

2012

ASP SOLUTIONS

RECOPILADO POR:
JDAGOBETHM

[TUTORIAL DE PL/SQL]

PL/SQL es un lenguaje dirigido a base de datos proporcionado por Oracle para extender las funciones del SQL estándar agregando otro tipo de instrucciones propios de los lenguajes de programación.

1. INTRODUCCIÓN

SQL es un lenguaje de consulta para los sistemas de bases de datos relacionales, pero que no posee la potencia de los lenguajes de programación.

PL/SQL amplía SQL con los elementos característicos de los lenguajes de programación, variables, sentencias de control de flujo, bucles...

Cuando se desea realizar una aplicación completa para el manejo de una base de datos relacional, resulta necesario utilizar alguna herramienta que soporte la capacidad de consulta del SQL y la versatilidad de los lenguajes de programación tradicionales. PL/SQL es el lenguaje de programación que proporciona **Oracle** para extender el SQL estándar con otro tipo de instrucciones.

¿Qué vamos a necesitar?

Para poder seguir este tutorial correctamente necesitaremos tener los siguientes elementos:

- Una instancia de ORACLE 8i o superior funcionando correctamente.
- Herramientas cliente de ORACLE, en particular SQL*Plus para poder ejecutar los ejemplos.
- Haber configurado correctamente una conexión a ORACLE.

2. PROGRAMACIÓN

Sin embargo, SQL es la herramienta ideal para trabajar con bases de datos. Cuando se desea realizar una aplicación completa para el manejo de una base de datos relacional, resulta necesario utilizar alguna herramienta que soporte la capacidad de consulta del SQL y la versatilidad de los lenguajes de programación tradicionales. PL/SQL es el lenguaje de programación que proporciona Oracle para extender el SQL estándar con otro tipo de instrucciones y elementos propios de los lenguajes de programación. Con PL/SQL vamos a poder programar las unidades de programa de la base de datos ORACLE, estas son:

- Procedimientos almacenados
- Funciones
- Triggers
- Scripts

Pero además PL/SQL nos permite realizar programas sobre las siguientes herramientas de ORACLE:

- Oracle Forms
- Oracle Reports
- Oracle Graphics
- Oracle Application Server

3. FUNDAMENTOS

Primeros pasos con PL/SQL

Para programar en PL/SQL es necesario conocer sus fundamentos.

Como introducción vamos a ver algunos elementos y conceptos básicos del lenguaje.

- PL/SQL no es CASE-SENSITIVE, es decir, no diferencia mayúsculas de minúsculas como otros lenguajes de programación como C o Java. **Sin embargo debemos recordar que ORACLE es CASE-SENSITIVE en la búsqueda de texto.**
- Una línea en PL/SQL contiene grupos de caracteres conocidos como UNIDADES LEXICAS, que pueden ser clasificadas como:
 - DELIMITADORES
 - IDENTIFICADORES
 - LITERALES
 - COMENTARIOS
 - EXPRESIONES
- DELIMITADOR: Es un símbolo simple o compuesto que tiene una función especial en PL/SQL. Estos pueden ser:
 - Operadores Aritméticos
 - Operadores Lógicos
 - Operadores Relacionales
- IDENTIFICADOR: Son empleados para nombrar objetos de programas en PL/SQL así como a unidades dentro del mismo, estas unidades y objetos incluyen:
 - Constantes
 - Cursores
 - Variables
 - Subprogramas
 - Excepciones
 - Paquetes
- LITERAL: Es un valor de tipo numérico, carácter, cadena o lógico no representado por un identificador (es un valor explícito).
- COMENTARIO: Es una aclaración que el programador induce en el código. Son soportados 2 estilos de comentarios, el de línea simple y de multi-línea, para lo cual son empleados ciertos caracteres especiales como son:
 - -- línea simple
 - /* conjunto de líneas */

4. TIPOS DE DATOS

Cada constante y variable tiene un tipo de dato en el cual se especifica el formato de almacenamiento, restricciones y rango de valores validos.

PL/SQL proporciona una variedad predefinida de tipos de datos. Casi todos los tipos de datos manejados por PL/SQL son similares a los soportados por SQL. A continuación se muestran los TIPOS de DATOS más comunes:

- **NUMBER** (*Número*): Almacena números enteros o de punto flotante, virtualmente de cualquier longitud, aunque puede ser especificada la precisión (Número de dígitos) y la escala que es la que determina el número de decimales.

```
-- NUMBER [(precisión, escala)]
saldo NUMBER(16,2);
/* Indica que puede almacenar un valor numérico de 16
   posiciones, 2 de ellas decimales. Es decir, 14 enteros
   y dos decimales */
```

- **CHAR** (*Carácter*): Almacena datos de tipo carácter con una longitud máxima de 32767 y cuyo valor de longitud por default es 1

```
-- CHAR [(longitud_maxima)]
nombre CHAR(20);
/* Indica que puede almacenar valores alfanuméricos de 20 posiciones */
```

- **VARCHAR2** (*Carácter de longitud variable*): Almacena datos de tipo carácter empleando sólo la cantidad necesaria aún cuando la longitud máxima sea mayor.

```
-- VARCHAR2 (longitud_maxima)
nombre VARCHAR2(20);
/* Indica que puede almacenar valores alfanuméricos de hasta 20 posiciones */
/* Cuando la longitud de los datos sea menor de 20 no se rellena con blancos */
```

- **BOOLEAN** (*lógico*): Se emplea para almacenar valores TRUE o FALSE.

```
hay_error BOOLEAN;
```

- **DATE** (*Fecha*): Almacena datos de tipo fecha. Las fechas se almacenan internamente como datos numéricos, por lo que es posible realizar operaciones aritméticas con ellas.
- **Atributos de tipo**: Un atributo de tipo PL/SQL es un modificador que puede ser usado para obtener información de un objeto de la base de datos. El atributo **%TYPE** permite conocer el tipo de una variable, constante o campo de la base de datos. El atributo **%ROWTYPE** permite obtener los tipos de todos los campos de una tabla de la base de datos, de una vista o de un cursor.

- PL/SQL también permite la creación de tipos personalizados (registros) y colecciones (tablas de PL/SQL), que veremos en sus apartados correspondientes.

Existen por supuesto más tipos de datos, la siguiente tabla los muestra:

Tipo de dato / Sintaxis	Oracle 8i	Oracle 9i	Descripción
dec(p, e)	La precisión máxima es de 38 dígitos.	La precisión máxima es de 38 dígitos.	Donde p es la precisión y e la escala. Por ejemplo: dec(3,1) es un número que tiene 2 dígitos antes del decimal y un dígito después del decimal.
decimal(p, e)	La precisión máxima es de 38 dígitos.	La precisión máxima es de 38 dígitos.	Donde p es la precisión y e la escala. Por ejemplo: decimal (3,1) es un número que tiene 2 dígitos antes del decimal y un dígito después del decimal.
double precision			
float			
int			
integer			
numeric(p, e)	La precisión máxima es de 38 dígitos.	La precisión máxima es de 38 dígitos.	Donde p es la precisión y e la escala. Por ejemplo: numeric(7,2) es un número que tiene 5 dígitos antes del decimal y 2 dígitos después del decimal.
number(p, e)	La precisión máxima es de 38 dígitos.	La precisión máxima es de 38 dígitos.	Donde p es la precisión y e la escala. Por ejemplo: number(7,2) es un número que tiene 5 dígitos antes del decimal y 2 dígitos después del decimal.
real			
smallint			
char (tamaño)	Hasta 32767 bytes en PLSQL Hasta 2000 bytes en Oracle 8i.	Hasta 32767 bytes en PLSQL Hasta 2000 bytes en Oracle 9i.	Donde tamaño es el número de caracteres a almacenar. Son cadenas de ancho fijo. Se rellena con espacios.
varchar2 (tamaño)	Hasta 32767 bytes en PLSQL Hasta 4000 bytes en Oracle 8i.	Hasta 32767 bytes en PLSQL Hasta 4000 bytes en Oracle 9i.	Donde tamaño es el número de caracteres a almacenar. Son cadenas de ancho variable.
long	Hasta 2 gigabytes.	Hasta 2 gigabytes.	Son cadenas de ancho variable.
raw	Hasta 32767 bytes en PLSQL Hasta 2000 bytes en Oracle 8i.	Hasta 32767 bytes en PLSQL Hasta 2000 bytes en Oracle 9i.	Son cadenas binarias de ancho variable.

long raw	Hasta 2 gigabytes.	Hasta 2 gigabytes.	Son cadenas binarias de ancho variable.
date	Una fecha entre el 1 de Enero de 4712 A.C. y el 31 de Diciembre de 9999 D.C.	Una fecha entre el 1 de Enero de 4712 A.C. y el 31 de Diciembre de 9999 D.C.	
timestamp (fractional seconds precision)	No soportado por Oracle 8i.	fractional seconds precision debe ser un número entre 0 y 9. (El valor por defecto es 6)	Incluye año, mes día, hora, minutos y segundos. Por ejemplo: timestamp(6)
timestamp (fractional seconds precision) with time zone	No soportado por Oracle 8i.	fractional seconds precision debe ser un número entre 0 y 9. (El valor por defecto es 6)	Incluye año, mes día, hora, minutos y segundos; con un valor de desplazamiento de zona horaria. Por ejemplo: timestamp(5) with time zone
timestamp (fractional seconds precision) with local time zone	No soportado por Oracle 8i.	fractional seconds precision debe ser un número entre 0 y 9. (El valor por defecto es 6)	Incluye año, mes día, hora, minutos y segundos; con una zona horaria expresada como la zona horaria actual. Por ejemplo: timestamp(4) with local time zone
interval year (year precision) to month	No soportado por Oracle 8i.	year precision debe ser un número entre 0 y 9. (El valor por defecto es 2)	Período de tiempo almacenado en años y meses. Por ejemplo: interval year(4) to month
interval day (day precision) to second (fractional seconds precision)	No soportado por Oracle 8i.	day precision debe ser un número entre 0 y 9. (El valor por defecto es 2) fractional seconds precision debe ser un número entre 0 y 9. (El valor por defecto es 6)	Incluye año, mes día, hora, minutos y segundos. Por ejemplo: interval day(2) to second(6)
rowid	El formato del campo rowid es: BBBBBBBB.RRRR.FFFFF donde BBBBBBBB es el bloque en el fichero de la base de datos; RRRR es la fila del bloque; FFFFF es el fichero de la base de datos.	El formato del campo rowid es: BBBBBBBB.RRRR.FFFFF donde BBBBBBBB es el bloque en el fichero de la base de datos; RRRR es la fila del bloque; FFFFF es el fichero de la base de datos.	Datos binarios de ancho fijo. Cada registro de la base de datos tiene una dirección física o rowid.
urowid [tamaño]	Hasta 2000 bytes.	Hasta 2000 bytes.	Rowid universal. Donde tamaño es opcional.
boolean	Válido en PLSQL, este tipo de datos no existe en Oracle 8i.	Válido en PLSQL, este tipo de datos no existe en Oracle 9i.	
nchar (tamaño)	Hasta 32767 bytes en	Hasta 32767 bytes en	Donde tamaño es el número de caracteres a

	PLSQL Hasta 2000 bytes en Oracle 8i.	PLSQL Hasta 2000 bytes en Oracle 9i.	almacenar. Cadena NLS de ancho fijo.
nvarchar2 (tamaño)	Hasta 32767 bytes en PLSQL Hasta 4000 bytes en Oracle 8i.	Hasta 32767 bytes en PLSQL Hasta 4000 bytes en Oracle 9i.	Donde tamaño es el número de caracteres a almacenar. Cadena NLS de ancho variable.
bfile	Hasta 4 gigabytes.	Hasta 4 gigabytes.	Localizadores de archivo apuntan a un objeto binario de sólo lectura fuera de la base de datos.
blob	Hasta 4 gigabytes.	Hasta 4 gigabytes.	Localizadores LOB apuntan a un gran objeto binario dentro de la base de datos.
clob	Hasta 4 gigabytes.	Hasta 4 gigabytes.	Localizadores LOB apuntan a un gran objeto de caracteres dentro de la base de datos.
nclob	Hasta 4 gigabytes.	Hasta 4 gigabytes.	Localizadores LOB apuntan a un gran objeto NLS de caracteres dentro de la base de datos.

5. OPERADORES

La siguiente tabla ilustra los operadores de PL/SQL.

Tipo de operador	Operadores
Operador de asignación	:= (dos puntos + igual)
Operadores aritméticos	+ (suma) - (resta) * (multiplicación) / (división) ** (exponente)
Operadores relacionales o de comparación	= (igual a) <> (distinto de) < (menor que) > (mayor que) >= (mayor o igual a) <= (menor o igual a)
Operadores lógicos	AND (y lógico) NOT (negación) OR (o lógico)
Operador de concatenación	

6. ESTRUCTURAS DE CONTROL

6.1. ESTRUCTURAS DE CONTROL DE FLUJO

En PL/SQL solo disponemos de la estructura condicional IF. Su sintaxis se muestra a continuación:

```
IF (expression) THEN
    -- Instructions
ELSIF (expression) THEN
    -- Instructions
ELSE
    -- Instructions
END IF;
```

Un aspecto a tener en cuenta es que la instrucción condicional anidada es *ELSIF* y no *ELSEIF*.

6.2. SENTENCIA GOTO

PL/SQL dispone de la sentencia GOTO. La sentencia GOTO desvía el flujo de ejecución a una determinada etiqueta.

En PL/SQL las etiquetas se indican del siguiente modo: << *etiqueta* >>

El siguiente ejemplo ilustra el uso de GOTO.

```
DECLARE
    flag NUMBER;
BEGIN
    flag := 1;
    IF (flag = 1) THEN
        GOTO paso2;
    END IF;
    <<paso1>>
        dbms_output.put_line('Ejecucion de paso 1');
    <<paso2>>
        dbms_output.put_line('Ejecucion de paso 2');
END;
```


6.3. BUCLES

En PL/SQL tenemos a nuestra disposición los siguientes iteradores o bucles:

- LOOP
- WHILE
- FOR

El bucle **LOOP**, se repite tantas veces como sea necesario hasta que se fuerza su salida con la instrucción **EXIT**. Su sintaxis es la siguiente:

```
LOOP
  -- Instructions'

  IF (expression) THEN
    -- Instructions
    EXIT;
  END IF;
END LOOP;
```

El bucle **WHILE**, se repite mientras que se cumpla *expresión*.

```
WHILE (expression) LOOP
  -- Instructions'
END LOOP;
```

El bucle **FOR**, se repite tanta veces como le indiquemos en los identificadores *inicio* y *final*.

```
FOR contador IN [REVERSE] inicio..final LOOP
  -- Instrucciones
END LOOP;
```

En el caso de especificar **REVERSE** el bucle se recorre en sentido inverso.

7. BLOQUES

Un programa de PL/SQL está compuesto por bloques. Un programa está compuesto como mínimo de un bloque. Los bloques de PL/SQL pueden ser de los siguientes tipos:

- Bloques anónimos
- Subprogramas

Estructura de un Bloque

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

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

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

```
[ declare | is | as ]
    /*Parte declarativa*/

begin
    /*Parte de ejecución*/

[ exception ]
    /*Parte de excepciones*/

end;
```

De las anteriores partes, únicamente la sección de ejecución es obligatoria, que quedaría delimitada entre las cláusulas **BEGIN** y **END**.

Veamos un ejemplo de bloque PL/SQL muy genérico. Se trata de un bloque anónimo, es decir no lo identifica ningún nombre. Los bloques anónimos identifican su parte declarativa con la palabra reservada **DECLARE**.

```
DECLARE
    /*Parte declarativa*/

    nombre_variable DATE;

BEGIN
    /*Parte de ejecucion
    * Este código asigna el valor de la columna "nombre_columna"
    * a la variable identificada por "nombre_variable"
    */

    SELECT SYSDATE
    INTO nombre_variable
    FROM DUAL;

EXCEPTION
    /*Parte de excepciones*/
    WHEN OTHERS THEN
        dbms_output.put_line('Se ha producido un error');

END;
```

A continuación vamos a ver cada una de estas secciones

Sección de Dedaración de Variables

En esta parte se dedaran las variables que va a necesitar nuestro programa. Una variable se dedara asignándole un nombre o "identificador" seguido del tipo de valor que puede contener. También se declaran cursores, de gran utilidad para la consulta de datos, y excepciones definidas por el usuario. También podemos especificar si se trata de una constante, si puede contener valor nulo y asignar un valor inicial.

La sintaxis genérica para la declaración de constantes y variables es:

```
nombre_variable [CONSTANT] <tipo_dato> [NOT NULL][:=valor_inicial]
```

Donde:

- `tipo_dato`: es el tipo de dato que va a poder almacenar la variable, este puede ser cualquiera de los tipos soportados por ORACLE, es decir **NUMBER**, **DATE**, **CHAR**, **VARCHAR**, **VARCHAR2**, **BOOLEAN** ... Además para algunos tipos de datos (NUMBER y VARCHAR) podemos especificar la longitud.
- La cláusula **CONSTANT** indica la definición de una constante cuyo valor no puede ser modificado. Se debe induir la inicialización de la constante en su declaración.

- La cláusula NOT NULL impide que a una variable se le asigne el valor nulo, y por tanto debe inicializarse a un valor diferente de NULL.
- Las variables que no son inicializadas toman el valor inicial NULL.
- La inicialización puede incluir cualquier expresión legal de PL/SQL, que lógicamente debe corresponder con el tipo del identificador definido.
- Los tipos escalares incluyen los definidos en SQL más los tipos VARCHAR y BOOLEAN. Este último puede tomar los valores TRUE, FALSE y NULL, y se suele utilizar para almacenar el resultado de alguna operación lógica. VARCHAR es un sinónimo de CHAR.
- También es posible definir el tipo de una variable o constante, dependiendo del tipo de otro identificador, mediante la utilización de las cláusulas %TYPE y %ROWTYPE. Mediante la primera opción se define una variable o constante escalar, y con la segunda se define una variable fila, donde identificador puede ser otra variable fila o una tabla. Habitualmente se utiliza %TYPE para definir la variable del mismo tipo que tenga definido un campo en una tabla de la base de datos, mientras que %ROWTYPE se utiliza para declarar variables utilizando cursores.

Ejemplos:

Estructura de un bloque anónimo.

```
DECLARE
    /* Se declara la variable de tipo VARCHAR2(15) identificada por v_location y se le asigna
       el valor "Granada" */
    v_location VARCHAR2(15) := 'Granada';

    /* Se declara la constante de tipo NUMBER identificada por PI y se le asigna
       el valor 3.1416 */
    PI CONSTANT NUMBER := 3.1416;

    /* Se declara la variable del mismo tipo que tenga el campo nombre de la tabla
       tabla_empleados identificada por v_nombre y no se le asigna ningún valor */
    v_nombre tabla_empleados.nombre %TYPE;

    /* Se declara la variable del tipo registro correspondiente a un supuesto cursor, llamado
       micursor, identificada por reg_datos */
    reg_datos micursor %ROWTYPE;

BEGIN
    /* Parte de ejecución */

EXCEPTION
    /* Parte de excepciones */

END;
```

Estructura de un subprograma:

```
CREATE PROCEDURE simple_procedure IS
    /* Se declara la variable de tipo VARCHAR2(15) identificada por v_location y se le asigna el valor
    "Granada" */
    v_location VARCHAR2(15) := 'Granada';

    /*Se declara la constante de tipo NUMBER identificada por PI y se le asigna el valor 3.1416*/
    PI CONSTANT NUMBER := 3.1416;

    /*Se declara la variable del mismo tipo que tenga el campo nombre de la tabla
    tabla_empleados identificada por v_nombre y no se le asigna ningún valor */
    v_nombre tabla_empleados.nombre%TYPE;

    /*Se declara la variable del tipo registro correspondiente a un supuesto cursor, llamado
    micursor, identificada por reg_datos*/
    reg_datos micursor%ROWTYPE;

BEGIN
    /*Parte de ejecución*/

EXCEPTION
    /*Parte de excepciones*/

END;
```

8. CURSORES

PL/SQL utiliza cursores para gestionar las instrucciones **SELECT**. Un cursor es un conjunto de registros devuelto por una instrucción SQL. Técnicamente los cursores son fragmentos de memoria que reservados para procesar los resultados de una consulta **SELECT**.

Podemos distinguir dos tipos de cursores:

- **Cursores implícitos.** Este tipo de cursores se utiliza para operaciones **SELECT INTO**. Se usan cuando la consulta devuelve un único registro.
- **Cursores explícitos.** Son los cursores que son declarados y controlados por el programador. Se utilizan cuando la consulta devuelve un conjunto de registros. Ocasionalmente también se utilizan en consultas que devuelven un único registro por razones de eficiencia. Son más rápidos.

Un cursor se define como cualquier otra variable de PL/SQL y debe nombrarse de acuerdo a los mismos convenios que cualquier otra variable. Los cursores implícitos no necesitan declaración.

El siguiente ejemplo declara un cursor explícito:

```
declare
  cursor c_paises is
    SELECT CO_PAIS, DESCRIPCION
    FROM PAISES;
begin
  /* Sentencias del bloque ... */
end;
```

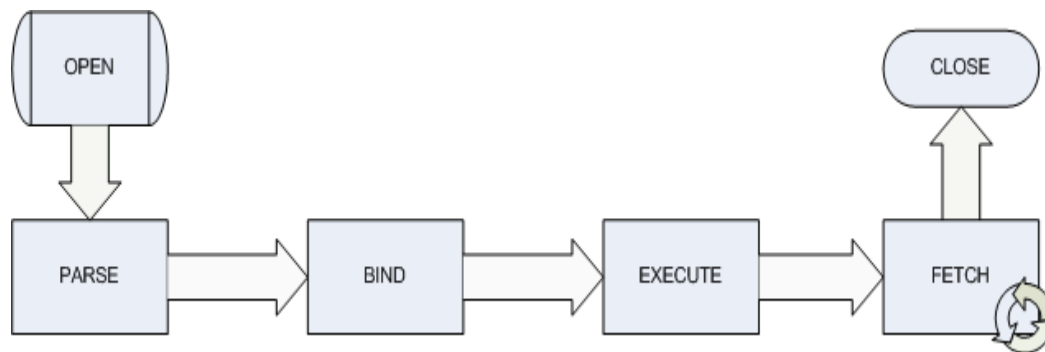
Para procesar instrucciones SELECT que devuelvan más de una fila, son necesarios cursores explícitos combinados con una estructura de bloque.

Un cursor admite el uso de parámetros. Los parámetros deben declararse junto con el cursor.

El siguiente ejemplo muestra la declaración de un cursor con un parámetro, identificado por p_continente.

```
declare
  cursor c_paises (p_continente IN VARCHAR2) is
    SELECT CO_PAIS, DESCRIPCION
    FROM PAISES
    WHERE CONTINENTE = p_continente;
begin
  /* Sentencias del bloque ... */
end;
```

El siguiente diagrama representa como se procesa una instrucción SQL a través de un cursor.



Fases para procesar una instrucción SQL

8.1. CURSORES IMPLÍCITOS

Declaración de cursores implícitos.

Los cursores implícitos se utilizan para realizar consultas **SELECT** que devuelven un único registro.

Deben tenerse en cuenta los siguientes puntos cuando se utilizan cursores implícitos:

- Con cada cursor implícito debe existir la palabra clave **INTO**.
- Las variables que reciben los datos devueltos por el cursor tienen que contener el mismo tipo de dato que las columnas de la tabla.
- Los cursores implícitos solo pueden devolver una única fila. En caso de que se devuelva más de una fila (o ninguna fila) se producirá una excepción. No se preocupe si aún no sabe que es una excepción, le valdrá conocer que es el medio por el que PL/SQL gestiona los errores.

El siguiente ejemplo muestra un cursor implícito:

```
declare
vdescripcion VARCHAR2(50);
begin
    SELECT DESCRIPCION
    INTO vdescripcion
    from PAISES
    WHERE CO_PAIS = 'ESP';

    dbms_output.put_line('La lectura del cursores: ' ||
vdescripcion);
end;
```

La salida del programa generaría la siguiente línea:

```
La lectura del cursor es: ESPAÑA
```

Excepciones asociadas a los cursores implícitos.

Los cursores implícitos sólo pueden devolver una fila, por lo que pueden producirse determinadas excepciones. Las más comunes que se pueden encontrar son **no_data_found** y **too_many_rows**. La siguiente tabla explica brevemente estas excepciones.

Excepción	Explicación
NO_DATA_FOUND	Se produce cuando una sentencia SELECT intenta recuperar datos pero ninguna fila satisface sus condiciones. Es decir, cuando "no hay datos"
TOO_MANY_ROWS	Dado que cada cursor implícito sólo es capaz de recuperar una fila, esta excepción detecta la existencia de más de una fila.

8.2. CURSORES EXPLICITOS

Declaración de cursores explícitos

Los cursores explícitos se emplean para realizar consultas SELECT que pueden devolver cero filas, o más de una fila.

Para trabajar con un cursor explícito necesitamos realizar las siguientes tareas:

- Declarar el cursor.
- Abrir el cursor con la instrucción OPEN.
- Leer los datos del cursor con la instrucción FETCH.
- Cerrar el cursor y liberar los recursos con la instrucción CLOSE.

Para declarar un cursor debemos emplear la siguiente sintaxis:

```
CURSOR nombre_cursor IS  
  
instrucción_SELECT
```

También debemos declarar los posibles parámetros que requiera el cursor:

```
CURSOR nombre_cursor(param1 tipo1, ..., paramN tipoN) IS  
  
instrucción_SELECT
```


Para abrir el cursor

```
OPEN nombre_cursor;  
  
o bien (en el caso de un cursor con parámetros)  
  
OPEN nombre_cursor(valor1, valor2, ..., valorN);
```

Para recuperar los datos en variables PL/SQL

```
FETCH nombre_cursor INTO lista_variables;  
  
-- o bien ...  
  
FETCH nombre_cursor INTO registro_PL/SQL;
```

Para cerrar el cursor:

```
CLOSE nombre_cursor;
```

El siguiente ejemplo ilustra el trabajo con un cursor explícito. Hay que tener en cuenta que al leer los datos del cursor debemos hacerlo sobre variables del mismo tipo de datos de la tabla (o tablas) que trata el cursor.

```
DECLARE  
  CURSOR cpaíses  
  IS  
    SELECT CO_PAIS, DESCRIPCION, CONTINENTE  
    FROM PAISES;  
  
  co_pais VARCHAR2(3);  
  descripcion VARCHAR2(50);  
  continente VARCHAR2(25);  
BEGIN  
  OPEN cpaíses;  
  FETCH cpaíses INTO co_pais, descripcion, continente;  
  CLOSE cpaíses;  
END;
```

Podemos simplificar el ejemplo utilizando el atributo de tipo %ROWTYPE sobre el cursor.

```
DECLARE
  CURSOR cpaíses
  IS
    SELECT CO_PAIS, DESCRIPCION, CONTINENTE
    FROM PAISES;

  registro cpaíses%ROWTYPE;
BEGIN
  OPEN cpaíses;
  FETCH cpaíses INTO registro;
  CLOSE cpaíses;
END;
```

El mismo ejemplo, pero utilizando parámetros:

```
DECLARE
  CURSOR cpaíses (p_continente VARCHAR2)
  IS
    SELECT CO_PAIS, DESCRIPCION, CONTINENTE
    FROM PAISES
    WHERE CONTINENTE = p_continente;

  registro cpaíses%ROWTYPE;
BEGIN
  OPEN cpaíses('EUROPA');
  FETCH cpaíses INTO registro;
  CLOSE cpaíses;
END;
```

Cuando trabajamos con cursores debemos considerar:

- Cuando un cursor está cerrado, no se puede leer.
- Cuando leemos un cursor debemos comprobar el resultado de la lectura utilizando los atributos de los cursores.
- Cuando se cierra el cursor, es ilegal tratar de usarlo.
- Es ilegal tratar de cerrar un cursor que ya está cerrado o no ha sido abierto

Atributos de cursores

Toman los valores TRUE, FALSE o NULL dependiendo de la situación:

Atributo	Antes de abrir	Al abrir	Durante recuperación	Al finalizar la recuperación	Después de cerrar
%NOTFOUND	ORA-1001	NULL	FALSE	TRUE	ORA-1001
%FOUND	ORA-1001	NULL	TRUE	FALSE	ORA-1001
%ISOPEN	FALSE	TRUE	TRUE	TRUE	FALSE
%ROWCOUNT	ORA-1001	0	*	**	ORA-1001

* Número de registros que ha recuperado hasta el momento

** Número de total de registros

Manejo del cursor

Por medio de ciclo LOOP podemos iterar a través del cursor. Debe tenerse cuidado de agregar una condición para salir del bucle: Vamos a ver varias formas de iterar a través de un cursor. La primera es utilizando un bucle LOOP con una sentencia EXIT condicionada:

```
OPEN nombre_cursor;

LOOP

FETCH nombre_cursor INTO lista_variables;

EXIT WHEN nombre_cursor%NOTFOUND;

    /* Procesamiento de los registros recuperados */

END LOOP;

CLOSE nombre_cursor;
```

Aplicada a nuestro ejemplo anterior:

```
DECLARE
  CURSOR cpaíses
  IS
    SELECT CO_PAIS, DESCRIPCION, CONTINENTE
    FROM PAISES;

  co_pais VARCHAR2(3);
  descripcion VARCHAR2(50);
  continente VARCHAR2(25);
BEGIN
  OPEN cpaíses;
  LOOP
    FETCH cpaíses INTO co_pais,descripcion,continente;
    EXIT WHEN cpaíses%NOTFOUND;
    dbms_output.put_line(descripcion);
  END LOOP;
  CLOSE cpaíses;
END;
```

Otra forma es por medio de un bucle WHILE LOOP. La instrucción FETCH aparece dos veces.

```
DECLARE
  CURSOR cpaíses
  IS
    SELECT CO_PAIS, DESCRIPCION, CONTINENTE
    FROM PAISES;

  co_pais VARCHAR2(3);
  descripcion VARCHAR2(50);
  continente VARCHAR2(25);
BEGIN
  OPEN cpaíses;
  FETCH cpaíses INTO co_pais,descripcion,continente;
  WHILE cpaíses%found
  LOOP
    dbms_output.put_line(descripcion);
    FETCH cpaíses INTO co_pais,descripcion,continente;
  END LOOP;
  CLOSE cpaíses;
END;
```

Por último podemos usar un bucle FOR LOOP. Es la forma más corta ya que el cursor es implícitamente se ejecutan las instrucciones OPEN, FETCH y CLOSE.

```
FOR variable IN nombre_cursor LOOP

    /* Procesamiento de los registros recuperados */

END LOOP;
```

```
BEGIN
  FOR REG IN (SELECT * FROM PAISES)
  LOOP
    dbms_output.put_line(reg.descripcion);
  END LOOP;
END;
```

8.3. CURSORES DE ACTUALIZACIÓN

Declaración y utilización de 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 <cursor_name>**.

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

El siguiente ejemplo muestra el uso de un cursor de actualización:

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

    FETCH cpaíses INTO co_pais,descripcion,continente;
  END LOOP;
  CLOSE cpaíses;
  COMMIT;

END;
```

Cuando trabajamos con cursores de actualización debemos tener en cuenta las siguientes consideraciones:

- Los cursores de actualización generan bloqueos en la base de datos.

9. EXCEPCIONES

Manejo de excepciones

En PL/SQL una advertencia o condición de error es llamada una excepción.

Las excepciones se controlan dentro de su propio bloque. La estructura de bloque de una excepción se muestra a continuación.

```
DECLARE
  -- Declaraciones
BEGIN
  -- Ejecucion
EXCEPTION
  -- Excepcion
END;
```

Cuando ocurre un error, se ejecuta la porción del programa marcada por el bloque **EXCEPTION**, transfiriéndose el control a ese bloque de sentencias. El siguiente ejemplo muestra un bloque de excepciones que captura las excepciones **NO_DATA_FOUND** y **ZERO_DIVIDE**. Cualquier otra excepción será capturada en el bloque **WHEN OTHERS THEN**.

```
DECLARE
-- Declaraciones
BEGIN
-- Ejecucion
EXCEPTION
WHEN NO_DATA_FOUND THEN
-- Se ejecuta cuando ocurre una excepcion de tipo NO_DATA_FOUND
WHEN ZERO_DIVIDE THEN
-- Se ejecuta cuando ocurre una excepcion de tipo ZERO_DIVIDE

WHEN OTHERS THEN
-- Se ejecuta cuando ocurre una excepcion de un tipo no tratado
-- en los bloques anteriores
END;
```

Como ya hemos dicho cuando ocurre un error, se ejecuta el bloque **EXCEPTION**, transfiriéndose el control a las sentencias del bloque. Una vez finalizada la ejecución del bloque de **EXCEPTION** no se continúa ejecutando el bloque anterior.

Si existe un bloque de excepción apropiado para el tipo de excepción se ejecuta dicho bloque. Si no existe un bloque de control de excepciones adecuado al tipo de excepción se ejecutará el bloque de excepción **WHEN OTHERS THEN** (si existe!). **WHEN OTHERS** debe ser el último manejador de excepciones.

Las excepciones pueden ser definidas en forma interna o explícitamente por el usuario. Ejemplos de excepciones definidas en forma interna son la división por cero y la falta de memoria en tiempo de ejecución. Estas mismas condiciones excepcionales tienen sus propio tipos y pueden ser referenciadas por ellos: **ZERO_DIVIDE** y **STORAGE_ERROR**.

Las excepciones definidas por el usuario deben ser alcanzadas explícitamente utilizando la sentencia **RAISE**.

Con las excepciones se pueden manejar los errores cómodamente sin necesidad de mantener múltiples chequeos por cada sentencia escrita. También provee claridad en el código ya que permite mantener las rutinas correspondientes al tratamiento de los errores de forma separada de la lógica del negocio.

Excepciones predefinidas

PL/SQL proporciona un gran número de excepciones predefinidas que permiten controlar las condiciones de error más habituales.

Las excepciones predefinidas no necesitan ser declaradas. Simplemente se utilizan cuando estas son lanzadas por algún error determinado.

La siguiente es la lista de las excepciones predeterminadas por PL/SQL y una breve descripción de cuándo son accionadas:

Excepcion	Se ejecuta ...	SQLCODE
ACCESS_INTO_NULL	El programa intentó asignar valores a los atributos de un objeto no inicializado	-6530
COLLECTION_IS_NULL	El programa intentó asignar valores a una tabla anidada a ún no inicializada	-6531
CURSOR_ALREADY_OPEN	El programa intentó abrir un cursor que ya se encontraba abierto. Recuerde que un cursor de ciclo FOR automáticamente lo abre y ello no se debe especificar con la sentencia OPEN	-6511
DUP_VAL_ON_INDEX	El programa intentó almacenar valores duplicados en una columna que se mantiene con restricción de integridad de un índice único (unique index)	-1
INVALID_CURSOR	El programa intentó efectuar una operación no válida sobre un cursor	-1001
INVALID_NUMBER	En una sentencia SQL, la conversión de una cadena de caracteres hacia un número falla cuando esa cadena no representa un número válido	-1722
LOGIN_DENIED	El programa intentó conectarse a Oracle con un nombre de usuario o password inválido	-1017
NO_DATA_FOUND	Una sentencia SELECT INTO no devolvió valores o el programa referenció un elemento no inicializado en una tabla indexada	100
NOT_LOGGED_ON	El programa efectuó una llamada a Oracle sin estar conectado	-1012
PROGRAM_ERROR	PL/SQL tiene un problema interno	-6501
ROWTYPE_MISMATCH	Los elementos de una asignación (el valor a asignar y la variable que lo contendrá) tienen tipos incompatibles. También se presenta este error cuando un parámetro pasado a un subprograma no es del tipo esperado	-6504
SELF_IS_NULL	El parámetro SELF (el primero que es pasado a un método MEMBER) es nulo	-30625
STORAGE_ERROR	La memoria se terminó o está corrupta	-6500
SUBSCRIPT_BEYOND_COUNT	El programa está tratando de referenciar un elemento de un arreglo indexado que se encuentra en una posición más grande que el número real de elementos de la colección	-6533
SUBSCRIPT_OUTSIDE_LIMIT	El programa está referenciando un elemento de un arreglo utilizando un número fuera del rango permitido (por ejemplo, el elemento "-1")	-6532
SYS_INVALID_ROWID	La conversión de una cadena de caracteres hacia un tipo rowid falló porque la cadena no representa un número	-1410
TIMEOUT_ON_RESOURCE	Se excedió el tiempo máximo de espera por un recurso en Oracle	-51
TOO_MANY_ROWS	Una sentencia SELECT INTO devuelve más de una fila	-1422
VALUE_ERROR	Ocurrió un error aritmético, de conversión o truncamiento. Por ejemplo, sucede cuando se intenta calzar un valor muy grande dentro de una variable más pequeña	-6502
ZERO_DIVIDE	El programa intentó efectuar una división por cero	-1476

Excepciones definidas por el usuario

PL/SQL permite al usuario definir sus propias excepciones, las que deberán ser declaradas y lanzadas explícitamente utilizando la sentencia **RAISE**.

Las excepciones deben ser declaradas en el segmento **DECLARE** de un bloque, subprograma o paquete. Se declara una excepción como cualquier otra variable, asignándole el tipo **EXCEPTION**. Las mismas reglas de alcance aplican tanto sobre variables como sobre las excepciones.

```

DECLARE
-- Declaraciones

    MyExcepcion EXCEPTION;
BEGIN
-- Ejecucion
EXCEPTION
-- Excepcion
END;
```


Reglas de Alcance

Una excepción es válida dentro de su ámbito de alcance, es decir el bloque o programa donde ha sido declarada. Las excepciones predefinidas son siempre válidas. Como las variables, una excepción declarada en un bloque es local a ese bloque y global a todos los sub-bloques que comprende.

La sentencia RAISE

La sentencia **RAISE** permite lanzar una excepción en forma explícita. Es posible utilizar esta sentencia en cualquier lugar que se encuentre dentro del alcance de la excepción.

```
DECLARE
-- Declaramos una excepcion identificada por VALOR_NEGATIVO

    VALOR_NEGATIVO EXCEPTION;

    valor NUMBER;
BEGIN
-- Ejecucion

    valor := -1;

    IF valor < 0 THEN

        RAISE VALOR_NEGATIVO;

    END IF;

EXCEPTION
-- Excepcion

    WHEN VALOR_NEGATIVO THEN

        dbms_output.put_line('El valor no puede ser negativo');

END;
```

Con la sentencia **RAISE** podemos lanzar una excepción definida por el usuario o predefinida, siendo el comportamiento habitual lanzar excepciones definidas por el usuario.

Recordar la existencia de la excepción **OTHERS**, que simboliza cualquier condición de excepción que no ha sido declarada. Se utiliza comúnmente para controlar cualquier tipo de error que no ha sido previsto. En ese caso, es común observar la sentencia **ROLLBACK** en el grupo de sentencias de la excepción o alguna de las funciones **SQLCODE** – **SQLERRM**, que se detallan en el próximo punto.

Uso de SQLCODE y SQLERRM

Al manejar una excepción es posible usar las funciones predefinidas **SQLCODE** y **SQLERRM** para aclarar al usuario la situación de error acontecida. **SQLCODE** devuelve el número del error de Oracle y un 0 (cero) en caso de éxito al ejecutarse una sentencia SQL.

Por otra parte, **SQLERRM** devuelve el correspondiente mensaje de error.

Estas funciones son muy útiles cuando se utilizan en el bloque de excepciones, para aclarar el significado de la excepción **OTHERS**.

Estas funciones no pueden ser utilizadas directamente en una sentencia SQL, pero sí se puede asignar su valor a alguna variable de programa y luego usar esta última en alguna sentencia.

```
DECLARE
  err_num NUMBER;
  err_msg VARCHAR2(255);
  result NUMBER;
BEGIN
  SELECT 1/0 INTO result
  FROM DUAL;

  EXCEPTION
  WHEN OTHERS THEN

    err_num := SQLCODE;
    err_msg := SQLERRM;
    DBMS_OUTPUT.put_line('Error: ' || TO_CHAR(err_num));
    DBMS_OUTPUT.put_line(err_msg);
END;
```

También es posible entregarle a la función **SQLERRM** un número negativo que represente un error de Oracle y ésta devolverá el mensaje asociado.

```
DECLARE
  msg VARCHAR2(255);
BEGIN
  msg := SQLERRM(-1403);
  DBMS_OUTPUT.put_line(MSG);
END;
```

9.1. EXCEPCIONES PERSONALIZADAS

RAISE_APPLICATION_ERROR

En ocasiones queremos enviar un mensaje de error personalizado al producirse una excepción PL/SQL

Para ello es necesario utilizar la instrucción **RAISE_APPLICATION_ERROR**;

La sintaxis general es la siguiente:

```
RAISE_APPLICATION_ERROR(<error_num>,<mensaje>);
```

Siendo:

- error_num es un entero negativo comprendido entre -20001 y -20999
- mensaje la descripción del error

```
DECLARE
  v_div NUMBER;
BEGIN
  SELECT 1/0 INTO v_div FROM DUAL;
EXCEPTION
  WHEN OTHERS THEN
    RAISE_APPLICATION_ERROR(-20001,'No se puede dividir por cero');
END;
```

9.2 PROPAGACIÓN DE EXCEPCIONES

Una de las características más interesantes de las excepciones es la propagación de excepciones.

Cuando se lanza una excepción, el control se transfiere hasta la sección **EXCEPTION** del bloque donde se ha producido la excepción. Entonces se busca un manejador válido de la excepción (**WHEN <excepcion> THEN**, **WHEN OTHERS THEN**) dentro del bloque actual.

En el caso de que no se encuentre ningún manejador válida el control del programa se desplaza hasta el bloque **EXCEPTION** del bloque que ha realizado la llamada PL/SQL.

Observemos el siguiente bloque de PL/SQL (Notese que se ha añadido una clausula **WHERE 1=2** para provocar una excepción **NO_DATA_FOUND**).

```
DECLARE

fecha DATE;
FUNCTION fn_fecha RETURN DATE
IS
    fecha DATE;
BEGIN
    SELECT SYSDATE INTO fecha
    FROM DUAL
    WHERE 1=2;
    RETURN fecha;
EXCEPTION
    WHEN ZERO_DIVIDE THEN
        dbms_output.put_line('EXCEPCION ZERO_DIVIDE CAPTURADA
                               EN fn_fecha');
END;

BEGIN

    fecha := fn_fecha();
    dbms_output.put_line('La fecha es ' || TO_CHAR(fecha, 'DD/MM/YYYY'));

EXCEPTION
    WHEN NO_DATA_FOUND THEN
        dbms_output.put_line('EXCEPCION NO_DATA_FOUND CAPTURADA EN
                               EL BLOQUE PRINCIPAL');
END;
```

La excepción **NO_DATA_FOUND** se produce durante la ejecución de la función `fn_fecha`, pero como no existe ningún manejador de la excepción en dicha función, la excepción se propaga hasta el bloque que ha realizado la llamada. En ese momento se captura la excepción.

10. SUBPROGRAMAS

Como hemos visto anteriormente los bloques de PL/SQL pueden ser bloques anónimos (scripts) y subprogramas. Los subprogramas son bloques de PL/SQL a los que asignamos un nombre identificativo y que normalmente almacenamos en la propia base de datos para su posterior ejecución; pueden recibir parámetros y pueden ser de varios tipos:

10.1. PROCEDIMIENTOS ALMACENADOS

Un procedimiento es un subprograma que ejecuta una acción específica y que no devuelve ningún valor. Un procedimiento tiene un nombre, un conjunto de parámetros (opcional) y un bloque de código.

La sintaxis de un procedimiento almacenado es la siguiente:

```
CREATE [OR REPLACE]
PROCEDURE <procedure_name> [(<param1>
[IN|OUT|IN OUT] <type>,
                        <param2> [IN|OUT|IN OUT]
<type>, ...)]
IS
-- Declaracion de variables locales
BEGIN
-- Sentencias
[EXCEPTION]
-- Sentencias control de excepcion
END [<procedure_name>];
```

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

La sintaxis es muy parecida a la de un bloque anónimo, salvo porque se reemplaza la sección **DECLARE** por la secuencia **PROCEDURE... IS** en la especificación del procedimiento.

Debemos especificar el tipo de datos de cada parámetro. **Al especificar el tipo de dato del parámetro no debemos especificar la longitud del tipo.**

Los parámetros pueden ser de entrada (**IN**), de salida (**OUT**) o de entrada salida (**IN OUT**). El valor por defecto es **IN**, y se toma ese valor en caso de que no especifiquemos nada.

```
CREATE OR REPLACE
PROCEDURE Actualiza_Saldo(cuenta NUMBER,
                        new_saldo NUMBER)
IS
-- Declaracion de variables locales
BEGIN
-- Sentencias
UPDATE SALDOS_CUENTAS
SET SALDO = new_saldo,
    FX_ACTUALIZACION = SYSDATE
WHERE CO_CUENTA = cuenta;

END Actualiza_Saldo;
```

También podemos asignar un valor por defecto a los parámetros, utilizando la cláusula **DEFAULT** o el operador de asignación (**:=**).

```
CREATE OR REPLACE
PROCEDURE Actualiza_Saldo(cuenta
NUMBER,
                        new_saldo NUMBER
DEFAULT 10 )
IS
-- Declaracion de variables locales
BEGIN
-- Sentencias
UPDATE SALDOS_CUENTAS
    SET SALDO = new_saldo,
    FX_ACTUALIZACION = SYSDATE
WHERE CO_CUENTA = cuenta;

END Actualiza_Saldo;
```

Una vez creado y compilado el procedimiento almacenado podemos ejecutarlo. Si el sistema nos indica que el procedimiento se ha creado con errores de compilación podemos ver estos errores de compilación con la orden **SHOW ERRORS** en SQL *Plus.

Existen dos formas de pasar argumentos a un procedimiento almacenado a la hora de ejecutarlo (en realidad es válido para cualquier subprograma). Estas son:

- **Notación posicional:** Se pasan los valores de los parámetros en el mismo orden en que el procedure los define.

```
BEGIN
    Actualiza_Saldo(200501,2500);
COMMIT;
END;
```

- **Notación nominal:** Se pasan los valores en cualquier orden nombrando explícitamente el parámetro.

```
BEGIN
    Actualiza_Saldo(cuenta =>
200501,new_saldo => 2500);
    COMMIT;
END;
```

10.2. FUNCIONES

La sintaxis para construir funciones es la siguiente:

```
CREATE [OR REPLACE]
FUNCTION <fn_name>[(<param1> IN
<type>, <param2> IN <type>, ...)]
RETURN <return_type>
IS
    result <return_type>;
BEGIN
    return(result);
[EXCEPTION]
    -- Sentencias control de excepcion
END [<fn_name>];
```

El uso de OR REPLACE permite sobrescribir una función existente. Si se omite, y la función existe, se producirá, un error.

La sintaxis de los parámetros es la misma que en los procedimientos almacenados, exceptuando que solo pueden ser de entrada.

Ejemplo:

```
CREATE OR REPLACE
FUNCTION fn_Obtener_Precio(p_producto
VARCHAR2)
RETURN NUMBER
IS
    result NUMBER;
BEGIN
    SELECT PRECIO INTO result
    FROM PRECIOS_PRODUCTOS
    WHERE CO_PRODUCTO = p_producto;
    return(result);
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        return 0;
END ;
```

Si el sistema nos indica que la función se ha creado con errores de compilación podemos ver estos errores de compilación con la orden SHOW ERRORS en SQL *Plus.

Una vez creada y compilada la función podemos ejecutarla de la siguiente forma:

```
DECLARE
  Valor NUMBER;
BEGIN
  Valor := fn_Obtener_Precio('000100');

END;
```

Las funciones pueden utilizarse en sentencias SQL de manipulación de datos (SELECT, UPDATE, INSERT y DELETE):

```
SELECT CO_PRODUCTO,
       DESCRIPCION,
       fn_Obtener_Precio(CO_PRODUCTO)
FROM PRODUCTOS;
```

10.3. TRIGGERS

Declaración de triggers.

Un trigger es un bloque PL/SQL asociado a una tabla, que se ejecuta como consecuencia de una determinada instrucción SQL (una operación DML: INSERT, UPDATE o DELETE) sobre dicha tabla.

La sintaxis para crear un trigger es la siguiente:

```
CREATE [OR REPLACE] TRIGGER <nombre_trigger>
{BEFORE|AFTER}
  {DELETE|INSERT|UPDATE [OF col1, col2, ...,
colN]
  [OR {DELETE|INSERT|UPDATE [OF col1, col2, ...,
colN]...}]}
ON <nombre_tabla>
[FOR EACH ROW [WHEN (<condicion>)]]
DECLARE
  -- variables locales
BEGIN
  -- Sentencias
[EXCEPTION]
  -- Sentencias control de excepcion
END <nombre_trigger>;
```


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

Los triggers pueden definirse para las operaciones INSERT, UPDATE o DELETE, y pueden ejecutarse antes o después de la operación. El modificador BEFORE AFTER indica que el trigger se ejecutará antes o después de ejecutarse la sentencia SQL definida por DELETE INSERT UPDATE. Si incluimos el modificador OF el trigger solo se ejecutará cuando la sentencia SQL afecte a los campos incluidos en la lista.

El alcance de los disparadores puede ser la fila o de orden. El modificador FOR EACH ROW indica que el trigger se disparará cada vez que se realizan operaciones sobre una 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.

La siguiente tabla resume los contenidos anteriores.

Valor	Descripción
INSERT, DELETE, UPDATE	Define qué tipo de orden DML provoca la activación del disparador.
BEFORE , AFTER	Define si el disparador se activa antes o después de que se ejecute la orden.
FOR EACH ROW	Los disparadores con nivel de fila se activan una vez por cada fila afectada por la orden que provocó el disparo. Los disparadores con nivel de orden se activan sólo una vez, antes o después de la orden. Los disparadores con nivel de fila se identifican por la cláusula FOR EACH ROW en la definición del disparador.

La cláusula WHEN sólo es válida para los disparadores con nivel de fila.

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.

El siguiente ejemplo muestra un trigger que inserta un registro en la tabla PRECIOS_PRODUCTOS cada vez que insertamos un nuevo registro en la tabla PRODUTOS:

```
CREATE OR REPLACE TRIGGER TR_PRODUCTOS_01
AFTER INSERT ON PRODUCTOS
FOR EACH ROW
DECLARE
-- local variables
```

```
BEGIN
INSERT INTO PRECIOS_PRODUCTOS
(CO_PRODUCTO,PRECIO,FX_ACTUALIZACION)
VALUES
(:NEW.CO_PRODUCTO,100,SYSDATE);
END ;
```

El trigger se ejecutará cuando sobre la tabla PRODUCTOS se ejecute una sentencia INSERT.

```
INSERT INTO PRODUCTOS
(CO_PRODUCTO, DESCRIPCION)
VALUES
('000100','PRODUCTO 000100');
```

Orden de ejecución de los triggers

Una misma tabla puede tener varios triggers. 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.

Restricciones de los triggers

El cuerpo de un 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:

- Un 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 el disparo es confirmada o cancelada, se confirma o cancela también el trabajo realizado por el disparador.

- Por razones idénticas, ningún procedimiento o función llamado por el disparador puede emitir órdenes de control de transacciones.
- El cuerpo del disparador no puede contener ninguna declaración de variables LONG o LONG RAW

Utilización de: OLD y :NEW

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, DELETE) que ha ejecutado el trigger. Utilizando esta variable podemos acceder a los datos que se están insertando, actualizando o borrando.

La siguiente tabla muestra los valores de OLD y NEW.

ACCION SQL	OLD	NEW
INSERT	No definido; todos los campos toman valor NULL.	Valores que serán insertados cuando se complete la orden.
UPDATE	Valores originales de la fila, antes de la actualización.	Nuevos valores que serán escritos cuando se complete la orden.
DELETE	Valores, antes del borrado de la fila.	No definidos; todos los campos toman el valor NULL.

Los registros OLD y NEW son sólo válidos dentro de los disparadores con nivel de fila.

Podemos usar OLD y NEW como cualquier otra variable PL/SQL

Utilización de predicados de los triggers: INSERTING, UPDATING y DELETING

Dentro de un disparador 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. Estos predicados son INSERTING, UPDATING y DELETING.

Su comportamiento es el siguiente:

Predicado	Comportamiento
INSERTING	TRUE si la orden de disparo es INSERT; FALSE en otro caso.
UPDATING	TRUE si la orden de disparo es UPDATE; FALSE en otro caso.
DELETING	TRUE si la orden de disparo es DELETE; FALSE en otro caso.

10.4. SUBPROGRAMA EN BLOQUES ANÓNIMOS

Dentro de la sección DECLARE de un bloque anónimo podemos declarar funciones y procedimientos almacenados y ejecutarlos desde el bloque de ejecución del script. Este tipo de subprogramas son menos conocidos que los procedimientos almacenados, funciones y triggers, pero son enormemente útiles.

El siguiente ejemplo declara y ejecuta utiliza una función (fn_multiplica_x2).

```
DECLARE
  idx NUMBER;
  FUNCTION fn_multiplica_x2(num NUMBER)
    RETURN NUMBER
  IS
    result NUMBER;
  BEGIN
    result := num *2;
    return result;
  END fn_multiplica_x2;
BEGIN
  FOR idx IN 1..10
  LOOP
    dbms_output.put_line
      ('Llamada a la funcion ... ' || TO_CHAR(fn_multiplica_x2(idx)));
  END LOOP;
END;
```

Nótese que se utiliza la función TO_CHAR para convertir el resultado de la función fn_multiplica_x2 (numérico) en alfanumérico y poder mostrar el resultado por pantalla.