si los requisitos cambian, sólo hay que actualizar el procedimiento.

```
CREATE [OR REPLACE] PROCEDURE nombre_procedimiento
      [(parametro1 [modo1] tipodato1,
        parametro2 [modo2] tipodao2,
        . . .)]
IS|AS
  Bloque PL/SQL;
```

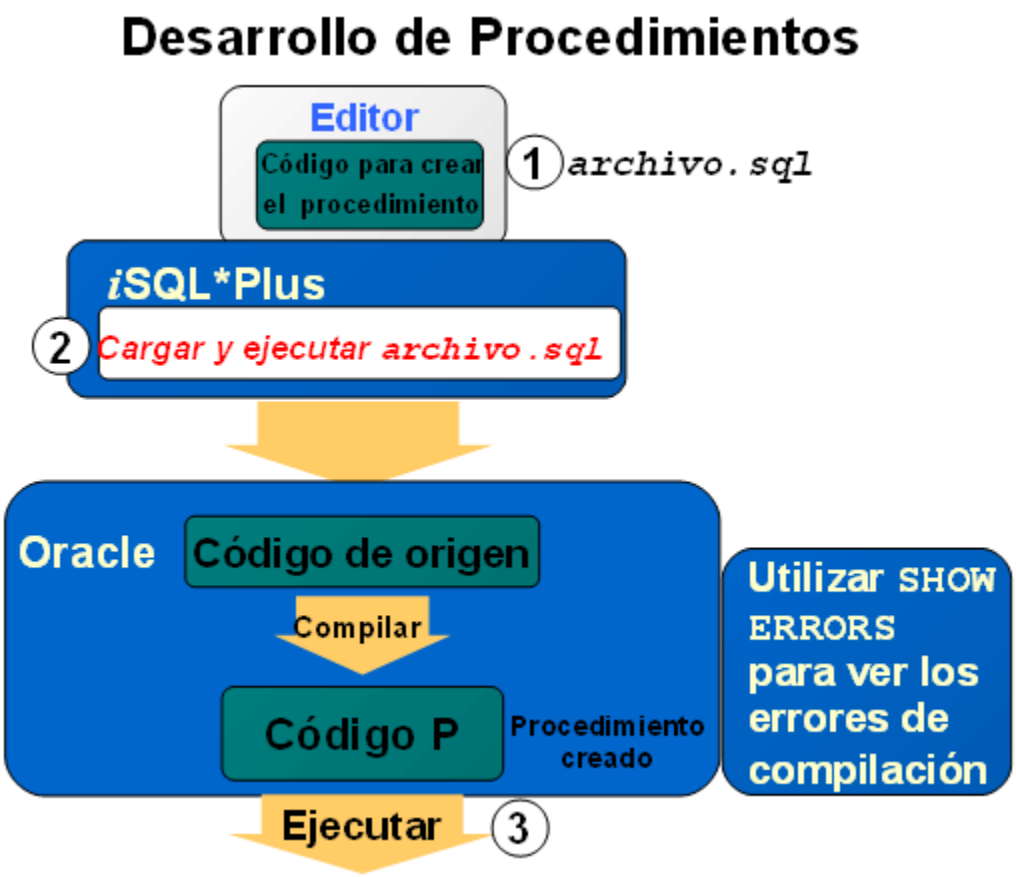
| Parámetro | Descripción |
|---|---|
| *nombre_procedimiento* | Nombre del procedimiento |
| *parámetro* | Nombre de una variable PL/SQL cuyo valor se rellena en el entorno de llamada o se transfiere a él, o ambas cosas, dependiendo del *modo* que se utilice |
| *modo* | Tipo de argumento:<br>IN (por defecto)<br>OUT<br>IN OUT |
| *Tipo de dato* | Tipo de dato del argumento; puede ser cualquier tipo de dato SQL / PL/SQL. Puede ser el tipo de dato TYPE, %ROWTYPE, o cualquier tipo de dato escalar o compuesto. |
| *Bloque PL/SQL* | Cuerpo procedural que define la acción que realiza el procedimiento |

La opción **REPLACE** indica que, si el procedimiento existe, se borrará y se sustituirá por la nueva versión que ha creado la sentencia.

No se puede limitar el tamaño del tipo de dato en los parámetros.

## 11.1. Desarrollo de procedimientos

Estos son los principales pasos para desarrollar un procedimiento almacenado:




a) Escriba la sintaxis: Introduzca el código para crear un procedimiento (sentencia `CREATE PROCEDURE`) en un editor del sistema o en un procesador de textos y guárdelo como un archivo de comandos SQL (con extensión .sql ).

b) Compile el código: Utilice SQL*Plus para cargar y ejecutar el archivo de comandos SQL. El código de origen se compila en el *código P* y se crea el procedimiento.

   Los archivos de comandos con la sentencia `CREATE PROCEDURE` (o `CREATE OR REPLACE PROCEDURE`) permiten cambiar la sentencia en el caso de que se produzca algún error de compilación o de tiempo de ejecución, o realizar cambios posteriores en la sentencia. No se puede llamar a un procedimiento que contiene un error de compilación o de tiempo de ejecución. En SQL*Plus, utilice **`SHOW ERRORS`** para ver los errores de compilación. Si ejecuta la sentencia `CREATE PROCEDURE`, el código de origen se almacenará en el diccionario de datos, aunque el procedimiento contenga

errores de compilación. Solucione los errores del código utilizando el editor y recompile el código.

c) Ejecute el procedimiento para realizar la acción deseada. Después de compilar el código de origen y de haber creado el procedimiento, dicho procedimiento se puede ejecutar todas las veces necesarias utilizando el comando **EXECUTE** desde SQL*Plus. El compilador PL/SQL genera el *pseudocódigo* o el código P basándose en el código analizado. El motor PL/SQL lo ejecuta cuando se llama al procedimiento.

   Nota: Si hay algún error de compilación y se realizan cambios posteriores en la sentencia `CREATE PROCEDURE`, primero debe utilizar `DROP` en el procedimiento o la sintaxis `OR REPLACE`.

## 11.2. Parámetros Formales y Parámetros Reales

Los **parámetros formales** son variables declaradas en la lista de parámetros de la especificación de un subprograma. Por ejemplo, en el procedimiento *incrementar_salario* , las variables *p_empno* y *p_ incremento* son parámetros formales.

Los **parámetros reales** son variables o expresiones a las que se hace referencia en la lista de parámetros de la llamada de un subprograma. Por ejemplo, en la llamada *incrementar_salario*(1122, 20) al procedimiento *incrementar_salario* , las variables 1122 y 20 son parámetros reales.

Los parámetros reales se evalúan y los resultados se asignan a parámetros formales durante la llamada del subprograma. Los parámetros reales también pueden ser expresiones .

Es conveniente utilizar diferentes nombres para los parámetros formales y los reales. En este curso, los parámetros formales tienen el prefijo p_.

Los parámetros formales y reales deberían tener tipos de datos compatibles. Si es necesario, antes de asignar el valor, PL/SQL convierte el tipo de dato del valor del parámetro real en el del parámetro formal.

```
CREATE OR REPLACE PROCEDURE incrementar_salario
  (p_empno  emple.emp_no%TYPE,p_incremento NUMBER)
IS
BEGIN
  UPDATE emple
  SET    salario := salario*(1+p_incremento/100)
  WHERE   emp_no = p_empno;
END incrementar_salario;
/

EXECUTE incrementar_salario(1122, 20)
```

## 11.3. Tipos de parámetros

Se pueden transferir valores hasta y desde el entorno de llamada por medio de parámetros. Hay que seleccionar uno de los tres modos para cada parámetro: **IN, OUT o IN OUT**. Cualquier intento de cambiar el valor de un parámetro IN, producirá un error.

| Tipo de Parámetro | Descripción |
|---|---|
| IN (por defecto) | Transfiere un valor constante desde el entorno de llamada al procedimiento |
| OUT | Transfiere un valor desde el procedimiento al entorno de llamada |
| IN OUT | Transfiere un valor desde el entorno de llamada al procedimiento y un valor posiblemente diferente desde el procedimiento al entorno de llamada utilizando el mismo parámetro |

El modo de parámetro **IN** es el modo por defecto. Es decir, si no se especifica ningún modo con un parámetro, se considera que el parámetro es IN. Los modos **OUT** e **IN OUT** se deben especificar explícitamente delante de dichos parámetros.

A un parámetro formal **IN** no se le puede asignar un valor. Es decir, los parámetros IN no se pueden modificar en el cuerpo del procedimiento.

Hay que asignar un valor a los parámetros **OUT** o **IN OUT** antes de que regresen al entorno de llamada.

A los parámetros **IN** se les puede asignar un valor por defecto en la lista de parámetros. A los parámetros **OUT** e **IN OUT** no se les pueden asignar valores por defecto.

Por defecto, los parámetros **IN** se pasan por referencia y los parámetros **OUT** e **IN OUT** por valor.

| IN | OUT | IN OUT |
|---|---|---|
| Modo por defecto | Se debe especificar | Se debe especificar |
| El valor se pasa al subprograma | Se devuelve al entorno de llamada | Se transfiere al subprograma; vuelve al entorno de llamada |
| El parámetro formal funciona como una constante | Variable no inicializada | Variable inicializada |
| El parámetro real puede ser un literal, una expresión, una constante o una variable inicializada | Debe ser una variable | Debe ser una variable |
| Se le puede asignar un valor por defecto | No se le puede asignar un valor por defecto | No se le puede asignar un valor por defecto |

**Ejemplo PARAMETROS IN:**

```
CREATE OR REPLACE PROCEDURE incrementar_salario
 (p_id IN emple.emp_no%TYPE)
IS
BEGIN
 UPDATE emple
 SET salario = salario * 1.10
 WHERE  emp_no = p_id;
END incrementar_salario;
/
```

El ejemplo muestra un procedimiento con un parámetro IN. Si se ejecuta esta sentencia en SQL*Plus, se creará el procedimiento INCREMENTAR_SALARIO. Una vez llamado, el procedimiento acepta el parámetro para el identificador del empleado y actualiza el registro de dicho empleado con un aumento de sueldo del 10 por ciento.

Para llamar a un procedimiento en SQL*Plus, hay que utilizar el comando EXECUTE.

```
EXECUTE incrementar_salario (176)
```

Para llamar a un procedimiento desde otro procedimiento, hay que utilizar una llamada directa. Introduzca el nombre del procedimiento y los parámetros reales en el lugar de llamada del nuevo procedimiento.

```
incrementar_salario(176);
```

Los parámetros IN se transfieren como constantes desde el entorno de llamada al procedimiento. Cualquier intento de cambiar el valor de un parámetro IN, producirá un error.

**Ejemplo PARÁMETROS OUT:**

```
CREATE OR REPLACE PROCEDURE consultar_empleado
 (p_id IN emple.emp_no%TYPE,
  p_nombre OUT emple.apellido%TYPE,
  p_salario OUT emple.salario%TYPE,
  p_comi OUT emple.comision%TYPE)
IS
BEGIN
      SELECT apellido, salario, comision
      INTO   p_nombre, p_salario, p_comision
      FROM emple
      WHERE emp_no = p_id;
END consultar_empleado;
/
```

En este ejemplo, se crea un procedimiento con parámetros OUT para recuperar información acerca de un empleado. El procedimiento acepta el valor 171 para el identificador de empleado y recupera en los tres parámetros de salida el nombre, el sueldo y la comisión del empleado con el identificador 171.

Este procedimiento posee cuatro parámetros formales. Tres de ellos son parámetros OUT que devuelven valores al entorno de llamada.
El procedimiento acepta un valor EMPLOYEE_ID para el parámetro P_ID. Los valores del nombre, el sueldo y el porcentaje de comisión correspondientes al identificador del empleado se recuperan en tres parámetros OUT cuyos valores se devuelven al entorno de llamada.

Una vez compilado, podemos ejecutar el procedimeiento de la siguiente forma:

```
EXECUTE consultar_empleado(1712,:nombre, :salario, :comision);
```

Donde :nombre, :salario y:comision son variables del host en SQL*Plus definidas utilizando el comando VARIABLE:

```
VARIABLE   nombre   VARCAHAR2(25)
VARIABLE   salario  NUMBER
VARIABLE   comision NUMBER
```

Tenga en cuenta el uso de los dos puntos (:) para hacer referencia a las variables del host en el comando EXECUTE.

Para ver los valores que se transfieren desde el procedimiento al entorno de llamada, utilice el comando **PRINT.**

**PRINT nombre salario comision**

No especifique un tamaño para la variable del host de tipo de dato NUMBER cuando utilice el comando VARIABLE. Las variables del host de tipo de dato CHAR o VARCHAR2 tienen una longitud de uno por defecto, a menos que se proporcione un valor entre paréntesis.

PRINT y VARIABLE son comandos de SQL*Plus.

Nota: Si se transfiere una constante o una expresión como un parámetro real a la variable OUT, se producirán errores de compilación.

**Ejemplo PARÁMETROS IN OUT:**

Cree un procedimiento con un parámetro **IN OUT** que acepte una cadena de caracteres de 10 dígitos y devuelva un número de teléfono con el formato (800) 633-0575.

```
CREATE OR REPLACE PROCEDURE formatear_tfno
  (p_tfno_no IN OUT VARCHAR2)
IS
BEGIN
  p_tfno_no := '(' || SUBSTR(p_tfno_no,1,3) ||
               ')' || SUBSTR(p_tfno_no,4,3) ||
               '-' || SUBSTR(p_tfno_no,7);
END formatear_tfno;
/
```

Con los parámetros IN OUT, puede transferir valores a un procedimiento y devolver un valor al entorno de llamada. El valor que se devuelve es el original, un valor que no ha cambiado o un nuevo valor definido en el interior del procedimiento.

Los parámetros IN OUT funcionan como variables inicializadas.

Para ejecutar el procedimiento:

```
VARIABLE g_tfno_no VARCHAR2(15)

BEGIN
  :g_tfno_no := '8006330575';
END;
/

PRINT g_tfno_no
EXECUTE formatear_ tfno (:g_tfno_no)
PRINT g_tfno_no
```

## 11.4. Eliminación de Procedimientos

Cuando un procedimiento almacenado ya no sea necesario, puede utilizar una sentencia SQL para borrarlo.

Para eliminar un procedimiento del servidor utilizando SQL*Plus, hay que ejecutar el comando SQL **DROP PROCEDURE *nombre_procedimiento*.**

La emisión de rollbacks no tiene ningún efecto si ya se ha ejecutado un comando DDL (Lenguaje de Definición de Datos) como DROP PROCEDURE, que valida cualquier transacción pendiente.

# 12. Creación de funciones

Una función es un bloque PL/SQL denominado que puede aceptar parámetros y que se puede llamar. En términos generales, las funciones sirven para calcular un valor. Las funciones y los procedimientos tienen una estructura similar. Las funciones deben devolver un valor al entorno de llamada, mientras que los procedimientos devuelven cero o más valores a su entorno de llamada. Al igual que los procedimientos, las funciones poseen una cabecera, una sección declarativa, una sección ejecutable y una sección de manejo de excepciones opcional. Las funciones deben tener una cláusula **RETURN** en la cabecera y al menos una sentencia `RETURN` en la sección ejecutable.

Las funciones se pueden almacenar en la base de datos como objetos de esquema para repetir su ejecución. Las funciones que están almacenadas en la base de datos se denominan funciones almacenadas.

La función puede formar parte de una expresión SQL o parte de una expresión PL/SQL. En una expresión SQL, la función debe obedecer una serie de reglas específicas para controlar los efectos secundarios. En una expresión PL/SQL, el identificador de la función actúa como una variable cuyo valor depende de los parámetros que se le transfieran.

```
CREATE [OR REPLACE] FUNCTION function_name
      [(parameter1 [mode1] datatype1,
        parameter2 [mode2] datatype2,
        . . .)]
RETURN datatype
IS|AS
PL/SQL Block;
```

| Parámetro | Descripción |
|---|---|
| *nombre_función* | Nombre de la función |
| *parámetro* | Nombre de una variable PL/SQL cuyo valor se transfiere a la función |
| modo | El tipo de parámetro; sólo se deberían declarar parámetros IN |
| *tipo_de_dato* | Tipo de dato del parámetro |
| *tipo_de_dato* RETURN | Tipo de dato del valor RETURN que debe dar como resultado la función |
| bloque PL/SQL | Cuerpo procedural que define la acción que realiza la función |

Debe haber al menos una sentencia **RETURN (*expression*)**.

**Ejemplo:**

Crear una función que devuelva el salario de un empleado dado.

```
CREATE OR REPLACE FUNCTION obtener_salario
      (p_id  IN  emple.emp_no%TYPE)
RETURN NUMBER
IS
      v_salario emple.salario%TYPE :=0;
BEGIN
      SELECT salario
      INTO   v_salario
      FROM   emple
      WHERE  empno = p_id;

      RETURN v_salary;
END obtener_salario;
/
```

Desde SQL*PLUS:

```
VARIABLE g_salario NUMBER

EXECUTE :g_salario := obtener_salario(117)

PRINT g_salario
```

Una función puede aceptar uno o varios parámetros, pero debe devolver un solo valor. Para llamar a funciones como parte de las expresiones PL/SQL, hay que utilizar variables que guarden el valor devuelto.

## 12.1. Llamada a Funciones Definidas por el Usuario desde Expresiones SQL

Las expresiones SQL pueden hacer referencia a las funciones PL/SQL definidas por el usuario. Las funciones definidas por el usuario se pueden colocar en los mismos lugares que las funciones SQL incorporadas.

**Ejemplo:**

```
CREATE OR REPLACE FUNCTION impuesto
(p_value IN NUMBER)
RETURN NUMBER
IS
BEGIN
   RETURN (p_value * 0.08);
END tax;
/


SELECT emp_no, apellido, salario, impuesto(salario)
FROM   emple
WHERE  dept_no = 10;
```

El ejemplo muestra cómo se crea la función **impuesto**, que se llama desde una sentencia SELECT. La función acepta un parámetro NUMBER y devuelve el impuesto después de multiplicar el valor del parámetro por 0.08.

En SQL*Plus, llame a la función **impuesto** en el interior de una consulta que muestre el código, el apellido, el sueldo y el impuesto del empleado.

Las funciones PL/SQL definidas por el usuario se pueden llamar desde cualquier expresión SQL desde la que se pueda llamar a una función incorporada.

**Ejemplo:**

```
SELECT  empno, impuesto(salario)
FROM    emple
WHERE impuesto(salario)>(SELECT MAX(impuesto(salario))
                         FROM emple
                         WHERE dept_no = 30)
ORDER BY impuesto(salario) DESC;
```

Restricciones al Llamar a Funciones desde Expresiones SQL

- Las funciones PL/SQL almacenadas no se pueden llamar desde la cláusula de restricción `CHECK` de un comando `CREATE o ALTER TABLE` ni se pueden utilizar para especificar un valor por defecto para una columna.
- Debe poseer o tener el privilegio `EXECUTE` sobre la función para poder llamarla desde una sentencia SQL.
- Las funciones deben devolver tipos de dato SQL válidos. No pueden ser tipos de dato específicos de PL/SQL como, por ejemplo, `BOOLEAN`, `RECORD o TABLE`. Esta misma restricción se aplica a los parámetros de la función.
- No pueden contener instrucciones **DML.**
- Si una instrucción DML modifica una determinada tabla, en dicha instrucción no se puede invocar a una función que realice consultas sobre la misma tabla.
- No pueden utilizar instrucciones de transacciones (**`COMMIT`**, **`ROLLBACK`**,...).
- La función no puede invocar a otra función que se salte alguna de las reglas anteriores.
- Desde las sentencias SQL sólo se puede llamar a funciones almacenadas. No se puede llamar a procedimientos almacenados.

**Ejemplo:**

```
CREATE OR REPLACE FUNCTION dml_call_sql (p_sal NUMBER)
RETURN NUMBER
IS
BEGIN
  INSERT INTO emple(emp_no,apellido,fecha_alt,oficio, salario,
dept_no)
   VALUES(1, 'RUIZ',SYSDATE, 'ANALISTA', 1000, 10);
 RETURN (p_sal + 100);
END;
/

UPDATE employees SET salary = dml_call_sql(2000)
WHERE empl_no = 1712;
```

La sentencia UPDATE devuelve un error que informa de que la tabla está mutando.

**Ejemplo:**

Observe el siguiente ejemplo, donde la función QUERY_CALL_SQL consulta la columna SALARIO de la tabla EMPLE:

```
CREATE OR REPLACE FUNCTION query_call_sql(a NUMBER)
RETURN NUMBER
IS
    s NUMBER;
BEGIN
     SELECT salario INTO s
     FROM emple
     WHERE emp_no = 1712;
     RETURN (s + a);
END;
/
```

Cuando se llama a la función anterior desde la siguiente sentencia **UPDATE**, devuelve un mensaje de error similar al anterior.

```
UPDATE emple SET salario = query_call_sql(100)
WHERE employee_id = 1712;
```

## 12.2. Eliminación de funciones

Para eliminar una función almacenada utilizando SQL*Plus, hay que ejecutar el comando SQL **DROP FUNCTION *nombre_función***

# 13. Vistas del diccionario de datos

### Uso de USER_OBJECTS

Para obtener el nombre de todos los objetos PL/SQL almacenados en un esquema, consulte la vista de diccionario de datos **USER_OBJECTS.**

También puede examinar las vistas **ALL_OBJECTS y DBA_OBJECTS**, cada una de las cuales contiene la columna OWNER adicional para el propietario del objeto.

| Columna | Descripción de la Columna |
|---|---|
| OBJECT_NAME | Nombre del objeto |
| OBJECT_ID | Identificador interno del objeto |
| OBJECT_TYPE | Tipo de objeto como, por ejemplo, TABLE, PROCEDURE, FUNCTION, PACKAGE, PACKAGE BODY, TRIGGER |
| CREATED | Fecha de creación del objeto |
| LAST_DDL_TIME | Fecha de la última modificación del objeto |
| TIMESTAMP | Fecha y hora de la última recompilación del objeto |
| STATUS | VALID o INVALID |

*Lista de columnas resumida

**Ejemplo:**

Mostrar los nombres de todos los procedimientos y todas las funciones que se han creado.

```
SELECT object_name, object_type
FROM  USER_OBJECTS
WHERE object_type in ('PROCEDURE','FUNCTION')
ORDER BY object_name;
```

## Uso de USER_SOURCE

Para obtener el texto de un procedimiento o una función almacenada, utilice la vista de diccionario de datos **USER_SOURCE**.

Examine también las vistas **ALL_SOURCE y DBA_SOURCE**, cada una de las cuales contiene la columna OWNER adicional para el propietario del objeto.

| Columna | Descripción de la Columna |
|---|---|
| NAME | Nombre del objeto |
| TYPE | Tipo de objeto como, por ejemplo, PROCEDURE, FUNCTION, PACKAGE, PACKAGE BODY |
| LINE | Número de línea del código de origen |
| TEXT | Texto de la línea del código de origen |

**Ejemplo:**

Utilice la vista de diccionario de datos USER_SOURCE para mostrar el texto completo del procedimiento QUERY_EMPLOYEE.

```
SELECT  text
FROM  USER_SOURCE
WHERE  name = 'QUERY_EMPLOYEE'
ORDER BY  line;
```

## Obtención de Errores de Compilación

Para obtener el texto de los errores de compilación, utilice la vista de diccionario de datos **USER_ERRORS** o el comando **SHOW ERRORS** de SQL*Plus.

Examine también las vistas **ALL_ERRORS y DBA_ERRORS**, cada una de las cuales contiene la columna OWNER adicional para el propietario del objeto.

| Columna | Descripción de la Columna |
|---|---|
| NAME | Nombre del objeto |
| TYPE | Tipo de objeto como, por ejemplo, PROCEDURE, FUNCTION, PACKAGE, PACKAGE BODY, TRIGGER |
| SEQUENCE | Número de secuencia, para realizar el ordenamiento |
| LINE | Número de la línea del código de origen en la que se produce el error |
| POSITION | Posición de la línea en la que se produce el error |
| TEXT | Texto del mensaje de error |

**Ejemplo:**

Lista de los errores de compilación utilizando **USER_ERRORS.**

```
SELECT line || '/' || position POS, text
FROM   USER_ERRORS
WHERE  name = 'LOG_EXECUTION'
ORDER BY line;
```

Utilice **SHOW ERRORS** sin ningún argumento en el prompt de SQL para obtener los errores de compilación del último objeto que haya compilado.

```
SHOW ERRORS PROCEDURE log_execution
```

### Descripción de Procedimientos y Funciones

Para mostrar un procedimiento o una función y su lista de parámetros, utilice el comando **DESCRIBE** de SQL*Plus.

**Ejemplo:**

Mostrar la lista de parámetros de los procedimientos QUERY_EMPLOYEE y ADD_DEPT y de la función TAX.

```
DESCRIBE query_employee
DESCRIBE add_dept
DESCRIBE tax
```

# 14. Excepciones

Se llama **excepción** a todo hecho que le sucede a un programa que causa que la ejecución del mismo finalice. Lógicamente eso causa que el programa termine de forma anormal.

Los bloques terminan siempre cuando PL/SQL produce una excepción, pero se puede especificar un manejador de excepciones para realizar las acciones finales.

Hay dos métodos para producir una excepción;

- Cuando se produce un error de Oracle, la excepción asociada se produce automáticamente . Por ejemplo, si se produce el error `ORA-01403` cuando no se recupera ninguna fila de la base de datos en una sentencia `SELECT`, PL/SQL emite la excepción `NO_DATA_FOUND`.

- Para emitir una excepción explícitamente, se produce la sentencia `RAISE` en el interior del bloque. La excepción que se produce puede ser definida por el usuario o predefinida.

Las excepciones se pueden capturar a fin de que el programa controle mejor la existencia de las mismas.

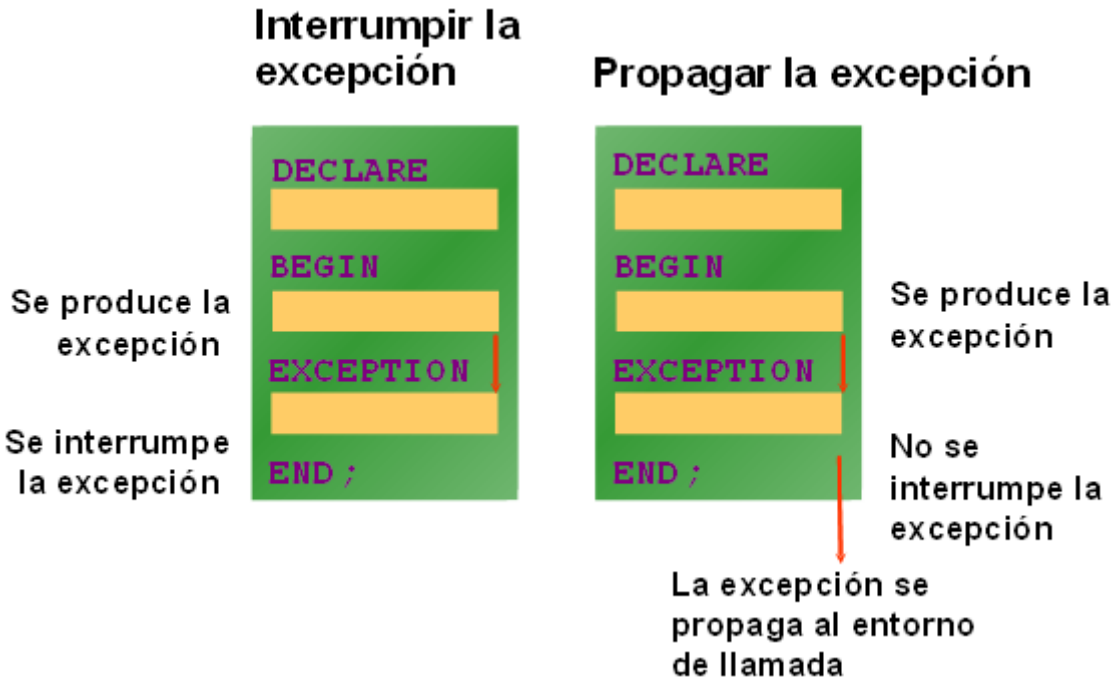
## 14.1. Manejo de excepciones

### Interrupción de una Excepción

Si la excepción se produce en la sección ejecutable del bloque, el procesamiento deriva al manejador de excepciones correspondiente en la sección de excepciones del bloque. Si PL/SQL consigue manejar la excepción, dicha excepción no se propaga al bloque o al entorno delimitador. El bloque PL/SQL finaliza con éxito.

### Propagación de una Excepción

Si la excepción se produce en la sección ejecutable del bloque y no existe un manejador de excepciones correspondiente, el bloque PL/SQL termina con un error y la excepción se propaga al entorno de llamada.



## 14.2. Tipos de excepciones

Existen tres tipos de excepciones:

| Excepción | Descripción | Instrucciones de Manejo |
|---|---|---|
| Error predefinido de Oracle Server | Uno de los aproximadamente 20 errores que se producen con más frecuencia en el código PL/SQL | No hay que declararlas y hay que dejar que Oracle Server las emita implícitamente |
| Error no predefinido de Oracle Server | Cualquier otro error estándar de Oracle Server | Hay que declararlas en la sección declarativa y hay que dejar que Oracle Server las emita implícitamente |
| Error definido por el usuario | Una condición que el desarrollador considera que no es normal | Hay que declararlas en la sección declarativa *y* emitirlas explícitamente |

## 14.3. Interrupción de Excepciones

Puede interrumpir cualquier error utilizando la rutina correspondiente en el interior de la sección de manejo de excepciones del bloque PL/SQL. Cada manejador se compone de una cláusula **WHEN**, que especifica una excepción, seguida de una secuencia de sentencias que se van a ejecutar cuando se produzca esa excepción.

```
EXCEPTION
      WHEN exception1 [OR exception2 . . .] THEN
      statement1;
      statement2;
. . .
      [WHEN exception3 [OR exception4 . . .] THEN
      statement1;
      statement2;
. . .  ]
      [WHEN OTHERS THEN
      statement1;
      statement2;
. . .  ]
```

En la sintaxis:

*excepción* es el nombre estándar de una excepción predefinida o el nombre de una excepción definida por el usuario que se ha declarado en la sección.

*Sentencia* es una o varias sentencias PL/SQL o SQL.

*OTHERS* es una clausula de manejo de excepciones opcional que interrumpe las excepciones no especificadas.

La sección de manejo de excepciones sólo interrumpe aquellas excepciones que se hayan especificado; no se interrumpe ninguna otra excepción, a menos que se utilice el manejador de excepciones `OTHERS`. Este manejador interrumpe cualquier excepción que todavía no se haya manejado. Por esta razón, `OTHERS` es el último manejador de excepciones que se define.

## 14.4. Errores predefinidos de Oracle Server

Para interrumpir un error predefinido de Oracle Server, haga referencia a su nombre estándar en el interior de la rutina de manejo de excepciones correspondiente.

Es conveniente manejar siempre las excepciones **`NO_DATA_FOUND`** y **`TOO_MANY_ROWS`**, que son las más habituales.

```
DECLARE
 x NUMBER :=0;
 y NUMBER := 3;
 res NUMBER;

BEGIN
 res:=y/x;
 DBMS_OUTPUT.PUT_LINE(res);

EXCEPTION
 WHEN ZERO_DIVIDE THEN
      DBMS_OUTPUT.PUT_LINE('No se puede dividir por cero') ;
 WHEN OTHERS THEN
      DBMS_OUTPUT.PUT_LINE('Error inesperado') ;
END;
```

| Nombre de excepción | Número de error | Ocurre cuando.. |
|---|---|---|
| ACCESS_INTO_NULL | ORA-06530 | Se intentan asignar valores a un objeto que no se había inicializado |
| CASE_NOT_FOUND | ORA-06592 | Ninguna opción WHEN dentro de la instrucción CASE captura el valor, y no hay instrucción ELSE |
| COLLECTION_IS_NULL | ORA-06531 | Se intenta utilizar un *varray* o una tabla anidada que no estaba inicializada |
| CURSOR_ALREADY_OPEN | ORA-06511 | Se intenta abrir un cursor que ya se había abierto |
| DUP_VAL_ON_INDEX | ORA-00001 | Se intentó añadir una fila que provoca que un índice único repita valores |
| INVALID_CURSOR | ORA-01001 | Se realizó una operación ilegal sobre un cursor |
| INVALID_NUMBER | ORA-01722 | Falla la conversión de carácter a número |
| LOGIN_DEINED | ORA-01017 | Se intenta conectar con Oracle usando un nombre de usuario y contraseña inválidos |
| NO_DATA_FOUND | ORA-01403 | El SELECT de fila única no devolvió valores |
| PROGRAM_ERROR | ORA-06501 | Error interno de Oracle |
| ROWTYPE_MISMATCH | ORA-06504 | Hay incompatibilidad de tipos entre el cursor y las variables a las que se intentan asignar sus valores |
| STORAGE_ERROR | ORA-06500 | No hay memoria suficiente |
| SUBSCRIPT_BEYOND_COUNT | ORA-06533 | Se hace referencia a un elemento de un *varray* o una tabla anidada usando un índice mayor que los elementos que poseen |
| SUBSCRIPT_OUTSIDE_LIMIT | ORA-06532 | Se hace referencia a un elemento de un *varray* o una tabla anidada usando un índice cuyo valor está fuera del rango legal |
| SYS_INVALID_ROWID | ORA-01410 | Se convierte un texto en un número de identificación de fila (ROWID) y el texto no es válido |
| TIMEOUT_ON_RESOURCE | ORA-00051 | Se consumió el máximo tiempo en el que Oracle permite esperar al recurso |
| TOO_MANY_ROWS | ORA-01422 | El SELECT de fila única devuelve más de una fila |
| VALUE_ERROR | ORA-06502 | Hay un error aritmético, de conversión, de redondeo o de tamaño en una operación |
| ZERO_DIVIDE | ORA-01476 | Se intenta dividir entre el número cero. |

## 14.5. Errores no predefinidos de Oracle Server

Pueden ocurrir otras muchas excepciones que no están en la lista anterior. En ese caso aunque no tienen un nombre asignado, sí tienen un número asignado. Ese número es el que aparece cuando Oracle muestra el mensaje de error tras la palabra ORA.

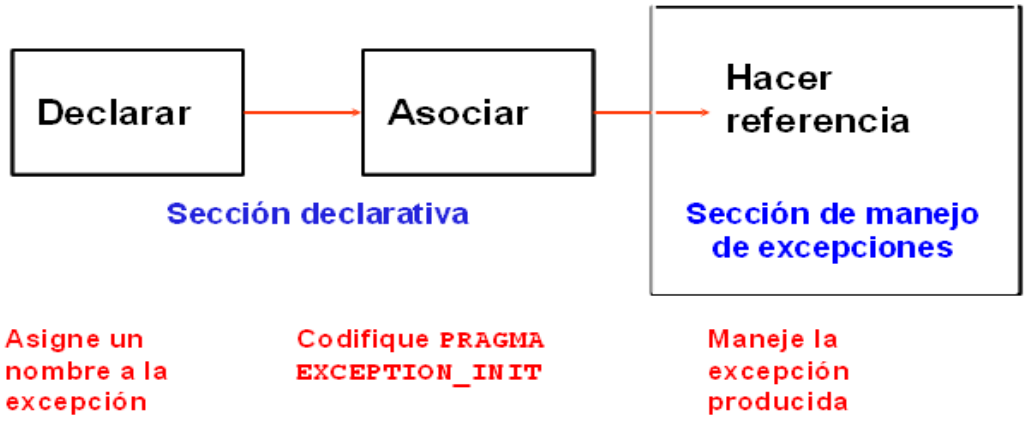
Por ejemplo en un error por restricción de integridad Oracle lanza un mensaje encabezado por el texto: **ORA-02292.** Por lo tanto el error de integridad referencia es el —02292.

Si deseamos capturar excepciones sin definir hay que:

- Declarar un nombre para la excepción que capturaremos. Eso se hace en el apartado **DECLARE** con esta sintaxis:
  nombreDeExcepción **EXCEPTION;**

- Asociar ese nombre al número de error correspondiente mediante esta sintaxis en el apartado **DECLARE** :
  **PRAGMA EXCEPTION_INIT(nombreDeExcepción, n°DeExcepción);**

- En el apartado **EXCEPTION** capturar el nombre de la excepción como si fuera una excepción normal.




**Ejemplo:**

Si en un departamento hay empleados, imprima un mensaje que informe al usuario de que el departamento no se puede eliminar.

```
DECLARE
 e_integridad EXCEPTION;
 PRAGMA EXCEPTION_INIT(e_integridad, -2292);

BEGIN
 DELETE FROM depart WHERE dept_no=20;
 COMMIT;

EXCEPTION
 WHEN e_integridad THEN
      DBMS_OUTPUT.PUT_LINE('No se puede borrar ese departamento
                            porque tiene empleados relacionados');
END;
```

## 14.6. Funciones de interrupción de errores

Cuando se produce una excepción, se puede identificar el código de error o el mensaje de error asociado utilizando dos funciones. Basándose en los valores del código o del mensaje, puede decidir qué acción va a realizar posteriormente en función del error.

**SQLCODE** devuelve el número de error de Oracle de las excepciones internas. Puede enviar un número de error a **SQLERRM**, para que le devuelva el mensaje asociado a ese número de error.

| Función | Descripción |
|---|---|
| SQLCODE | Devuelve el valor numérico del código de error (puede asignarlo a una variable NUMBER). |
| SQLERRM | Devuelve datos de caracteres que contienen el mensaje asociado al número de error |

| Valor de SQLCODE | Descripción |
|---|---|
| 0 | No se ha encontrado ninguna excepción |
| 1 | Excepción definida por el usuario |
| +100 | Excepción NO_DATA_FOUND |
| *número negativo* | Otro número de error de Oracle Server |

**Ejemplo:**

```
DECLARE
  v_error_code      NUMBER;
  v_error_message   VARCHAR2(255);
BEGIN
...
EXCEPTION
...
  WHEN OTHERS THEN

    ROLLBACK;
    v_error_code := SQLCODE ;
    v_error_message := SQLERRM ;

    INSERT INTO errors  VALUES(v_error_code, v_error_message);
END;
```

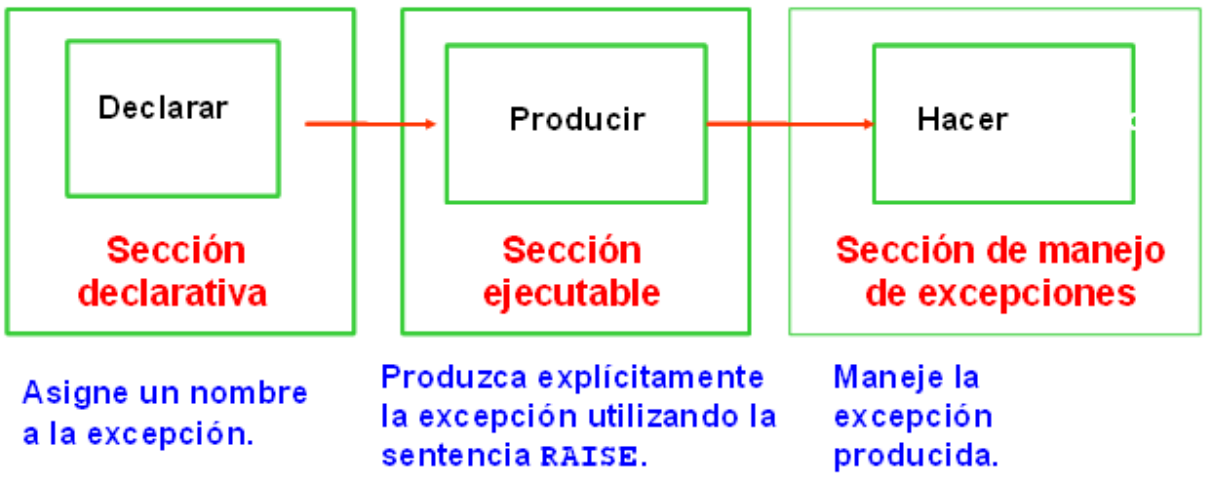
**Ejemplo:**

```
EXCEPTION
...
WHEN OTHERS THEN
DBMS_OUTPUT.PUT_LINE('Ocurrió el error ' || SQLCODE
||'mensaje: ' || SQLERRM);
END;
```

## 14.7. Interrupción de excepciones definidas por el usuario

PL/SQL le permite definir sus propias excepciones. Las excepciones definidas por el usuario deben ser:

- Declaradas en la sección declarativa de un bloque PL/SQL.
- Producidas explícitamente con sentencias RAISE.



**Ejemplo:**

Este bloque actualiza la descripción de un departamento. El usuario proporciona el número del departamento y el nuevo nombre. Si el usuario introduce un número de departamento que no existe, no se actualizará ninguna fila de la tabla DEPART. Produzca una excepción e imprima un mensaje que informe al usuario de que ha introducido un número de departamento que no es válido.
Nota: Utilice la sentencia RAISE propia en un manejador de excepciones para volver a producir la misma excepción en el entorno de llamada.

```
DEFINE p_depart_nombre = 'Information Technology '
DEFINE p_depart_no = 300

DECLARE
  e_invalid_depart EXCEPTION;
BEGIN
  UPDATE  depart
  SET     dnombre = '&p_depart_nombre'
  WHERE   dept_no = &p_depart_no;

  IF SQL%NOTFOUND THEN
    RAISE e_invalid_department;
  END IF;
  COMMIT;
EXCEPTION
  WHEN e_invalid_department  THEN
    DBMS_OUTPUT.PUT_LINE('No existe el departamento.');
END;
```

Otra forma es utilizar la función **RAISE_APPLICATION_ERROR** que simplifica los tres pasos anteriores. Sintaxis:

```
RAISE_APPLICATION_ERROR(nºDeError, mensaje, [{TRUE|FALSE}]);
```

Esta instrucción se coloca en la sección ejecutable o en la de excepciones y sustituye a los tres pasos anteriores. Lo que hace es lanzar un error cuyo número debe de estar entre el -20000 y el -20999 y hace que Oracle muestre el mensaje indicado. El tercer parámetro puede ser TRUE o FALSE (por defecto TRUE) e indica si el error se añade a la pila de errores existentes.

```
DECLARE
BEGIN
DELETE FROM depart WHERE dept_no=60;
IF SQL%NOTFOUND THEN
 RAISE_APPLICATION_ERROR(-20001,'No existe ese departamento');
END IF;
END;
```

En el ejemplo, si el departamento no existe, entonces SQL %NOTFOUND devuelve verdadero ya que el DELETE no elimina ningún departamento. Se lanza la excepción de usuario -20001 haciendo que Oracle utilice el mensaje indicado. Oracle lanzará el mensaje: ORA-20001: No existe ese departamento.

# 15. Cursores

Oracle Server utiliza unas áreas de trabajo, que se denominan áreas SQL privadas, para ejecutar sentencias SQL y almacenar la información del procesamiento. Se pueden utilizar cursores PL/SQL para asignar un nombre a un área SQL privada y acceder a la información que tiene almacenada.

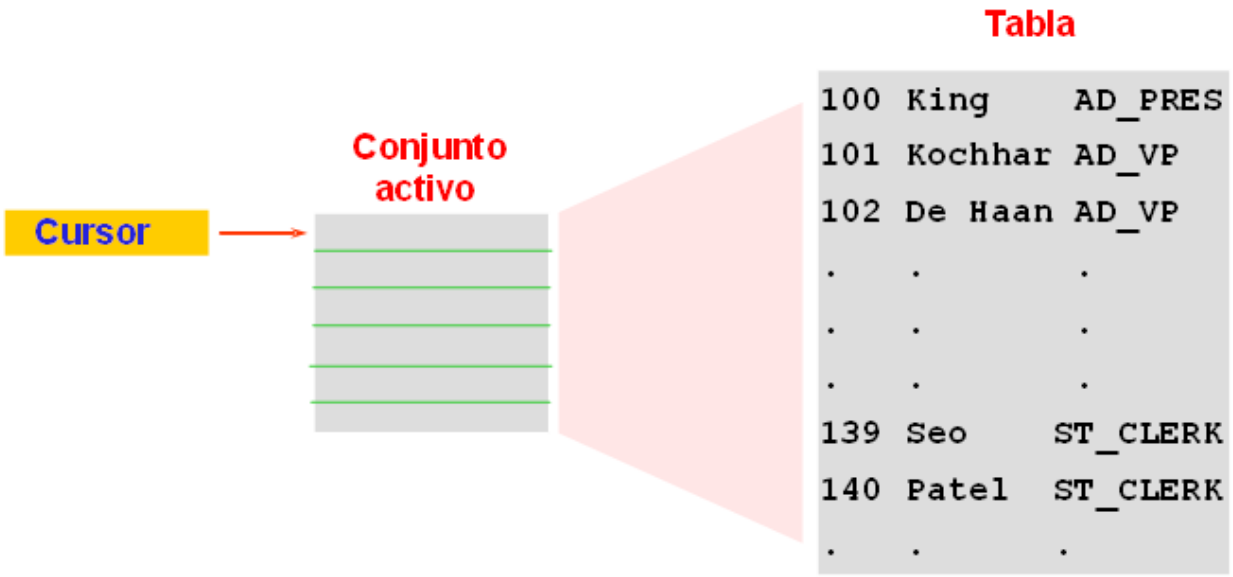
| Tipo de Cursor | Descripción |
|---|---|
| Implícito | PL/SQL declara implícitamente los cursores implícitos para todas las sentencias DML y SELECT de PL/SQL, incluidas las consultas que sólo devuelven una fila. |
| Explícito | En las consultas que devuelven más de una fila, el programador declara y asigna un nombre a los cursores explícitos y se manipulan por medio de sentencias específicas en las acciones ejecutables del bloque. |

## 15.1. Cursores explícitos

Utilice cursores explícitos para procesar individualmente cada fila que se devuelve en una sentencia `SELECT` de múltiples filas.

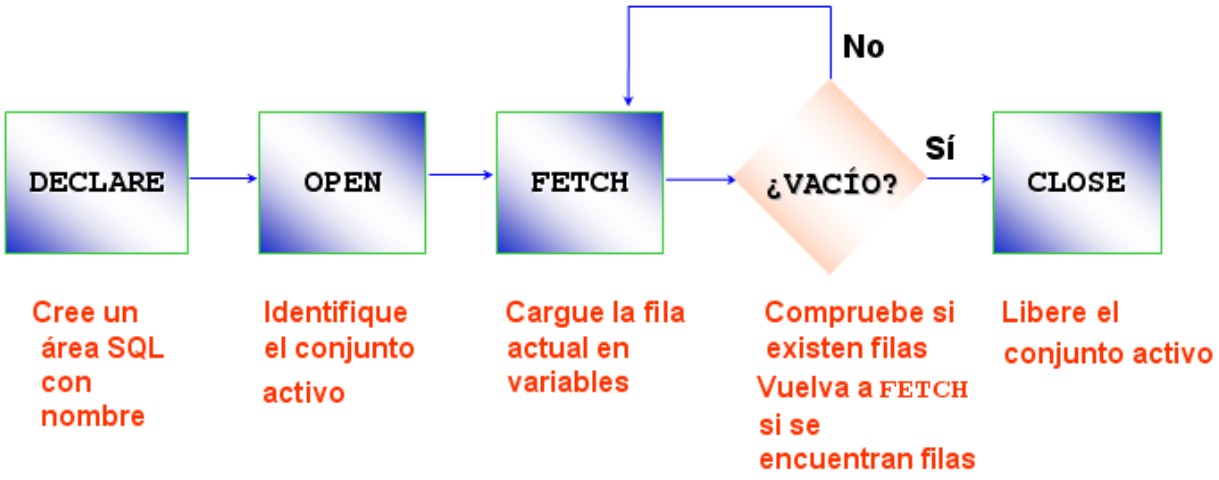
El conjunto de filas que devuelve una consulta de múltiples filas se denomina **conjunto activo**. Su tamaño equivale al número de filas que cumplen el criterio de búsqueda. El diagrama de la siguiente página muestra cómo un cursor explícito "apunta" a la *fila actual* del conjunto activo. De esta manera, el programa puede procesar las filas de una en una.

Los programas PL/SQL abren un cursor, procesan las filas que devuelve una consulta y luego cierran el cursor. El cursor marca la posición actual en el conjunto activo.




## 15.2. Control de cursores explícitos

- Declare el cursor asignándole un nombre y definiendo la estructura de la consulta que se le va a realizar.
- Abra el cursor. La sentencia **OPEN** ejecuta la consulta y enlaza cualquier variable a la que se haga referencia. Las filas que se identifican en la consulta se denominan conjunto activo y ya están disponibles para realizar recuperaciones.
- Recupere datos desde el cursor mediante la sentencia **FETCH**. En el diagrama de flujo, después de cada recuperación hay que probar el cursor para comprobar si hay alguna fila. Si no hay más filas que procesar, debe cerrar el cursor.
- Cierre el cursor. La sentencia **CLOSE** libera el conjunto de filas activo. Ahora ya se puede volver a abrir el cursor para establecer un nuevo conjunto activo.




Para controlar un cursor, utilice las sentencias **OPEN, FETCH y CLOSE**.

La sentencia **OPEN** ejecuta la consulta asociada al cursor, identifica el conjunto de resultados y coloca el cursor delante de la primera fila.

La sentencia **FETCH** recupera la fila actual y avanza hasta la siguiente fila hasta que ya no quedan más filas o hasta que se cumple la condición especificada.

Cierre el cursor cuando se haya procesado la última fila. La sentencia **CLOSE** desactiva el cursor.

## 15.3. Declaración del cursor

Utilice la sentencia **CURSOR** para declarar un cursor explícito. Puede hacer referencia a variables en la consulta, pero debe declararlas antes que la sentencia CURSOR.

```
CURSOR nombre_cursor IS
   sentencia_select;
```

En la sintaxis:

| | |
|---|---|
| *nombre_cursor* | es un identificador PL/SQL. |
| *sentencia_select* | es una sentencia **SELECT sin una cláusula INTO**. |

**Nota**

- No incluya la cláusula `INTO` en la declaración del cursor ya que aparece después en la sentencia `FETCH`.
- El cursor puede ser cualquier sentencia SELECT ANSI válida, incluir uniones, etc.

**Ejemplo:**

El cursor emp_cursor se declara para recuperar las columnas EMP_NO y APELLIDO de la tabla EMPLE.

```
DECLARE
  v_empno    emple.emp_no%TYPE;
  v_ape      emple.ape%TYPE;

  CURSOR emp_cursor IS
    SELECT emp_no, apellido
    FROM   emple;
BEGIN
...
```

## 15.4. Apertura del cursor

La sentencia **OPEN** ejecuta la consulta asociada al cursor, identifica el conjunto de resultados y coloca el cursor delante de la primera fila.

```
OPEN nombre_cursor;
```

Si la consulta no devuelve ninguna fila cuando el cursor está abierto, PL/SQL no produce ninguna excepción.

## 15.5. Recuperación de datos desde el cursor

La sentencia **FETCH** recupera las filas del conjunto activo de una en una. Después de cada recuperación, el cursor avanza hasta la siguiente fila del conjunto activo.

```
FETCH nombre_cursor INTO    [variable1,variable2,...]|nombre_registro];
```

En la sintaxis:

| | |
|---|---|
| *nombre_cursor* | es el nombre del cursor declarado previamente. |
| *variable* | es una variable de salida para almacenar los resultados. |
| *nombre_registro* | es el nombre del registro en el cual se almacenan los datos recuperados. (La variable del registro se puede declarar utilizando el atributo **%ROWTYPE**.). |

Instrucciones:

- Incluya tantas variables en la cláusula `INTO` de la sentencia `FETCH` como columnas en la sentencia `SELECT`, y asegúrese de que los tipos de datos son compatibles.
- Haga coincidir cada variable para que sus posiciones se correspondan con las de las columnas.
- Defina también un registro para el cursor y haga referencia al registro en la cláusula `FETCH INTO`.
- Compruebe si el cursor contiene filas. Si no se recupera ningún valor, no quedan más filas para procesar en el conjunto activo y no se registra ningún error.

Nota: La sentencia `FETCH` realiza las siguientes operaciones:

1. Lee los datos de la fila actual en las variables de salida PL/SQL.
2. Avanza el puntero hasta la siguiente fila del conjunto identificado.

```
Recupere los primeros 10 empleados, uno por uno.

SET SERVEROUTPUT ON
DECLARE
      v_empno  emple.emp_no%TYPE;
      v_ape    emple.apellido%TYPE;

       CURSOR   emp_cursor IS
         SELECT emp_no, apellido
         FROM   emple;
BEGIN
     OPEN emp_cursor;

     FOR i IN 1..10 LOOP
           FETCH emp_cursor INTO v_empno, v_ape;
           DBMS_OUTPUT.PUT_LINE (TO_CHAR(v_empno)||' '|| v_ape);
     END LOOP;
END ;
```

## 15.6. Cierre del cursor

La sentencia **CLOSE** desactiva el cursor, y el conjunto activo queda indefinido. Cierre el cursor después de terminar el procesamiento de la sentencia `SELECT`. Este paso permite volver a abrir el cursor, si es necesario. Por lo tanto, puede establecer un conjunto activo varias veces.

```
CLOSE nombre_cursor;
```

No intente recuperar datos desde un cursor después de haberlo cerrado, o se producirá la excepción `INVALID_CURSOR`.

La sentencia `CLOSE` libera el área de contexto.

Aunque se puede terminar el bloque PL/SQL sin cerrar los cursores, debería tener como costumbre cerrar cualquier cursor que haya declarado explícitamente para liberar recursos.

**Ejemplo:**

```
 OPEN emp_cursor
 FOR i IN 1..10 LOOP
   FETCH emp_cursor INTO v_empno, v_ape;
         ...
 END LOOP;
 CLOSE emp_cursor;
END;
```

## 15.7. Atributos de los cursores explícitos

Al igual que en los cursores implícitos, existen cuatro atributos para obtener información acerca del estado del cursor. Cuando se agregan al nombre de la variable de cursor, estos atributos devuelven información muy útil acerca de la ejecución de una sentencia de manipulación de datos.

| Atributo | Tipo | Descripción |
|---|---|---|
| %ISOPEN | Booleano | Se evalúa como TRUE si el cursor está abierto |
| %NOTFOUND | Booleano | Se evalúa como TRUE si la recuperación más reciente no devuelve una fila |
| %FOUND | Booleano | Se evalúa como TRUE si la recuperación más reciente devuelve una fila; complementa a %NOTFOUND |
| %ROWCOUNT | Numérico | Se evalúa como el número total de filas recuperadas hasta ese momento |

**Ejemplo:**

Recuperar los diez primeros empleados uno por uno.

```
DECLARE
  v_empno emple.emp_no%TYPE;
  v_ape   emple.apellido%TYPE;
  CURSOR emp_cursor IS
     SELECT emp_no, apellido
     FROM   emple;
BEGIN
  OPEN emp_cursor;
  LOOP
   FETCH emp_cursor INTO v_empno, v_ape;
     EXIT WHEN emp_cursor%ROWCOUNT > 10 OR emp_cursor%NOTFOUND;
     DBMS_OUTPUT.PUT_LINE (TO_CHAR(v_empno)||' '|| v_ape);
   END LOOP;
  CLOSE emp_cursor;
END;
```

Nota: Antes de la primera recuperación, %NOTFOUND se evalúa como NULL. Por lo tanto, si FETCH nunca se ejecuta con éxito, nunca se saldrá del bucle. Esto es debido a que la sentencia EXIT WHEN sólo se ejecuta si la condición WHEN es verdadera. Para estar seguro, utilice la siguiente sentencia EXIT: EXIT WHEN emp_cursor%NOTFOUND OR emp_cursor%NOTFOUND IS NULL;

## 15.8. Declaración de registros con el atributo %ROWTYPE

Para declarar un registro basándose en una recopilación de columnas de una tabla o una vista de base de datos, hay que utilizar el atributo **%ROWTYPE**. Los campos del registro obtienen sus nombres y tipos de dato de las columnas de la tabla o la vista. El registro también puede almacenar una fila completa de datos que se ha recuperado desde un cursor o una variable de cursor.

**Ejemplo:**

En el siguiente ejemplo, se ha declarado un registro utilizando %ROWTYPE como especificador de tipo de dato.

```
DECLARE  emp_record     emple%ROWTYPE;
```

La estructura del registro emp_record constará de cada uno de los campos de la tabla EMPLE.

Sintaxis

```
DECLARE
      identificador referencia%ROWTYPE;
```

donde:

*identificador* es el nombre elegido para el registro.

*referencia* es el nombre de la tabla, la vista, el cursor, o la variable de cursor en el cual se va a basar el registro. Para que esta referencia sea válida, la tabla o la vista deben existir.

Para hacer referencia o inicializar un campo individual, utilice la notación de puntos y la siguiente sintaxis:

```
nombre_registro.nombre_campo
```

Por ejemplo, para hacer referencia al campo *comision* del registro emp_record, hay que hacer lo siguiente:

```
emp_record.comision
```

Luego, puede asignar un valor a un campo de un registro de la siguiente manera:

```
emp_record.comision:= 0.35;
```

En el siguiente ejemplo, un empleado se va a jubilar. La información acerca de los empleados jubilados se agrega a una tabla en la que se guarda este tipo de información. El usuario proporciona el número de empleado. El registro del empleado especificado por el usuario se recupera de EMPLE y se almacena en la variable emp_rec, que se declara con el atributo %ROWTYPE.

```
DEFINE emple_number = 124

DECLARE
  emp_rec   emple%ROWTYPE;

BEGIN
  SELECT * INTO emp_rec
  FROM emple
  WHERE  emp_no = &emple_number;

  INSERT INTO emp_jubilados
  VALUES (emp_rec.emp_no, emp_rec.apellido, emp_rec.oficio,
          emp_rec.dir, SYSDATE, emp_rec.salario,emp_rec.comision,
          emp_rec.dept_no);
  COMMIT;
END;
/
```

```
A continuación se muestra el registro que se ha insertado en
la tabla EMP_JUBILADOS:

 SELECT * FROM EMP_JUBILADOS;
```

## 15.9. Cursores y registros

Ya ha comprobado que puede definir registros que tienen una estructura de columnas de una tabla. También puede definir un registro basándose en la lista de columnas seleccionada en un cursor explícito. Esto es conveniente para procesar las filas del conjunto activo, porque puede limitarse a realizar recuperaciones en el registro. Por lo tanto, los valores de la fila se cargan directamente en los campos correspondientes del registro.

```
Utilice un cursor para recuperar los números y los nombres de
los empleados y para rellenar una tabla de base de datos,
TEMP_LIST, con esta información.

    DECLARE
      CURSOR emp_cursor IS
        SELECT emp_no, apellido
        FROM   emple;
      emp_record emp_cursor%ROWTYPE;
    BEGIN
      OPEN emp_cursor;
      LOOP
        FETCH emp_cursor INTO emp_record;
        EXIT WHEN emp_cursor%NOTFOUND;
        INSERT INTO temp_list (empid, empape)
        VALUES (emp_record.emp_no, emp_record.apellido);
      END LOOP;
      COMMIT;
      CLOSE emp_cursor;
    END ;
    /
```

## 15.10. Bucles FOR de cursor

Los bucles FOR de cursor procesan las filas en un cursor explícito. Se trata de un método abreviado, porque el cursor se abre, se recuperan las filas una vez por cada iteración del bucle, se sale del bucle cuando se procesa la última fila y el cursor se cierra automáticamente. El propio bucle termina automáticamente al final de la iteración, donde se recupera la última fila.

```
FOR nombre_registro IN nombre_cursor LOOP
        sentencia1;
        sentencia2;
        . . .
```

```
END LOOP;
```

En la sintaxis:

| | |
|---|---|
| *nombre_registro* | es el nombre del registro declarado implícitamente. |
| *nombre_cursor* | es un identificador PL/SQL del cursor declarado previamente. |

Instrucciones:

- No declare el registro que controla el bucle, porque ya está declarado implícitamente.
- Pruebe los atributos del cursor durante el bucle, si es necesario.
- No utilice un bucle FOR de cursor cuando haya que gestionar explícitamente las operaciones del cursor.

**Ejemplo:**

```
Recupere los empleados, uno por uno, e imprima una lista de
aquellos que trabajen actualmente en el departamento de ventas
(dept_no = 30).

DECLARE
   CURSOR emp_cursor IS
        SELECT apellido, dept_no
        FROM   emple;
BEGIN
   FOR emp_record IN emp_cursor LOOP
    -- se produce una apertura y una lectura implícita.
     IF emp_record.dept_no = 30 THEN
         DBMS_OUTPUT.PUT_LINE ('El empleado '||
         emp_record.apellido|| ' trabaja en el dpto. ');
     END IF;
   END LOOP; -- se produce un cierre y una salida del bucle
                implícita
END ;
```

Si utiliza una **subconsulta en un bucle FOR**, no es necesario que declare un cursor. Este ejemplo hace lo mismo que el de la página anterior. A continuación está el código completo:

```
BEGIN
   FOR emp_record IN (SELECT apellido, dept_no
                      FROM   employees) LOOP
    -- se produce una apertura y una lectura implícita.
     IF emp_record.dept_no = 30 THEN
         DBMS_OUTPUT.PUT_LINE ('El empleado '||
         emp_record.apellido|| ' trabaja en el dpto. ');
     END IF;
   END LOOP; -- se produce un cierre y una salida del bucle
                implícita
```

```
END ;
```

## 15.11. Cursores Parametrizados

Los cursores son aquellos que permiten utilizar la orden OPEN para pasarle al cursor el valor de uno o varios de sus parámetros.

```
DECLARE
  CURSOR cArt (cFml Articulos.cArtFml%TYPE)
    IS SELECT cArtCdg,cArtDsc FROM Articulos WHERE cArtFml = cFml;
  xCod Articulos.cArtCdg%TYPE;
  xDes Articulos.cArtDsc%TYPE;
BEGIN
  OPEN cArt('F1');
  LOOP
    FETCH cArt INTO xCod,xDes;
    EXIT WHEN cArt%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE (xDes);
  END LOOP;
  CLOSE cArt;
END;
```

## 15.12. Cursores de actualización

Los cursores de actualización se declaran igual que los cursores explícitos, añadiendo FOR UPDATE al final de la sentencia SELECT.

```
CURSOR nombre_cursor
IS instrucción_SELECT FOR UPDATE
```

Para actualizar los datos del cursor hay que ejecutar una sentencia UPDATE especificando la cláusula WHERE CURRENT OF .

```
UPDATE <nombre_tabla> SET <campo_1> = <valor_1>[,<campo_2> = <valor_2>]
WHERE CURRENT OF <cursor_name>
```

Cuando trabajamos con cursores de actualización debemos tener en cuenta que la sentencia UPDATE genera bloqueos en la base de datos ( transacciones, disparadores,etc).

```
DECLARE
  CURSOR cpaises IS
   select CO_PAIS, DESCRIPCION, CONTINENTE from paises
  FOR UPDATE;
  co_pais VARCHAR2(3);
  descripcion VARCHAR2(50);
  continente VARCHAR2(25);
BEGIN
  OPEN cpaises;
  FETCH cpaises INTO co_pais,descripcion,continente;
  WHILE cpaises%found
    LOOP
      UPDATE PAISES SET CONTINENTE = CONTINENTE || '.'
      WHERE CURRENT OF cpaises;
      FETCH cpaises INTO co_pais,descripcion,continente;
    END LOOP;
  CLOSE cpaises;
  COMMIT;
END;
```

# 16. Disparadores o Triggers

Los triggers o disparadores son bloques de PL/SQL almacenados que se ejecutan o disparan automáticamente cuando se producen ciertos eventos. Hay tres grandes clases de disparadores de bases de datos:

- Disparadores de tablas.

  Asociados a una tabla y que se ejecutan automáticamente como reacción a una operación DML específica (INSERT, UPDATE o DELETE) sobre dicha tabla. En definitiva, los disparadores son eventos a nivel de tabla que se ejecutan automáticamente cuando se realizan ciertas operaciones sobre la tabla.

- Disparadores de sustitución.

  Asociados a vistas. Se disparan cuando se intenta ejecutar un comando de manipulación que afecta a la vista (inserción, borrado o modificación de filas)

- Disparadores del sistema.

  Se disparan cuando ocurre un evento del sistema (arranque o parada de la base de datos, entrada o salida de un usuario, etc) o una instrucción de definición de datos (creación, modificación o eliminación de una tabla u otro objeto)..

Para crear un disparador utilizaremos la siguiente instrucción:

```
CREATE {OR REPLACE} TRIGGER nombre_disp
  [BEFORE|AFTER]
    [DELETE|INSERT|UPDATE {OF columnas}] [ OR [DELETE|INSERT|UPDATE {OF
columnas}]...]
  ON tabla
  [FOR EACH ROW [WHEN condicion disparo]]
[DECLARE]
  -- Declaración de variables locales
BEGIN
  -- Instrucciones de ejecución
[EXCEPTION]
  -- Instrucciones de excepción
END;
```

El uso de  permite sobrescribir un trigger existente. Si se omite, y el trigger existe, se producirá, un error.

En principio, dentro del cuerpo de programa del trigger podemos hacer uso de cualquier orden de consulta o manipulación de la BD, y llamadas a funciones o procedimientos siempre que:

- No se utilicen comandos DDL
- No se acceda a las tablas que están siendo modificadas con DELETE, INSERT o UPDATE en la misma sesión
- No se violen las reglas de integridad, es decir no se pueden modificar claves primarias, ni actualizar claves externas
- No se utilicen sentencias de control de transacciones (Commit, Rollback o Savepoint)
- No se llamen a procedimientos que trasgredan la restricción anterior
- No se llame a procedimientos que utilicen sentencias de control de transacciones

#### Borrar un Disparador

```
DROP TRIGGER <NombreT>
```

#### Habilitar/Deshabilitar un Disparador, sin necesidad de borrarlo

```
ALTER TRIGGER <NombreT> {ENABLE | DISABLE} ( Esto no puede hacerse con los
subprogramas)
```

#### Diccionario de Datos

Todos los datos de un TRIGGER están almacenados en la vista USER_TRIGGERS.

Para ver todos los disparadores definidos por un usuario:

```
SELECT TRIGGER_NAME FROM USER_TRIGGERS
```

Para ver el cuerpo de un disparador:

```
SELECT TRIGGER_BODY FROM USER_TRIGGERS WHERE TRIGGER_NAME =
'nombre_disparador';
```

Para ver la descripción de un disparador:

```
SELECT DESCRIPTION FROM USER_TRIGGERS WHERE TRIGGER_NAME =
'nombre_disparador';
```

## 16.1. Restricciones de los Triggers.

El cuerpo de un disparador o trigger 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:

- disparador no puede emitir ninguna orden de control de transacciones ( COMMIT, ROLLBACK o SAVEPOINT). El disparador se activa como parte de la ejecución de la orden que provocó el disparo, y forma parte de la misma transacción que dicha orden. Cuando la orden que provoca la orden es confirmada o cancelada, se confirma o se cancela también el trabajo realizado por el disparador.
- Por las mismas razones, ningún procedimiento o función llamado por el disparador puede emitir órdenes de control de transacciones.
- El disparador no puede declarar variables de tipo LONG.
- No se utilicen comandos DDL
- No se acceda a las tablas que están siendo modificadas con DELETE, INSERT o UPDATE en la misma sesión
- No se violen las reglas de integridad, es decir no se pueden modificarclaves primarias, ni actualizar claves foraneas.

## 16.2. Disparadores o Triggers de Tabla

Se ejecuta de forma implícita cuando se ejecuta cierta operación DML: INSERT, DELETE o UPDATE. Contrariamente, los procedimientos y las funciones se ejecutan haciendo una llamada explícita a ellos. Un disparador no admite argumentos.

Cuando se crea un trigger con el comando CREATE TRIGGER hay que definir:

- La/s operaciones DML (INSERT, DELETE o UPDATE.) que van a provocar la ejecución del trigger.
- Para cada operación DML hay que indicar cuando dispararse si antes o después de la operación.El modificador BEFORE o AFTER indica que el trigger se ejecutará antes o después de ejecutarse la sentencia SQL definida por DELETE, INSERT o UPDATE. Si incluimos el modificador OF el trigger solo se ejecutará cuando la sentencia SQL afecte a los campos incluidos en la lista.

- Finalmente hay que indicar el nivel de alcance de los disparadores. La presencia o ausencia de la opción FOR EACH ROW determina si el disparador es a nivel de filas (row trigger) o a nivel de sentencia activadora (statement trigger)

  El alcance de los disparadores puede ser la fila o de orden DML.

  - Nivel de Orden (statement trigger): Se activan sólo una vez, antes o después de la Orden u operación DML.
  - Nivel de Fila (row trigger): Se activan una vez por cada Fila afectada por la operación DML (una misma operación DML puede afectar a varias filas).

    El modificador FOR EACH ROW indica que el trigger se disparará cada vez que se realizan operaciones sobre cada fila de la tabla. Si se acompaña del modificador WHEN, se establece una restricción; el trigger solo actuará, sobre las filas que satisfagan la restricción.

| ***Categoría*** | ***Valores*** | ***Comentarios*** |
|---|---|---|
| Orden | INSERT, DELETE, UPDATE | Define que tipo de operación DML provoca la activación del trigger |
| Temporalización | BEFORE o AFTER | Define si el disparador se activa antes o después de que se ejecute la operación DML |
| Nivel | Fila u Orden | Los disparadores con nivel de fila se activan una vez por cada fila afectada por la orden que provocó el disparo.<br>Los Triggers con nivel de orden se activan sólo una vez, antes o después de la orden.<br>Los disparadores con nivel de fila se identifican por la cláusula FOR EACH ROW en la definición del disparador. |

Ejemplo: Guardar en una tabla de control, Ctrl_Emple, la fecha y el usuario que modificó la tabla Empleados: (NOTA: Este trigger es AFTER y a nivel de orden)

```
CREATE OR REPLACE TRIGGER Control_Empleados
AFTER INSERT OR DELETE OR UPDATE ON Empleados
BEGIN
INSERT INTO Ctrl_Empleados (Tabla,Usuario,Fecha)
   VALUES ('Empleados', USER, SYSDATE);
END Control_Empleados;
```

Sólo se puede definir un trigger de cada tipo por tabla. Esto da doce posibilidades:

| | |
|---|---|
| BEFORE UPDATE row | AFTER UPDATE row |
| BEFORE DELETE row | AFTER DELETE row |
| BEFORE INSERT row | AFTER INSERT row |
| BEFORE UPDATE statement | AFTER UPDATE statement |
| BEFORE DELETE statement | AFTER DELETE statement |
| BEFORE INSERT statement | AFTER INSERT statement |

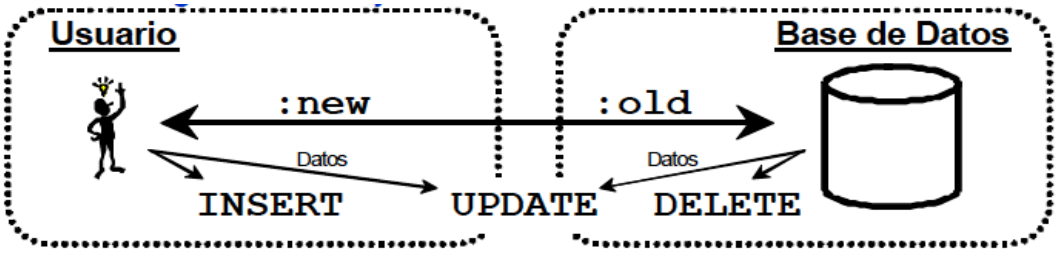
## 16.3. Disparadores a nivel de fila: utilización de :old y :new

Un disparador con nivel de fila se ejecuta una vez por cada fila procesada por la orden que provoca el disparo. Para acceder a la fila procesada en ese momento, se usan dos Pseudo-Registros de tipo <TablaDisparo>%ROWTYPE: :old y :new.

Asi dentro del ámbito de un trigger disponemos de las variables OLD y NEW . Estas variables se utilizan del mismo modo que cualquier otra variable PL/SQL, con la salvedad de que no es necesario declararlas, son de tipo %ROWTYPE y contienen una copia del registro antes (OLD) y después(NEW) de la acción SQL (INSERT, UPDATE, DELTE) que ha ejecutado el trigger. Utilizando esta variable podemos acceder a los datos que se están insertando, actualizando o borrando.

Estos Pseudo-Registros sirven para referirse al valor anterior y posterior a una modificación:

- INSERT: solo existe :new
- DELETE: solo existe :old
- UPDATE: existen :old y :new




Para usarlos en una clausula WHEN se le quitan los dos puntos.

Tabla resumen:

| ***Orden de Disparo*** | ***:old*** | ***:new*** |
|---|---|---|
| INSERT | No definido; todos los campos toman el 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 el valor NULL. |

Ejemplo:
Programar un disparador que calcule el campo código de alumno (cod_al) cada vez que se inserte un nuevo alumno**:**

```
CREATE OR REPLACE TRIGGER NuevoAlumno
BEFORE INSERT ON alumno FOR EACH ROW
```

```
BEGIN
  -- Establecer el nuevo número de alumno:
  SELECT MAX(cod_al)+1 INTO :new.cod_al FROM Alumno;
  IF :new.cod_al IS NULL THEN
   :new.cod_al := 1;
  END IF;
END NuevoAlumno;
```

Modificar Pseudo-Registros:
- No puede modificarse el Pseudo-Registro :new en un disparador AFTER a nivel de fila.
- El Pseudo-Registro :old nunca se modificará: Sólo se leerá.

**Formato: ... FOR EACH ROW WHEN <Condición_Disparo>**

Sólo es válida en disparadores a nivel de fila y siempre es opcional. Si existe, el cuerpo del disparador se ejecutará sólo para las filas que cumplan la condición de disparo especificada. En la condición de disparo pueden usarse los Pseudo-Registros :old y :new, pero en ese caso no se escribirán los dos puntos (:), los cuales son obligatorios en el cuerpo del disparador.

Ejemplo: Deseamos que los precios grandes no tengan más de 1 decimal. Si tiene 2 ó más decimales, redondearemos ese precio: si el Precio>200, redondearlos a un decimal:

```
CREATE OR REPLACE TRIGGER Redondea_Precios_Grandes
BEFORE INSERT OR UPDATE OF Precio ON Pieza
FOR EACH ROW WHEN (new.Precio > 200)
BEGIN
:new.Precio := ROUND(:new.Precio,1);
END Redondea_Precios_Grandes;
```

- Se puede escribir ese disparador sin la cláusula WHEN, usando un IF:

```
   BEGIN
            IF :new.Precio > 200 THEN
            :new.Precio := ROUND(:new.Precio,1);
   END IF;
```

**Otro ejemplo:**

```
CREATE OR REPLACE TRIGGER PesoPositivo -- Se activará cada vez que se inserte
o actualice
BEFORE INSERT OR UPDATE OF Peso ON Pieza FOR EACH ROW
BEGIN
  IF :new.Peso < 0
     THEN RAISE_APPLICATION_ERROR(-20100, 'Peso no válido');
  END IF;
END PesoPositivo;
```

## 16.4. Orden de ejecución de los triggers

Una misma tabla puede tener varios triggers asociados. En tal caso es necesario conocer el orden en el que se van a ejecutar.

Los disparadores se activan al ejecutarse la sentencia SQL.

- Si existe, se ejecuta el disparador de tipo BEFORE (disparador previo) con nivel de orden.
- Para cada fila a la que afecte la orden:
  - Se ejecuta si existe, el disparador de tipo BEFORE con nivel de fila.
  - Se ejecuta la propia orden.
  - Se ejecuta si existe, el disparador de tipo AFTER (disparador posterior) con nivel de fila.
- Se ejecuta, si existe, el disparador de tipo AFTER con nivel de orden.

## 16.5. Predicados condicionales

Dentro de un trigger en el que se disparan distintos tipos de órdenes DML (INSERT, UPDATE y DELETE), hay tres funciones booleanas que pueden emplearse para determinar de qué operación se trata.

Así cuando se crea un trigger para más de una operación DML, se puede utilizar un predicado condicional en las sentencias que componen el trigger que indique que tipo de operación o sentencia ha disparado el trigger.

Estos predicados son INSERTING, UPDATING y DELETING.

- **Inserting**: Retorna **true** cuando el trigger ha sido disparado por un **INSERT**
- **Deleting**: Retorna **true** cuando el trigger ha sido disparado por un **DELETE**
- **Updating**: Retorna **true** cuando el trigger ha sido disparado por un **UPDATE**
- **Updating (columna)**: Retorna **true** cuando el trigger ha sido disparado por un **UPDATE** y la columna ha sido modificada.

Ejemplo:

```
CREATE TRIGGER audit_trigger BEFORE INSERT OR DELETE OR UPDATE
  ON classified_table FOR EACH ROW
BEGIN
  IF INSERTING THEN
    INSERT INTO audit_table
    VALUES (USER || ' esta insertando' || ' la clave: ' || :new.key);
  ELSIF DELETING THEN
    INSERT INTO audit_table
    VALUES (USER || ' esta borrando' || ' la clave: ' || :old.key);
  ELSIF UPDATING THEN
    INSERT INTO audit_table
    VALUES (USER || ' esta cambiando' || ' el dato: ' || old.formula || ' por
este nuevo dato ' || :new.formula);
    END IF;
END;
```

Veamos otro ejemplo:

Creamos un trigger de actualización a nivel de fila sobre la tabla "libros". Ante cualquier modificación de los registros de "libros", se debe ingresar en la tabla "control", el nombre del usuario que realizó la actualización y la fecha; pero, controlamos que NO se permita modificar el campo "codigo", en caso de suceder, la acción no debe realizarse y debe mostrarse un mensaje de error indicándolo:

```
CREATE OR REPLACE TRIGGER tr_actualizar_libros
BEFORE UPDATE
ON libros
FOR EACH ROW
BEGIN
  if updating('codigo') then
   raise_application_error(-20001,'No se puede modificar el código libros');
  else
   insert into control values(user,sysdate);
  end if;
END;
```

Si se actualiza cualquier campo de "libros", se dispara el trigger; si se actualiza el campo "codigo", aparece un mensaje de error y la actualización no se realiza; en caso de actualizarse cualquier otro campo, se almacenará en "control", el nombre del usuario y la fecha.

## 16.6. Disparadores de Sustitución

Son disparadores que se ejecuta en lugar de la orden DML (ni antes ni después, sino sustituyéndola). Está asociados a una vísta.

Cuando se intenta modificar una vista esta modificación puede no ser posible debido al formato de esa vista. Características:

- Sólo pueden definirse sobre Vistas.
- Se activan en lugar de la Orden DML que provoca el disparo, o sea, la orden disparadora no se ejecutará nunca.
- Deben tener Nivel de Filas.
- Se declaran usando INSTEAD OF en vez de BEFORE/AFTER.

Sobre una vista podemos hacer un select pero si hacemos un insert, delete o update puede darnos problemas y no dejar ejecutarse la orden correctamente. Los trigger sobre vistas vas a sustituir a estas operaciones por las correspondientes en las tablas que forman la vista.

Supongamos que tenemos la siguiente vista

```
CREATE VIEW vista AS SELECT edificio, sum(numero_asientos)
 FROM habitaciones
GROUP BY edificio;
```

No se puede hacer una operación de borrado directamente en la vista:

```
DELETE FROM vista WHERE edificio='edificio 7';
```

Sin embargo, se puede crear un disparador de sustitución que efectúe el borrado equivalente pero sobre la tabla habitaciones.

```
CREATE TRIGGER borra_en_vista
INSTEAD OF DELETE ON vista FOR EACH ROW
BEGIN
  DELETE FROM habitaciones
  WHERE edificio = :OLD.edificio;
END borra_en_vista;
```

## 16.7. Ejemplo

Crear un trigger sobre la tabla empleados para que no se permita que un empleado sea jefe de más de cinco empleados.

La tabla empleados tiene la siguiente estructura:

```
DROP TABLE empleados;
CREATE TABLE empleados
 (id char(4),
 nomemp varchar2(15),
 idjefe char(4),
 PRIMARY KEY (id), FOREIGN KEY (idjefe) references empleados on delete
cascade);
```

En este ejemplo, se define un trigger para forzar que los datos de la tabla Empleado verifiquen la siguiente regla: "Un jefe no puede supervisar a más de cinco empleados". Para asegurar que cuando se modifique la tabla Empleado se cumpla esta restricción, tenemos que definir un trigger asociado a la tabla Empleado (ON Empleado).

Este trigger se dispara antes de insertar (Before Insert) un nuevo empleado ó antes de actualizar el atributo cojefe (Before Update Of cojefe) de un empleado que ya existe.

Cuando se dispara el trigger para cada fila (For Each Row), Oracle ejecuta implícitamente la acción: se comprueba si el número de empleados que supervisará el jefe si se realiza la modificación no supera a cinco. La modificación de la tabla Empleado (insertar un nuevo empleado o modificar el atributo cojefe de uno que ya existe) se llevará a cabo, si después de realizarla los datos verifican la regla.

En el bloque PL/SQL que define la acción, podemos acceder a los valores anteriores y nuevos de las columnas: OLD.columna y NEW.columna.

Esta regla de integridad de los datos de la tabla Empleado no podemos forzarla declarando Restricciones de Integridad.

```
CREATE OR REPLACE TRIGGER jefes
BEFORE INSERT ON empleados FOR EACH ROW
DECLARE supervisa INTEGER;
BEGIN
  SELECT count(*) INTO supervisa
    FROM empleados
    WHERE idjefe = :new.idjefe;
  IF supervisa > 4
    THEN raise_application_error (-20600,:new.idjefe||'no se puede supervisar
mas de 5');
  END IF;
END;
/
```

Ejecutar este código nos generaría un error de tabla mutante, una tabla mutante es una tabla que está siendo modificada por una sentencia SQL (insert, update, delete) o por el efecto de un DELETE CASCADE asociado a la sentencia SQL.
Una posible solucion seria, implementar la busqueda del numero de empleados a traves de un cursor.
DECLARE
CURSOR CEMPLE IS (select * from emple where jefe.emp_no=:new.dir);
…...
De esta forma no se accede a la misma tabla en el bloque PL/SQL sino que lo hace a traves de la carga de datos del cursor.

## 16.8. Disparadores de Sistema

Pueden funcionar para un esquema (o usuario) concreto o para toda la base de datos (todos los usuarios).

Los eventos que se pueden considerar son:

- CREATE, ALTER o DROP
- Conexión y desconexión
- Arranque y parada de la base de datos.

  Cuando se produzca un error concreto o un error cualquiera.

Sintaxis:

- Evento DDL (creación, modificación o borrado de un objeto):

```
CREATE OR REPLACE TRIGGER NombreTrigger
[BEFORE | AFTER]
[CREATE|ALTER|DROP] OR [CREATE|ALTER|DROP] …
ON [DATABASE | SCHEMA]

  DECLARE
  BEGIN
  EXCEPTION
  END;
```

Eventos del sistema:

```
CREATE OR REPLACE TRIGGER NombreTrigger
[eventosys] OR [eventosys] … ON [DATABASE | SCHEMA]
DECLARE
BEGIN
EXCEPTION
END;
eventosys puede ser uno de los siguientes:
  AFTER SERVERERROR
  AFTER LOGON
  BEFORE LOGOFF
  AFTER STARTUP
  BEFORE SHUTDOWN
```

# 17. Paquetes (Packages)

Un Paquete (Package ) es un objeto PL/SQL que agrupa lógicamente otros objetos PL/SQL relacionados entre sí, encapsulándolos y convirtiéndolos en una unidad dentro de la base de datos.

La estructura denominada Package (Paquete) agrupa procedimientos, funciones, definiciones de tipos de datos y declaraciones de variables en una misma estructura. Esta agrupación lógica nos permite no sólo mejorar la calidad de diseño de nuestras aplicaciones sino también optimizar el desempeño de las mismas.

Los Paquetes están divididos en 2 partes:

- especificación (obligatoria)
  La **especificación o encabezado** es la interfaz entre el Paquete y las aplicaciones que lo utilizan y es allí donde se declaran los tipos, variables, constantes, excepciones, cursores, procedimientos y funciones que podrán ser invocados desde fuera del paquete.

- cuerpo (no obligatoria).
  En el **cuerpo** del paquete se implementa la especificación del mismo. El cuerpo contiene los detalles de implementación y declaraciones privadas, manteniéndose todo esto oculto a las aplicaciones externas, siguiendo el conocido concepto de "caja negra". Sólo las declaraciones hechas en la especificación del paquete son visibles y accesibles desde fuera del paquete (por otras aplicaciones o procedimientos almacenados) quedando los detalles de implementación del cuerpo del paquete totalmente ocultos e inaccesibles para el exterior.

Para acceder a los elementos declarados en un paquete basta con anteceder el nombre del objeto referenciado con el nombre del paquete donde está declarado y un punto, de esta manera: **Paquete.Objeto**, donde Objeto puede ser un tipo, una variable, un cursor, un procedimiento o una función declarados dentro del paquete.

Para referenciar objetos desde dentro del mismo paquete donde han sido declarados no es necesario especificar el nombre del paquete y pueden (deberían) ser referenciados directamente por su nombre.

Finalmente y siguiendo a la parte declarativa del cuerpo de un paquete puede, opcionalmente, incluirse la sección de inicialización del paquete. En esta sección pueden, por ejemplo, inicializarse variables que utiliza el paquete. La sección de inicialización se ejecuta sólo la primera vez que una aplicación referencia a un paquete, esto es, se ejecuta sólo una vez por sesión.

Como hemos dicho anteriormente, la creación de un paquete pasa por dos fases:

1. Crear la cabecera del paquete donde se definen que procedimientos, funciones, variables, cursores, etc. Disponibles para su uso posterior fuera del paquete. En esta parte solo se declaran los objetos, no se implementa el código.

2. Crear el cuerpo del paquete, donde se definen los bloques de código de las funciones y procedimientos definidos en la cabecera del paquete.

Para crear la cabecera del paquete utilizaremos la siguiente instrucción:

```
CREATE {OR REPLACE} PACKAGE nombre_de_paquete IS
-- Declaraciones
END;
```

Para crear el cuerpo del paquete utilizaremos la siguiente instrucción:

```
CREATE {OR REPLACE} PACKAGE BODY nombre_paquete IS
--Bloques de código
END;
```

Hay que tener en cuenta que toda declaración de función o procedimiento debe estar dentro del cuerpo del paquete, y que todo bloque de código contenido dentro del cuerpo debe estar declarado dentro de la cabecera de paquete.

Cuando se quiera acceder a las funciones, procedimientos y variables de un paquete se debe anteponer el nombre de este:

**Nombre_paquete.función(x);**
**Nombre_paquete.procedimiento(x);**
**Nombre_paquete.variable;**

Ejemplo:

```
CREATE OR REPLACE PACKAGE PK1 IS
  PROCEDURE xLis (xfamilia IN Articulos.cArtFml%TYPE);
END;

CREATE OR REPLACE PACKAGE BODY PK1 IS
  procedure xLis (xfamilia Articulos.cArtCdg%TYPE)
  IS
    xfam Articulos.cArtFml%type;
    xCod Articulos.cArtCdg%TYPE;
    xDes Articulos.cArtDsc%TYPE;

    CURSOR cArticulos IS SELECT cArtCdg,cArtDsc FROM Articulos
        WHERE cArtFml = xfam;
  BEGIN
    xfam := xfamilia;
    OPEN cArticulos;
    LOOP
      FETCH cArticulos INTO xCod,xDes;
      EXIT WHEN cArticulos%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE (xDes);
    END LOOP;
    CLOSE cArticulos;
  END;
END;
```

## 17.1. Ventajas del uso de Paquetes (Packages)

Dentro de las ventajas que ofrece el uso de paquetes podemos citar que:

- **Permite modularizar el diseño de nuestra aplicación**
  El uso de Paquetes nos permite encapsular aquellos elementos que estén relacionados entre sí (tipos, variables, procedimientos, funciones) en un único módulo PL/SQL que llevará un nombre que identifique el conjunto.

- **Otorga flexibilidad al momento de diseñar la aplicación**
  A la hora de diseñar una aplicación todo lo que necesitaremos inicialmente es la información de interfaz en la especificación del paquete. Puede codificarse y compilarse la especificación sin su cuerpo para posibilitar que otros programas que llaman a estos elementos declarados puedan compilarse sin errores. De esta manera podremos armar un "prototipo" de nuestro sistema antes de codificar el detalle del mismo.

- **Permite ocultar los detalles de implementación**
  Pueden especificarse cuáles tipos, variables y programas dentro del paquete son públicos (visibles y accesibles por otras aplicaciones y programas fuera del paquete) o privados (ocultos e inaccesibles fuera del paquete). Por ejemplo, dentro del paquete pueden existir procedimientos y funciones que serán invocados por otros programas, así como también otras rutinas de uso interno del paquete que no tendrán posibilidad de ser accedidas fuera del mismo. Esto asegura que cualquier cambio en la definición de estas rutinas internas afectará sólo al paquete donde se encuentran, simplificando el mantenimiento y protegiendo la integridad del conjunto.

- **Agrega mayor funcionalidad a nuestro desarrollo**
  Las definiciones públicas de tipos, variables y cursores hechas en la especificación de un paquete permanecen a lo largo de una sesión. Por lo tanto pueden ser compartidas por todos los programas y/o paquetes que se ejecutan en ese entorno durante esa sesión. Por ejemplo, puede utilizarse esta técnica (en dónde sólo se define una especificación de paquete y no un cuerpo) para mantener tipos y variables globales a todo el sistema.

- **Introduce mejoras al rendimiento**
  En relación a su ejecución, cuando un procedimiento o función que está definido dentro de un paquete es llamado por primera vez, todo el paquete es cargado en memoria. Por lo tanto, posteriores llamadas al mismo u otros programas dentro de ese paquete realizarán un acceso a memoria en lugar de a disco mejorando así el rendimiento. Esto no sucede con procedimientos y funciones estándares.

- **Permite la "Sobrecarga de funciones" (Overloading).**
  PL/SQL nos permite que varios procedimientos o funciones almacenadas, declaradas dentro de un mismo paquete, tengan el mismo nombre. Esto nos es muy útil cuando necesitamos que los programas puedan aceptar parámetros que contengan diferentes tipos de datos en diferentes instancias. En este caso Oracle ejecutará la "versión" de

la función o procedimiento cuyo encabezado se corresponda con la lista de parámetros recibidos.

## 17.2. PL-SQL Oracle desde Java

De forma resumida, se vera como devolver valores simples y conjuntos de resultados desde procedimientos almacenados en paquetes.

**Valores simples**

Supongamos un procedimiento usa la sentencia SELECT COUNT(*) para contar registros. El resultado se puede guardar en nuestra variable temporal vTotal, o se puede guardar en el parámetro de salida pTotal, el cual podremos leer desde Java.

```
SELECT COUNT(*) INTO vTotal
      FROM tabla WHERE campo = datoABuscar;
...
SELECT COUNT(*) INTO pTotal
      FROM tabla WHERE campo = datoABuscar;
```

**Conjunto de valores**

Tenemos que usar cursores para retornar conjuntos de datos a Java (por medio de un ResultSet).

Si queremos hacer una consulta y devolver sus resultados en un ResultSet, tenemos que utilizar el parámetro de salida de tipo CURSOR como medio de transporte. Esto se hace:

Si es un procedimiento no va en un paquete, el cursor de salida que recoge los datos del procedimiento llevara el tipo de dato **Sys_Refcursor**.

Ejemplo:

```
create or replace Procedure ProGetPrueba2(prfCursor out Sys_Refcursor, pcodigo
out Number, pnNumber  In Number)  Is
   V_EMPNO emple.emp_no%TYPE;
  V_APELLIDO emple.apellido%TYPE;
Begin
     --Valor 1: funciona OK, Valor diferente a 1 retorna error.
     if  pnNumber = 1 then
              Select 1  into pcodigo
              From   DUAL;
              Open prfCursor For 'SELECT emp_no,apellido FROM emple';
        else
          RAISE_APPLICATION_ERROR (-20000, 'El valor no puede ser distinto de
uno');
     end if;
End ProGetPrueba2;
```

Si el procedimiento pertenece a un paquete, declararemos en la cabecera del paquete un tipo nuevo que va a referenciar a un CURSOR. Esto se hace usando cursores genéricos los cuales permiten crear el cursor y en tiempo de ejecución se asignará la consulta que el cursor debe manejar. Para usar cursores genéricos utilizaremos OPEN <nombre-cursor> FOR <consulta>

Sintaxis para la declaración del tipo de dato en la cabecera del paquete:

```
TYPE <NOMBRE_TIPO> IS REF CURSOR;
VARIABLE <NOMBRE_TIPO>
```

Sintaxis de implementación del cursor genérico:

```
OPEN <VARIABLE> FOR
      SELECT campo FROM tabla [WHERE campo = otroDato];
```

Ejemplo. En la cabecera del paquete tenemos definido el tipo vCursor

```
TYPE vCursor IS REF CURSOR;
```

En la cabecera del paquete en la definición del procedimiento hemos de especificado el nombre del mismo y 4 parámetros que acepta: 2 de entrada y 2 de salida.

```
CREATE OR REPLACE PACKAGE  nombrePaquete IS
PROCEDURE procedimiento (
  datoABuscar IN tabla.campo%TYPE,
  otroDato IN tabla.campo%TYPE,
  vTotal OUT NUMBER,
  vUserCursor OUT vCursor
)
```

El cuerpo del procedimiento es:

```
CREATE OR REPLACE PACKAGE BODY nombrePaquete IS
PROCEDURE procedimiento (
  datoABuscar IN tabla.campo%TYPE,
  otroDato IN tabla.campo%TYPE,
  vTotal OUT NUMBER,
  vUserCursor OUT vCursor
) IS
vTotal NUMBER;
BEGIN
      SELECT COUNT(*) INTO vTotal
            FROM tabla WHERE campo = datoABuscar;
      IF vTotal = 0 THEN
            INSERT INTO tabla (campo) VALUES (otroDato);
      ELSE
            UPDATE tabla SET campo = otroDato WHERE campo = datoABuscar;
      END IF;
      SELECT COUNT(*) INTO vTotal
            FROM tabla WHERE campo = datoABuscar;
      OPEN vUserCursor FOR
            SELECT campo FROM tabla WHERE campo = otroDato;
END procedimiento;
END nombrePaquete
```