C.F.G.S. DESARROLLO DE APLICACIONES WEB

DISTANCIA

MÓDULO:

BASES DE DATOS

Unidad 7:

Programación de Bases de datos, PL/SQL.

Parte II

ÍNDICE DE CONTENIDOS

OBJETIVOS

Con esta unidad se pretende:

- Controlar las excepciones.
- Crear subprogramas y paquetes almacenados en el catálogo de la base de datos.
- Construir disparadores de base de datos.
- Manejar paquetes estándar existentes en una base de datos.

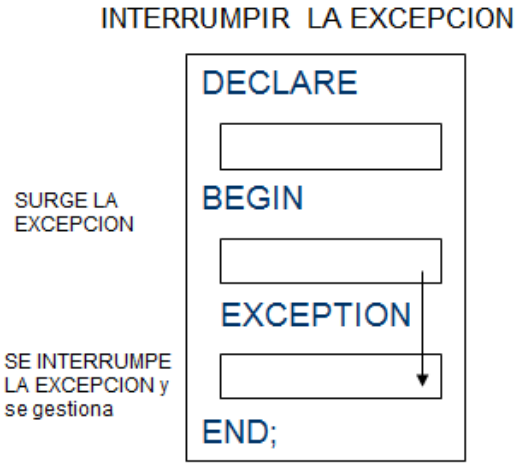
1.- Manejo de errores.

Una excepción es un identificador PL/SQL que surge durante la ejecución del código provocado por un error, o bien porque el programador lo lanza explícitamente.

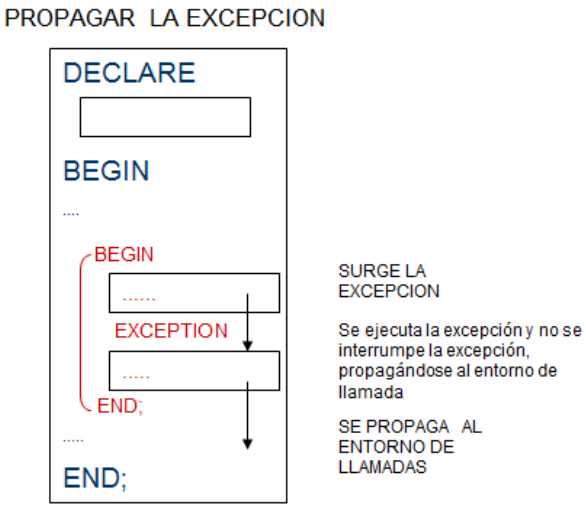
Cuando se detecta un error, una excepción es lanzada, es decir, la ejecución normal se para y el control se transfiere a la parte de manejo de excepciones. La parte de manejo de excepciones es la parte etiquetada como **EXCEPTION** y constará de sentencias para el manejo de dichas excepciones, llamadas **manejadores de excepciones.**

Existen dos formas de gestionar una excepción:

- **Interrumpir la Excepción**: El procesamiento se ramifica al correspondiente manejador de excepciones de la sección de excepciones del propio bloque. El PL/SQL termina satisfactoriamente.



- **Propagar la excepción**: Si la excepción surge en la sección ejecutable (BEGIN....END) del bloque y no hay ningún manejador de excepciones correspondiente, el bloque PL/SQL propaga la excepción al correspondiente bloque padre de forma sucesiva hasta encontrar un manejador.



La sintaxis es la siguiente:

```
DECLARE
     Sección de declaraciones
BEGIN
     Sección de instrucciones
EXCEPTION
     WHEN  <nom_excepcion-1> THEN
          Instrucciones-1;
     WHEN  <nom_excepcion-2> THEN
          Instucciones-2;
     ....
     [WHEN OTHERS THEN
          instrucciones;]
END;
```

Cuando ocurre una determinada excepción, se comprueba el primer WHEN para comprobar si el nombre de la excepción ocurrida coincide con el de dicho WHEN captura; si es así se ejecutan las instrucciones, si no es así se comprueba el siguiente WHEN y así sucesivamente.

Si existe la clausula WHEN OTHERS THEN, entonces las excepciones que no estaban reflejadas en los demás apartados WHEN ejecutan las instrucciones del WHEN OTHERS. Esta cláusula debe ser la última.

1.1.- Tipos de excepciones.

Oracle soporta dos tipos de excepciones:

- **Excepciones provocadas implícitamente**.
  Estas excepciones pueden ser:
  - **Predefinidas por el servidor de Oracle:** Son las más comunes. No se declaran y el servidor Oracle las soporta automáticamente.
    Se disparan automáticamente al producirse determinados errores de Oracle. Cada error Oracle tiene un número, pero las cláusulas WHEN reconocen las excepciones por nombre.
  - **No predefinidas por el Servidor de Oracle:** En este caso son errores estándar reconocidos por el Servidor de Oracle pero no predefinidos. Se declaran en el apartado de declaraciones y el servidor los reporta automáticamente.
- **Provocadas explícitamente por el programador**
  Forzadas por el programador. Se declaran en la sección de declaraciones y se generan de forma explícita, por programa.

1.1.1.- Excepciones predefinidas por el servidor de Oracle.

PL/SQL tiene predefinidas excepciones correspondientes a los errores más comunes:

EXCEPCIONES PREDEFINIDAS EN ORACLE		
Nombre de Excepción	Número Error Oracle	Descripción
NO_DATA_FOUND	ORA-01403	Una SELECT no devolvió filas de datos
TOO_MANY_ROWS	ORA-01422	Una SELECT devolvió más de una fila
INVALID_CURSOR	ORA-01001	Se produjo una operación ilegal sobre un de cursor
ZERO_DIVIDE	ORA-01476	Se intentó dividir entre 0
DUP_VAL_ON_INDEX	ORA-00001	Se intentó insertar un valor duplicado
INVALID_NUMBER	ORA-01722	Fallo de conversión de una cadena de caracteres a números, y la cadena contenía caracteres no numéricos.
LOGIN_DENIED	ORA-01017	Conexión con un usuario y/o contraseña no válidos
ACCESS_INTO_NULL	ORA-09530	Intento de asignar valores a los atributos de un objeto no inicializado
COLLECTION_IS_NULL	ORA-06531	Intento de aplicar a una tabla o array no inicializada cualquier método que no sea EXISTS
CURSOR_ALREADY_OPEN	ORA-06511	Intento de abrir un cursor ya abierto
NOT_LOGGED_ON	ORA-01012	PL/SQL hace una llamada a una BD sin estar conectado
PROGRAM_ERROR	ORA-06501	PL/SQL tiene un problema interno
TIMEOUT_ON_RESOURCE	ORA-00051	Ocurre un Time Out cuando Oracle está esperando un recurso.(tiempo de espera agotado)
VALUE_ERROR	ORA-06502	Error aritmético, conversión, truncado o tamaño
SUBSCRIPT_BEYOND_COUNT	ORA-006533	Acceso a un elemento de una tabla array usando un índice mayor que el número de elementos que tiene

Ejemplo-1

```
BEGIN
      .....
EXCEPTION
       WHEN NO_DATA_FOUND THEN
       DBMS_OUTPUT.PUT_LINE ('La Select no ha devuelto ninguna fila);
       WHEN TOO_MANY_ROWS THEN
       DBMS_OUTPUT.PUT_LINE ('La Select ha devuelto más de 1 fila);
       WHEN OTHERS THEN
       DBMS_OUTPUT.PUT_LINE ('Error no catalogado');
END;
```

1.1.2.- •Excepciones no predefinidas por el Servidor de Oracle:

Son errores reconocidos por el servidor Oracle, y permitirán asociar las acciones a un determinado error de Oracle no predefinido.

Son excepciones producidas por el servidor Oracle que no tienen asociadas ningún nombre, pero si un número y un breve mensaje de error, y a las cuales el usuario puede asignar un nombre para procesarlas después como cualquier excepción predefinida. Por defecto estos errores transfieren el control a la sección **EXCEPTION**, y se tratarán en la cláusula **OTHERS.**

Dispone de

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

```
……..
EXCEPTION
     ….
     WHEN OTHERS THEN
     DMBS_OUTPUT.PUT_LINE ( 'Error: ' || SQLCODE || ' ' || SQLERRM);
     ….
END;
```

Una vez conocido el código de error y el mensaje de error el usuario puede asignarle un nombre y utilizarla como cualquier excepción predefinida, para ello será necesario utilizar la sentencia **PRAGMA EXCEPTION_INIT**

Ejemplo-1

```
DECLARE
     mi_excepcion                    EXCEPTION;     - - declaro una nueva exception
     PRAGMA EXCEPTION_INIT (mi_excepcion, -2292);   -- se asocia a un error de ORACLE
BEGIN
     ….
     …. /*no es necesario levantar la excepción (pues es un error de los detectables por el servidor de
        Oracle), cuando se produzca el error en ORACLE salta a la parte excepción asociada a ese
        código */
     …
EXCEPTION
…..
          WHEN mi_excepcion THEN   /*automáticamente se ejecuta la excepción cuando el  servidor
                                     de Oracle detecte el error -2292*/
          instrucción 1;
          instrucción 2;
          …..
END;
```

1.1.3.- Excepciones definidas por el usuario.

El propio programador declara sus interrupciones asociadas al propio funcionamiento del programa PL/SQL Son errores propios del programa del usuario y que el usuario tratará de una forma concreta. Para su utilización hay que dar 3 pasos:

1.- Declarar la excepción en la sección **DECLARE**

```
nombre_excepcion_usario EXCEPTION;
```

2.- Cuando en la sección ejecutable se detecte el error, se levantará la excepción con la orden **RAISE**

RAISE nom_excepción_usr;

```
RAISE nombre_excepcion_usario
```

3.- Se tratará en la sección EXCEPTION como una excepción más

```
WHEN nom_excepcion_usr THEN
      .....
```

Ejemplo-2

```
DECLARE
    ....
    importe_errorneo   EXCEPTION;
    ...
BEGIN
    ....
    IF precio NOT BETWEEN precio_min AND precio_max THEN
          RAISE importe_erroneo;   - - fuerzo la excepción por programa
    END_IF;
    ....
EXCEPTION
    ....
    WHEN importe_erroneo THEN
          DBMS_OUTPUT.PUT_LINE ('Importe erróneo. Venta  cancelada');
    ....
END;
```

La sentencia **RAISE** se utiliza para activar excepciones previamente declaradas, aunque también admite el nombre de una excepción predefinida. Ya sabemos que las excepciones predefinidas se activan automáticamente, pero si queremos tratar una determinada condición con la rutina que trata un error predefinido podemos activar dicha excepción en la sentencia que analiza la condición.

Ejemplo-3

Si queremos tratar la condición registro.cuota IS NULL igual que la situación NO_DATA_FOUND, pondríamos

```
IF registro.cuota IS NULL THEN
    RAISE NO_DATA_FOUND;
END IF;
```

Ejemplo-4

Con todo tipo de excepciones:

```
DECLARE
    cod_err     number(6);
    vnif                varchar2(10);
    vnom                varchar2(15);
    err_blancos         EXCEPTION; -- para error de programa de usuario
    no_hay_espacio      EXCEPTION;  -- para error del servidor de oracle no declarados
    PRAGMA EXCEPTION_INIT  (no_hay_espacio, -1547); -- se asocia a un error de ORACLE
BEGIN
    SELECT col1, col2 INTO vnif, vnom FROM TABLA;
    IF SUBSTR(vnom,1,1) == ' ' THEN
        RAISE     err_blancos;  -- error de prog de usuario, tengo que forzar la excepción
    END IF;
    UPDATE clientes SET nombre=vnom  WHERE nif=vnif;
EXCEPTION
    WHEN  err_blancos THEN
        INSERT INTO tabla(col1)  VALUES ('ERR blancos');
    WHEN no_hay_espacio THEN
        INSERT INTO tabla(col1) VALUES ('ERR TABLESPACE');
    WHEN NO_DATA_FOUND THEN
        INSERT INTO tabla(col1) VALUES ('ERR no había datos');
    WHEN TOO_MANY_ROWS THEN
        INSERT INTO tabla(col1) VALUES ('ERR demasiados datos');
    WHEN OTHERS THEN
        cod_err :=SQLCODE;
        INSERT INTO tabla(col1) values (cod_err);
END;
```

1.2.- Propagación de Excepciones.

Cuando un subbloque gestiona una excepción, éste termina, y el control se reanuda en el bloque padre que lo contiene, en la sentencia siguiente al **END** del subbloque.

Cuando un bloque genera una excepción, si no tiene un manejador para esa excepción, ésta se propaga a los bloques padres sucesivos hasta que encuentra el correspondiente manejador o hasta que resulta una excepción no gestionada en el entorno Oracle. En el caso de ser gestionada se devuelve el control al bloque padre donde fue gestionada.

Una ventaja de este comportamiento es que es posible incluir sentencias que necesitan su propia y exclusiva gestión en su propio bloque, y dejar las más generales al bloque padre.

Ejemplo-5

```
DECLARE
    err_no_filas        exception;
    err_integridad      exception;
    PRAGMA EXCEPTION_INIT (err_integridad,-2292);
BEGIN
    FOR v_reg IN cursor_emp LOOP
        BEGIN
            SELECT ..... --los errores producidos por el select serán gestionados por el begin externo
            UPDATE.....
                RAISE err_no_filas;
            END IF;
            EXCEPTION
                WHEN err_integridad  THEN .....
                WHEN err_no_filas  THEN....
        END;
    END LOOP;
EXCEPTION
    WHEN NO_DATA_FOUND THEN....
    WHEN TOO_MANY_ROWS THEN...
END;
```

1.3.- Ámbito de las Excepciones.

Reglas a tener en cuenta con las excepciones, en el diseño de aplicaciones:

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

  nombreBloque.nombreExcepcion.

- Las excepciones predefinidas están definidas globalmente. No necesitamos (ni debemos) redefinir las excepciones predefinidas.
- Cuando manejamos una excepción no podemos continuar por la siguiente sentencia a la que la lanzó. Pero sí podemos encerrar la sentencia dentro de un bloque, y ahí capturar las posibles excepciones, para continuar con las siguientes sentencias.
- Cuando ejecutamos varias sentencias seguidas del mismo tipo y queremos capturar alguna posible excepción debida al tipo de sentencia, podemos encapsular cada sentencia en un bloque y manejar en cada bloque la excepción, o podemos utilizar una variable localizadora para saber qué sentencia ha sido la que ha lanzado la excepción (aunque de esta manera no podremos continuar por la siguiente sentencia).
- Si la excepción es capturada por un manejador de excepción apropiado, ésta es tratada y posteriormente el control es devuelto al bloque superior. Si la excepción no es capturada y no existe bloque superior, el control se devolverá al entorno. También puede darse que la excepción sea manejada en un bloque superior a falta de manejadores para ella en los bloques internos, la excepción se propaga de un bloque al superior y así hasta que sea manejada o no queden bloques superiores con lo que el control se devuelve al entorno. Una excepción también puede ser relanzada en un manejador. En la siguiente presentación puedes ver cómo se propagan diferentes excepciones entre diferentes bloques.

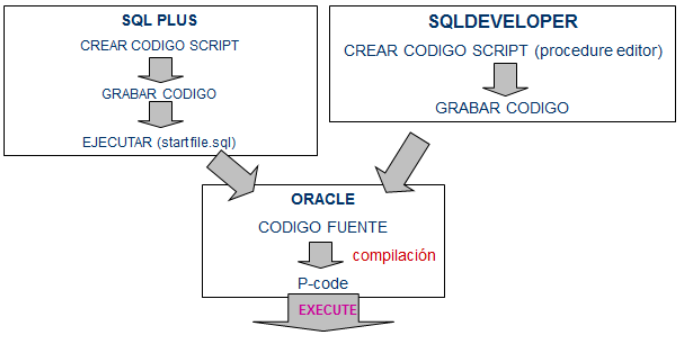
2.- Subprogramas.

Hoy día cualquier lenguaje de programación permite definir diferentes grados de abstracción en sus programas. La abstracción permite a los programadores crear unidades lógicas y posteriormente utilizarlas pensando en qué hace y no en cómo lo hace. La abstracción se consigue utilizando funciones, procedimientos, librerías, objetos, etc.

PL/SQL nos permite definir funciones y procedimientos. Además nos permite agrupar todas aquellas que tengan relación en paquetes. También permite la utilización de objetos.

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

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



Para ejecutar un subprograma tenemos la sentencia **EXECUTE** con la sintaxis siguiente:

```
EXECUTE nombreSubprograma [ valores parámetros];
```

2.1.- Procedimientos.

Un procedimiento es un bloque PL/SQL al que se le asigna un nombre y realiza una determinada tarea. Los procedimientos son compilados y almacenados en la base de datos e invocarlos cuando sea necesario. Pueden invocarse las veces que haga falta, con ello se consigue una reutilización eficiente del código.

Un procedimiento tiene la siguiente sintaxis:

```
CREATE OR REPLACE
    PROCEDURE nom_procedimiento [(lista_parámetros)]      } cabecera
IS/AS
    declaraciones locales;
BEGIN
    instrucciones;                                        } cuerpo
EXCEPTION
    excepciones;
END [<nom_procedimiento>];
```

Vemos que el procedimiento tiene dos partes bien definidas que con la cabecera o especificación y el cuerpo.

- **Cabecera**:
  - **REPLACE** sustituye el procedimiento el procedimiento si ya existiera.
  - Se le asigna un nombre y opcionalmente los parámetros a través de los cuales se pasaran valores cuando se le invoque.
  - Si existe más de un parámetro, van separados por comas.
  - Los parámetros tienen el siguiente formato:

```
nombreParametro [IN |OUT | IN OUT] tiopoDato [{:= | DEFAULT} expresión]
```

  - Al indicar los parámetros de debe indicar **solo el tipo** pero **no el tamaño**, y tampoco se permite la restricción **NOT NULL**.
  - Las opciones **IN, OUT, IN OUT** hacen referencia al tipo de parámetro (entrada, salida o entrada/salida). Los de tipo **IN** se utilizan para pasar valores al procedimiento en la invocación, los de tipo **OUT** para que el procedimiento devuelva valores a quien le invoca y los de tipo **IN OUT** realizan la doble función de pasar y devolver valores.
  - Los parámetros son variables accesibles desde el cuerpo del procedimiento y que admiten distintas formas de uso según el tipo.

- **Cuerpo:**

  El cuerpo comienza a continuación de **IS** y consta de las tres partes de cualquier bloque:
  - **Parte declarativa**, opcional, que se extiende hasta **BEGIN**, y es donde se declaran las variables locales al procedimiento.
  - **Parte ejecutable**, obligatoria, entre **BEGIN** y **EXCEPTION**.
  - **Parte de manejo de excepciones**, opcional, hasta **END**.

```
Los procedimientos NO pueden ser utilizados en expresiones

suma:=  valor1 + mi_proced (param1, param2);

dbms_output.put_line ('Nombre departamento es:'|| mi_proced(param1,param2));
```

2.1.1.- Creación y utilización de un Procedimiento desde SQL*PLUS.

- Creamos el procedimiento con la sentencia CREATE PROCEDURE… en un editor y lo salvamos como un script (Es decir un fichero con extensión SQL). Por ejemplo **miprocedimiento.sql**
- Desde SQL*PLUS, ejecutar el script para almacenar el código fuente y compilar el procedimiento. Ejecutamos con

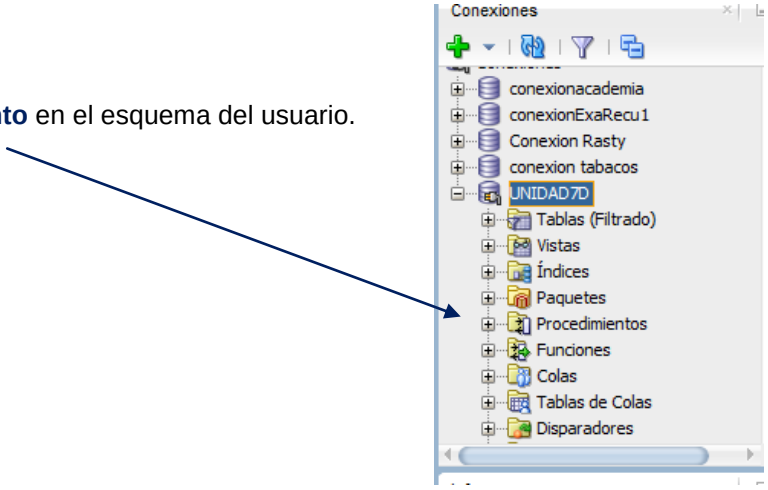
  start ruta: nombrefichero.sql

- Usar SHOW ERRORS para ver los errores de compilación
- Una vez compilado sin errores está preparado para ejecutarse y ser utilizado.
- Para ejecutarse podemos hacerlo con
  - ✓ La orden: **EXECUTE nombreProcedimiento(parámetros)**
  - ✓ Generando un bloque u otro procedimiento que lo invoque.

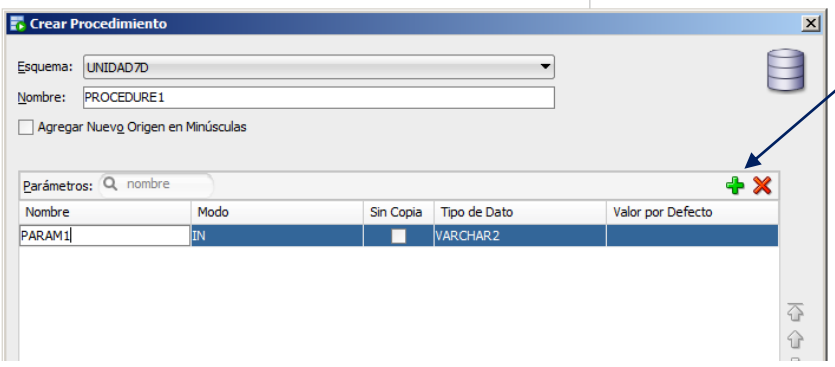
2.1.2.- Crear un procedimiento en el sqlDeveloper.

Para crear un procedimiento en el sqlDeveloper seguiremos los siguientes pasos:

- Seleccionar el objeto a **procedimiento** en el esquema del usuario.

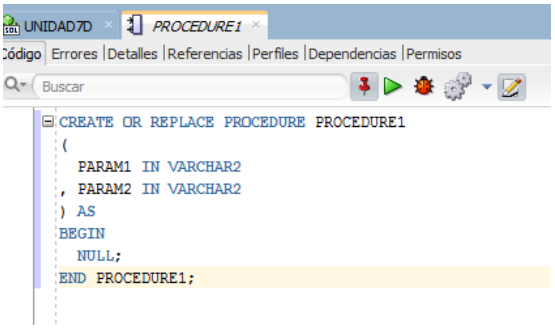


- Pulsando el botón derecho del ratón seleccionar nuevo procedimiento, dándole nombre y agregando los parámetros si los conocemos ya.

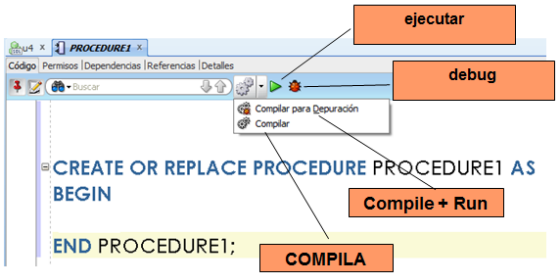


Cuando acabemos damos en aceptar

- En la ventana de edición nos aparece el procedimiento creado para que continuemos con la definición completa de nuestro procedimiento.



- Compilar para comprobar errores de escritura y en su caso corregir



- Si pinchamos 

  Se abre una ventana con un bloque que contiene un programa con el procedimiento para que copiemos y lo probemos en una ventana nueva de edición

- Otra posibilidad es directamente escribiendo el comando de creación con la sintaxis de procedimiento en la ventana de edición de nuestra conexión de usuario.
- Una vez completado pulsamos ejecutar  y nos compilara y guardara.

- Después podemos ejecutarlo con la sentencia EXECUTE.

Ejemplo-6

Crear un procedimiento que modifique el oficio de un empleado, teniendo como parámetros el numero de empleado y el oficio.

Solución

```
CREATE OR REPLACE PROCEDURE cambiarOficio (numeEmpleado NUMBER, nuevoOficio VARCHAR2)
AS
    viejoOficio    empleado.oficio%TYPE;
BEGIN
    SELECT oficio INTO viejoOficio FROM empleado WHERE numemp = numeEmpleado;
    UPDATE empleado SET oficio = nuevoOficio WHERE numemp = numeEmpleado;
    DBMS_OUTPUT.PUT_LINE (numeEmpleado||'.....Anterior oficio:'||viejoOficio || '.....Nuevo
                          oficio:' || nuevoOficio);
END cambiarOficio;
```

Ejemplo-7

Realizar un bloque que llame al procedimiento anterior para modificar el oficio del empleado 7902 que será DIRECTOR.

Solución

```
BEGIN
    cambiarOficio(7934,'ANALISTA');
END;
```

También podíamos hacerlos simplemente con la orden EXECUTE.

```
EXECUTE cambiarOficio(7902,'DIRECTOR');
```

2.1.3.- Privilegios a usuarios sobre Procedimientos.

- El procedimiento se ejecutará con los privilegios del propietario por defecto.
- Solo podrán utilizar un procedimiento el propietario y los usuarios con los privilegios

```
EXECUTE ANY PROCEDURE
```

- Para asignar un privilegio de ejecución de un procedimiento a un usuario:

```
GRANT EXECUTE ON nombre_procedure TO usuario;
```

```
GRANT EXECUTE ANY PROCEDURE TO usuario;
```

- Para ejecutar un procedimiento propiedad de otro usuario:

```
EXECUTE usuario.procedimiento;
```

2.2.- Funciones.

Las funciones PL/SQL tienen una estructura y funcionalidad similar a los procedimientos, pero **devuelven siempre un valor**, el del **RETURN** que debe de coincidir con el tipo de dato de la función.

También permiten utilizar parámetros IN, OUT, IN OUT

Se diferencian de los procedimientos en que **SI** se pueden utilizar como parte de una expresión.
Por ejemplo:

```
suma := salario + nvl(comision,0) + mi_funcion(param1,param2);
dbms_output.put_line('El valor es: '|| mi_funcion(param1,param2);
```

Las funciones tiene la siguiente sintaxis:

```
CREATE [OR REPLACE]
FUNCTION nombreFuncion [(lista de parámetros)] RETURN tipo_valor_devuelto
 IS/AS
      declaraciones locales
BEGIN
      instrucciones;
      RETURN expresión ;
EXCEPTION
      excepciones;
END [ nombreFuncion];
```

Tienen que coincidir el tipo de datos, sino da error

Para ejecutar la función:

```
EXECUTE variable:= nombreFuncion [(valores de parámetros)]
```

O bien crear un bloque para mostrar los resultados donde se utilice la función.

A su vez los parámetros tienen la siguiente sintaxis:

```
Parámetro [IN | OUT | IN OUT] tipo_dato [{:= | DEFAULT } expresión]
```

Como los procedimientos, las funciones tienen dos partes: Cabecera o especificación y cuerpo. La especificación se define entre **FUNCTION** e **IS** y el cuerpo comprende el resto.
Una función debe tener en su parte ejecutable al menos una sentencia **RETURN expresión.**

Al ejecutarse la sentencia **RETURN** termina la ejecución de la función y el valor de la expresión es asignado al identificador de la función en la expresión del bloque desde el que se invocó.

Puede haber varias sentencias **RETURN** con lo que la función dispondrá de varios puntos de finalización.
La cláusula **RETURN** de la especificación de la función indica el tipo de dato devuelto por la función. El valor resultante de la evaluación de la expresión de la sentencia RETURN debe ser compatible con ese tipo.

Ejemplo-8

```
CREATE OR REPLACE
   FUNCTION encontrarEempleado(vApellido VARCHAR2) RETURN INTEGER
   AS
       numeEmpleado        empleado.numemp%TYPE;
BEGIN
       SELECT numemp INTO numeEmpleado FROM empleado WHERE apell= vApellido;
       RETURN      numeEmpleado;
END encontrarEempleado;

Para ejecutarlo:
BEGIN
       DBMS_OUTPUT.PUT_LINE (encontrarEempleado('MUÑOZ'));
END;
```

2.3.- Uso de parámetros en los subprogramas.

Los parámetros se utilizan para pasar o recibir información en Procedimientos y Funciones.
Pueden se:

- **Parámetros actuales o reales**: las variables o expresiones indicadas en la llamada al subprograma. También se suelen llamar **argumentos.**
- **Parámetros formales**: variables declaradas en la especificación del subprograma

El compilador asocia los parámetros actuales a los formales basándose en su posición. Cuando llamamos a un subprograma, los parámetros actuales podemos escribirlos utilizando notación posicional o notación nombrada, es decir, la asociación entre parámetros actuales y formales podemos hacerla por posición o por nombre.

- ✓ En la **notación posicional**, el primer parámetro actual se asocia con el primer parámetro formal, el segundo con el segundo, y así para el resto.
- ✓ En la **notación nombrada** usamos el operador **=>** para asociar el parámetro actual al parámetro formal. También podemos usar notación mixta.
- ✓ Los parámetros que se corresponden deben ser del mismo tipo.

2.3.1.- Características de los parámetros.

Un parámetro tiene distintas características según se declare de tipo IN, OUT o IN OUT, sabemos que el tipo por defecto es IN.

TIPO	CARACTERÍSTICAS Y UTILIZACIÓN
IN	– Permite pasar valor a un subprograma – Dentro del subprograma no se puede modificar. Se comporta como constante – El parámetro actual puede ser una variable, constante, literal o expresión – Es la opción por defecto en el paso de parámetro
OUT	– Permite devolver valores al bloque que llamó al subprograma. Debe de especificarse – Dentro del subprograma, el parámetro actúa como una variable no inicializada – No puede intervenir en ninguna expresión, salvo para tomar un valor – El parámetro actual debe ser una variable
IN OUT	– Permite pasar valor inicial y devolver un valor actualizado. Debe de especificarse – Dentro del subprograma, el parámetro actúa como una variable inicializada – Puede intervenir en otras expresiones y puede tomar nuevos valores – El parámetro actual debe ser una variable

Ejemplo-9

Incrementar el salario de un empleado en un 10% mediante un procedimiento. El número de empleado será el parámetro.

```
CREATE OR REPLACE
PROCEDURE incrementaSalario (vNume IN empleado.numemp%TYPE)
IS
BEGIN
      UPDATE empleado SET salario= salario*1.10 WHERE numemp= vNume;
END incrementaSalario;

Comprobamos el funcionamiento
EXECUTE incrementaSalario (7900);
```

2.4.- Sobrecarga de subprogramas y recursividad.

PL/SQL también nos ofrece la posibilidad de sobrecargar funciones o procedimientos, es decir, llamar con el mismo nombre subprogramas que realizan el mismo cometido y que aceptan distinto número y/o tipo de parámetros.

PL/SQL también nos ofrece la posibilidad de utilizar la recursión o recursividad en nuestros subprogramas. Un subprograma es recursivo si éste se invoca a sI mismo hasta que se cumpla una condición de parada, momento a partir del cual se puede empezar a realizar los cálculos pendientes en memoria de las sucesivas llamadas.

Ejemplo-10

Desarrollar una función que calcule el factorial de un número.

```
CREATE OR REPLACE
    FUNCTION factorial (nume NATURAL) RETURN INTEGER
AS
BEGIN
    IF nume= 0  THEN
        RETURN 1;
    ELSE
        RETURN nume* factorial (nume-1);
    END IF;
END factorial;
```

Para comprobar el funcionamiento definimos el bloque siguiente:

```
BEGIN
    DBMS_OUTPUT.PUT_LINE(factorial(8));
END;
```

2.5.- Procedimientos frente a funciones.

PROCEDIMIENTO	FUNCIÓN
Se ejecuta como una sentencia PL/SQL	Son llamadas como parte de una expresión
No devuelve un tipo de dato asociado al nombre del procedimiento	Deben contener RETURN <tipo_dato>
Puede devolver 0, uno o más valores según la declaración de los parámetros realizada	Devuelve siempre un valor asociado al nombre de la función, aunque podemos usar también parámetros de forma IN, OUT, IN OUT

3.- Paquetes.

Un paquete es un objeto de la base de datos que agrupa lógicamente definiciones de tipos de datos, variables, constantes, excepciones, cursores y subprogramas, que tienen cierta relación entre ellos.

Un paquete tiene dos partes que se definen y almacenan por separado:

- **La especificación**: La especificación contiene las **declaraciones públicas** de subprogramas, tipos, constantes, variables, cursores, excepciones,...
  En el caso de los subprogramas solo contiene una declaración previa.
- **El cuerpo**: El cuerpo contienen la especificación completa de cursores y subprogramas. Estas declaraciones **son privadas** accesibles solo desde los objetos del paquete.
  Los paquetes que **no incluyen** cursores ni subprogramas carecen de cuerpo.

Los paquetes permiten leer múltiples objetos en memoria de una sola vez y **No** pueden ser llamados, parametrizados o anidados

3.1.- Creación de paquetes.

Para crear y almacenar un paquete hay que hacer las dos partes por separado, la sintaxis es la siguiente:

```
CREATE OR REPLACE PACKAGE nombePaquete
AS
      declaraciones de tipos, constantes, variables, cursores, excepciones,...
      especificación de subprogramas
END nombrePaquete;

CREATE OR REPLACE PACKAGE BODY nombrePaquete
AS
      declaraciones de tipos, constantes, variables, cursores, excepciones,..
      especificación de subprogramas
BEGIN
      ....
      instrucciones;
      ....
END nombrePaquete;
```

Todos los objetos declarados en el paquete pueden ser utilizados por los demás objetos del mismo Para llamar a un subprograma u objeto de un paquete, se ejecutaria con la siguiente sintaxis:

```
EXECUTE nombrePaquete.nombreSubprograma(parametros);
```

4.- Disparadores o Triggers de la base de datos.

Un disparador es un procedimiento PL/SQL almacenado asociado a una tabla que Oracle ejecuta o dispara automáticamente cuando se realiza una determinada operación sobre la tabla (inserción, borrado o modificación de filas) o sobre el sistema. Pudiendo elegir si se dispara antes o después de producirse uno o varios de los eventos.

Con los disparadores podemos:

- Prevenir transacciones erróneas
- Implementar reglas administrativas complejas
- Generar automáticamente valores derivados
- Auditar actualizaciones e, incluso, enviar alertas
- Gestionar réplicas remotas de la tabla

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

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

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

4.1.- Creación de un disparador.

Antes de codificar un disparador es necesario decidir sobre lo siguiente:

Parte	Descripción	Valores Posibles
Momento	Cuando se ejecuta el trigger con relación al evento	BEFORE AFTER INSTEAD OF (ejecuta el cuerpo del trigger en lugar de la sentencia. Para vistas que no pueden ser modificadas de otra forma)
Evento	Qué manipulación de datos sobre la tabla causa la ejecución del trigger	INSERT UPDATE DELETE
TABLA	Tabla sobre la que actúa el trigger	Nombre de la tabla
TIPO TRIGGER	Número de veces que se ejecuta el cuerpo del trigger	Sobre SENTENCIA (solo actúa una vez, por defecto) Sobre FILA (una vez para cada registro afectado por el evento
WHEN condición	Cuando se cumpla la condición indicada	
CUERPO DEL TRIGGER	Bloque PL/SQL	

La sintaxis para la creación de un disparador es:

```
CREATE [OR REPLACE] TRIGGER nombreTriggers
     {BEFORE | AFTER}
     {[DELETE] [[OR] INSERT]  [[OR] UPDATE [ OF columnas…]]}
ON  nombreTabla
     [REFERENCING {OLD AS  viejoNombre | NEW AS nuevoNombre … ]
     [FOR  EACH  ROW [ WHEN condición]]
/*Inicio del bloque PL/SQL */
DECLARE
     Declaraciones;
BEGIN
     Sentencias;
     [ EXCEPTION
         Gestión de excepciones]
END;
```

Donde:

- **Nombre:** nos indica el nombre que le damos al disparador.
- **BEFORE o AFTER**: nos dice cuándo será lanzado el disparador.
- **INSERT, DELETE, UPDATE**: será la acción que provoca el lanzamiento del disparador. Se puede especificar más de un evento separándolos por OR.
- **ON nombreTabla**: tabla sobre la que actúa el disparador.
- **REFERENCING** y **WHEN**: sólo podrán ser utilizados con disparadores para filas. **REFERENCING** nos permite asignar un alias a los valores **NEW** o/y **OLD** de las filas afectadas por la operación, y **WHEN** nos permite indicar al disparador que sólo sea lanzado cuando sea TRUE una cierta condición evaluada para cada fila afectada.
- **FOR EACH ROW:** Cuando el trigger solo se dispara para determinadas filas se empleará la opción FOR EACH ROW WHEN condición.

Ejemplo-12

Si intento hacer un INSERT INTO EMP en horario no laboral se dispara el trigger

```
CREATE OR REPLACE TRIGGER seguridadEmpleado
BEFORE INSERT ON empleado
BEGIN
    IF ( (TO_CHAR(SYSDATE,'DY') in ('SAB','DOM'))
        OR (TO_CHAR(sysdate,'HH24') NOT BETWEEN '08' AND '18'))
     THEN RAISE_APPLICATION_ERROR (-20500,'Solo se puede insertar en la tabla EMP
        en horario laboral');
    END IF;
END;
```

En el cuerpo de un disparador también podemos acceder a unos predicados que nos dicen qué tipo de operación se está llevando a cabo, que son: **INSERTING, UPDATING** y **DELETING**. Permiten combinar varios eventos de trigger en uno solo.
La sintaxis sería:

```
CREATE OR REPLACE TRIGGER nombreTriggers
    BEFORE  INSERT OR  UPDATE OR DELETE ON  nombreTabla
BEGIN
   IF INSERTING THEN
        .....
   ELSIF DELETING THEN
            .....
        ELSIF UPDATING ('nombreColumna') || UPDATING  THEN
            .......
   END IF;
   .....
END;
```

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

Si tenemos varios tipos de disparadores sobre una misma tabla, el orden de ejecución será:

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

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

```
DESC USER_TRIGGERS;
```