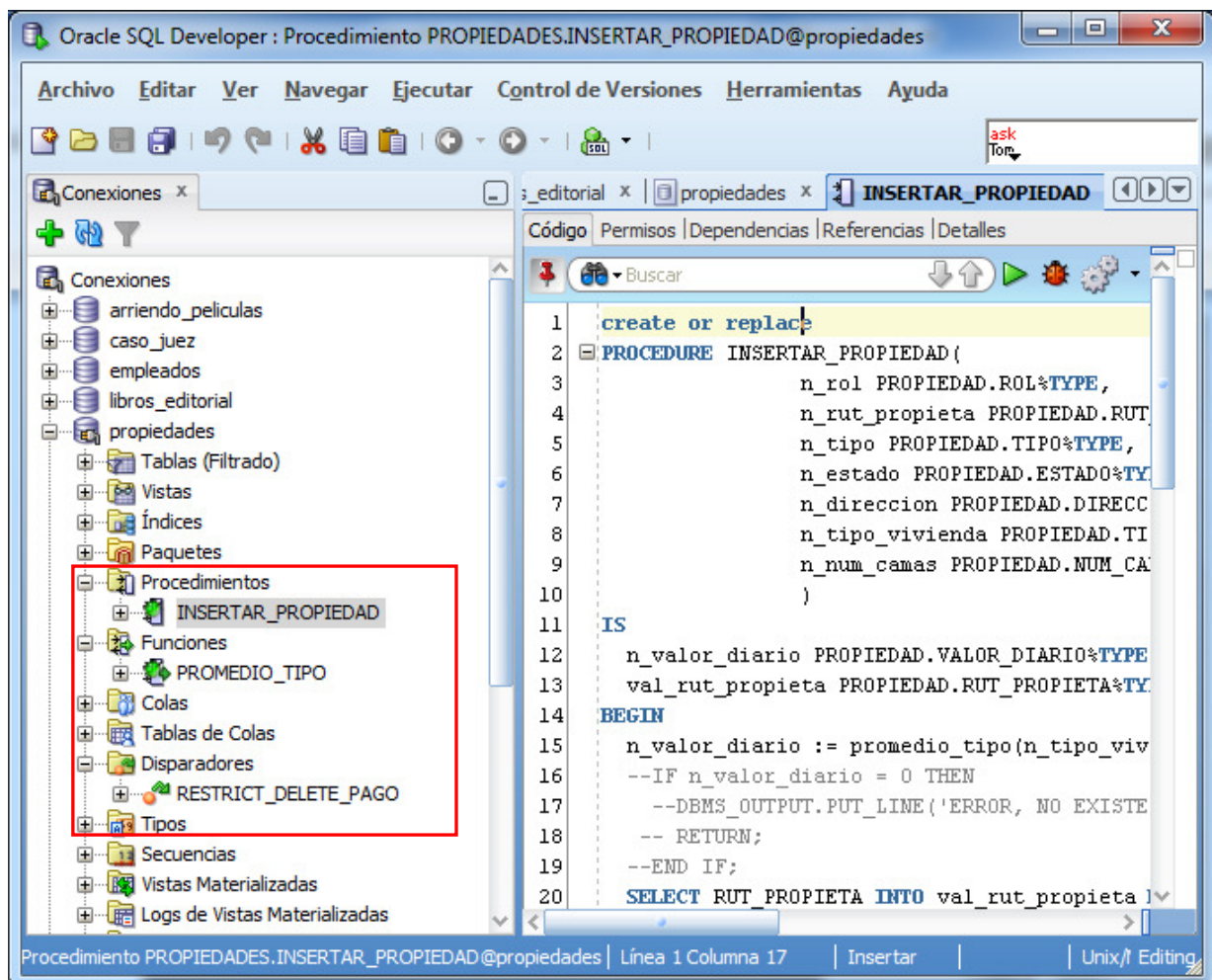


6. Subprogramas

Los subprogramas son bloques de instrucciones de PL/SQL que pueden ser invocados por otros y recibir parámetros.

- 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.
- Los subprogramas pueden recibir parámetros.
- Los subprogramas pueden ser de varios tipos:
 - Procedimientos almacenados.
 - Funciones.
 - Triggers o Disparadores.
 - Subprogramas en bloques anónimos.

Subprogramas – SqlDeveloper



Subprogramas

- En PL/SQL existen dos tipos de subprogramas PRINCIPALES:
 - Los *Procedimientos* y las *Funciones*.
 - Por regla general, se utiliza un procedimiento para ejecutar una acción específica y una función para calcular un valor.
- Los subprogramas también constan de una sección de declaraciones, un cuerpo que se ejecuta y una sección opcional de manejo de excepciones.

Ejemplo:

```
CREATE OR REPLACE PROCEDURE debit_account (acct_id INTEGER, amount
REAL)
IS
    old_balance REAL;
    new_balance REAL;
    overdrown EXCEPTION;
BEGIN
    SELECT bal INTO old_balance FROM accts WHERE acct_no = acct_id;
    new_balance := old_balance - amount;
    IF new_balance < 0 THEN
        RAISE overdrown;
    ELSE
        UPDATE accts SET bal = new_balance WHERE acct_no = acct_id;
    END IF;
EXCEPTION
    WHEN overdrown THEN
        ...
END debit_account;
```

- En el ejemplo, cuando el subprograma sea invocado, recibe los parámetros *acct_id* y *amount*.
- Con el primero de ellos selecciona el valor del campo “bal” y lo almacena en *old_balance*.
- Luego almacena una diferencia en otra variable, *new_balance*, la que de ser negativa gatillará una condición de excepción definida por el usuario (*overdrown*).

Ventajas de los subprogramas

- Los subprogramas proveen *extensibilidad*, es decir, permite crear nuevos programas cada vez que se necesiten, los cuales pueden ser invocados fácilmente y sus resultados utilizados de inmediato.
 - También aportan *modularidad*. Esto es, permite dividir un gran programa en módulos lógicos más pequeños y fáciles de manejar.
 - Esto apoya el diseño de programas utilizando la metodología top-down.
- Ventajas de los subprogramas
- Además, los subprogramas proveen las características de *reusabilidad* y *mantenibilidad*.

- Una vez construido, un subprograma puede ser utilizado en cualquier número de aplicaciones.
- Si la definición del tema que implementa es cambiada, entonces sólo se debe alterar el subprograma y no todos los lugares donde es referenciado.
- Finalmente, construir subprogramas agregan abstracción, lo que implica que es preciso conocer sólo qué es lo que hacen y no cómo están implementados necesariamente.

Procedimientos

- Un procedimiento es un subprograma que ejecuta una acción específica.
- La sintaxis para construirlos es la siguiente:

```
CREATE [OR REPLACE]
PROCEDURE <procedure_name> [(<param1> [IN|OUT|IN OUT] <type>,
<param2> [IN|OUT|IN OUT] <type>, ...)]
IS
-- Declaración 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.**

- Ejemplo:

```
CREATE OR REPLACE PROCEDURE xxx (param01 CHAR(5))
IS ...
```

–Esta sentencia es inválida. Debería decir sólo ...
param01 CHAR..., sin especificar el largo del carácter.

- Sin embargo, si es absolutamente necesario restringir el largo de una cadena, se puede corregir la situación codificando la llamada al procedimiento, de la siguiente manera:

```
DECLARE
temp CHAR(5);
SUBTYPE Char5 IS temp%TYPE;
CREATE OR REPLACE PROCEDURE xxx (param01 Char5) IS ...
```

- 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.

Uso de Parámetros

- Los subprogramas traspasan información utilizando parámetros.
- Las variables o expresiones referenciadas en la lista de parámetros de una llamada a un subprograma son llamados parámetros actuales.
- Las variables declaradas en la especificación de un subprograma y utilizadas en el cuerpo son llamados parámetros formales.
- En este último caso (para los parámetros formales) se pueden explicitar tres modos diferentes para definir la conducta de los parámetros: IN, OUT e IN OUT.
- Sin embargo, evite usar los tipos de parámetros OUT e IN OUT en las funciones, ya que por su naturaleza éstas devuelven un sólo dato y no a través de parámetros.
- **Es una mala práctica hacer que una función devuelva muchos valores; para eso utilice los procedimientos.**

Parámetros de modo IN

- Estos parámetros son la entrada a las funciones y procedimientos y actúan dentro de ellas como constantes, es decir, sus valores no pueden ser alterados dentro de un subprograma.
- Los parámetros actuales (que se convierten en formales dentro de un subprograma) pueden ser constantes, literales, variables inicializadas o expresiones.
- Dentro del procedimiento el parámetro formal se considera como de solo lectura. No puede ser cambiado.
- Cuando termina el procedimiento, y se devuelve el control al entorno que realizó la invocación, el parámetro no sufre cambios

Parámetros de modo OUT

- Un parámetro de salida (OUT) permite comunicar valores al bloque de programa que invocó al subprograma que utiliza este parámetro. Esto quiere decir además, que es posible utilizar un parámetro de este tipo como si fuera una variable local al subprograma. En el programa que invoca la función o procedimiento, el parámetro de modo OUT debe ser una variable, nunca una expresión o una constante.
- Se ignora cualquier valor que tenga el parámetro cuando se llama al procedimiento. Dentro del procedimiento, el parámetro formal se considera como de solo escritura, no puede ser leído, sino que tan solo pueden asignársele valores.
- Cuando termina el procedimiento y se devuelve el control al entorno que realizó la llamada, los contenidos del procedimiento formal se asignan al parámetro OUT.

Parámetros de modo IN OUT

- Esta modalidad permite proporcionar valores iniciales a la rutina del subprograma y luego devolverlos actualizados. Al igual que el tipo anterior, un parámetro de este tipo debe corresponder siempre a una variable.
- Si la salida de un subprograma es exitosa, entonces PL/SQL asignará los valores que corresponda a los parámetros actuales (los que reciben los valores de los parámetros formales en la rutina que llama al subprograma).
- Por el contrario, si la salida del subprograma se produce con un error no manejado en alguna excepción, PL/SQL no asignará ningún valor a los parámetros actuales.
- Este modo es una combinación de IN y OUT.
- El valor del parámetro se pasa al procedimiento cuando éste es invocado.
- Dentro del procedimiento, el parámetro puede ser tanto leído como escrito.
- Cuando termina el procedimiento y se devuelve el control al entorno que realizó la llamada, los contenidos del parámetro se asignan al parámetro IN OUT.

Resumen de las características de los parámetros:

IN	OUT	IN OUT
es el tipo por defecto	debe ser especificado	debe ser especificado
pasa valores a un subprograma	retorna valores a quien lo llamó	pasa valores iniciales al subprograma y retorna un valor actualizado a quien lo llamó
un parámetro formal actúa como una constante	parámetros formales actúan como variables	parámetros formales actúan como una variable inicializada
a un parámetro formal no se le puede asignar un valor	a un parámetro formal debe asignársele un valor	a un parámetro formal podría asignársele un valor
los parámetros actuales pueden ser constantes, variables inicializadas, literales o expresiones	los parámetros actuales deben ser variables	los parámetros actuales deben ser variables

- Un procedimiento posee dos partes: una especificación y un cuerpo.
- La especificación es simple, comienza con la palabra CREATE PROCEDURE y termina (en la misma línea) con el nombre del procedimiento o la lista de parámetros (que es opcional).
- El cuerpo del procedimiento comienza con la palabra reservada “IS” y termina con “END”, seguido opcionalmente por el nombre del procedimiento.

Ejemplo:

```
CREATE OR REPLACE
PROCEDURE Actualiza_Saldo (cuenta NUMBER,
new_saldo NUMBER)
IS
-- Declaración 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
-- Declaración 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**.
- 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
–Notación nominal

•**Notación posicional:** Se pasan los valores de los parámetros en el mismo orden en que se definieron en el procedure.

BEGIN

Actualiza_Saldo(200501,2500);
 COMMIT;

END;

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

=> : Es un operador de asociación (Cap. 2)

BEGIN

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

END;

•Ejecutar un procedimiento:

execute nombre_procedimiento(lista_parámetros);

O

exec nombre_procedimiento(lista_parámetros);

En SQL*PLUS:

SQL> execute tipo_salario(3000);

7902 Fabián

7788 Silvia

7901 Cecilia

Procedimiento PL/SQL terminado con éxito.

Llamar a un Procedimiento Almacenado desde JDBC:

•JDBC permite llamar a un procedimiento almacenado en la base de datos desde una aplicación escrita en Java.

•El primer paso es crear un objeto **CallableStatement**. Al igual que con los objetos **Statement** y **PreparedStatement**, esto se hace con una conexión abierta, **Connection**.

•Un objeto **CallableStatement** contiene una llamada a un procedimiento almacenado; no contiene el propio procedimiento.

•La primera línea del código siguiente crea una llamada al procedimiento almacenado **SHOW_SUPPLIERS** utilizando la conexión **con**.

•La parte que está encerrada entre corchetes es la sintaxis de escape para los procedimientos almacenados.

• Cuando un controlador encuentra "{call SHOW_SUPPLIERS}", traducirá esta sintaxis de escape al SQL nativo utilizado en la base de datos para llamar al procedimiento almacenado llamado **SHOW_SUPPLIERS**.

```
CallableStatement cs = con.prepareCall("{call SHOW_SUPPLIERS}");
ResultSet rs = cs.executeQuery();
```

```
import oracle.jdbc.OracleTypes;

public class EjemploDao {

    public int ejemplo(int datoABuscar, int otroDato) {
        Connection con = null;
        ResultSet rs = null;
        CallableStatement cs = null;
        int resultado = -1;
        try {
            con = dataSource.getConnection(); // Cambiar por tu implementación favorita
            cs = con.prepareCall(Constants.PROCEDIMIENTO1);
            int pos = 0;

            // Cargamos los parametros de entrada IN
            cs.setInt(++pos, datoABuscar);
            cs.setInt(++pos, otroDato);

            // Registramos los parametro de salida OUT
            cs.registerOutParameter(++pos, java.sql.Types.INTEGER);
            cs.registerOutParameter(++pos, OracleTypes.CURSOR);

            // Ejecutamos
            cs.execute();

            // Cosechamos los parametros de salida OUT
            resultado = cs.getInt(3); // Nuestro number
            rs = (ResultSet) cs.getObject(4); // Nuestro cursor, convertido en ResultSet
            while (rs.next()) {
                // Aqui hacemos lo que queramos...
            }
        } catch (Exception e) {
            e.printStackTrace();
        } finally {
            SQLTools.close(rs, cs, con);
        }
        return resultado;
    }
}
```


Con PHP...

```
<?
$msuscriptor_numero='79714306';
$msuscriptor_nombre="";
$db_conexion= mssql_connect(host, user, pass) or die("No se pudo conectar a la Base de
datos");
mssql_select_db("base de datos") or die(mssql_error());

$stmt=mssql_init("buscarXnumero",$db_conexion);
//buscar por numero es el nombre del procedimiento almacenado
[b]//aqui agregamos los parámetros de entrada[/b]
mssql_bind($stmt,"@suscriptor_numero",$msuscriptor_numero,SQLINT4);

[b]//aqui agregamos los parámetros de salida[/b]
mssql_bind($stmt,"@suscriptor_nombre",&$msuscriptor_nombre,SQLVARCHAR,TRUE,False,50);

[b]//aqui agregamos el parámetro return del procedimiento almacenado[/b]

mssql_bind($stmt,"RETVAL",&$filas,SQLINT4);

[b]aqui ejecutamos el procedimiento almacenado [/b]
mssql_execute($stmt);
echo "<h2>numero ".$msuscriptor_numero." Nombre ".$msuscriptor_nombre." </h2>";
mssql_close($db_conexion);
?>
```

Con ODBC .NET y Visual Basic .NET

Ejemplos de sintaxis de llamada:

- A continuación se incluye un ejemplo de la sintaxis de llamada de un procedimiento almacenado real de la base de datos de ejemplo xxxx que espera un único parámetro de entrada:

{CALL MUESTRA_EMPLEADO (?)}

- A continuación se incluye un ejemplo de una sintaxis de llamada de un procedimiento almacenado que espera un parámetro de entrada y devuelve un parámetro de salida y un valor. El primer marcador de posición representa el valor devuelto:

{? = CALL Procedure1 (?, ?) }

```
Dim cn As OdbcConnection

Try
    cn = New OdbcConnection("Driver={SQL Server};Server={local};Database=Northwind;Trusted_Connection=Yes")

    Dim cmd As OdbcCommand = New OdbcCommand("{? = call usp_TestParameters (?, ?)}", cn)
```

- Para eliminar un procedimiento:

DROP PROCEDURE nombre_procedimiento;

Funciones

- Una función es un subprograma que calcula un valor.
- 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 **sólo pueden ser de entrada**.
- La función también posee una especificación y un cuerpo.
- El segmento de especificación comienza con la palabra CREATE FUNCTION y termina con la cláusula RETURN, la cual especifica el tipo de dato retornado por la función.
- El cuerpo comienza con la palabra “IS” y termina con la palabra “END”, es decir, incluye las secciones de declaraciones, sentencias ejecutables y una parte opcional de manejo de excepciones.

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

DECLARE

Valor **NUMBER**;

BEGIN

Valor := **fn_Obtener_Precio**('000100');

DBMS_OUTPUT.PUT_LINE(Valor);

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;
```

Ejemplo:

Create or Replace FUNCTION revisa_salario (salario REAL, cargo CHAR) RETURN
BOOLEAN IS

salario_minimo REAL;

salario_maximo REAL;

BEGIN

SELECT lowsal, highsals INTO salario_minimo, salario_maximo

FROM salarios WHERE job = cargo ;

RETURN (salario >= salario_minimo) AND (salario <= salario_maximo);

END revisa_salario ;

- Esta misma función de ejemplo puede ser llamada desde una sentencia PL/SQL que reciba un valor booleano, como por ejemplo, en:

DECLARE

renta_actual REAL;

codcargo CHAR(10);

BEGIN

...

IF revisa_salario (renta_actual, codcargo) THEN ...

- La función *revisa_salario* actúa como una variable de tipo booleano, cuyo valor depende de los parámetros recibidos.

La sentencia RETURN

- Esta sentencia termina inmediatamente la ejecución de un programa, retornando el control al bloque de programa que lo llamó.

- No se debe confundir con la cláusula *return* de las funciones, que especifica el tipo de dato devuelto por ella.
- Un subprograma puede contener varias sentencias *Return*. Si se ejecuta cualquiera de ellas, el subprograma completo se termina.
- La sintaxis para los procedimientos es simple, sólo se necesita la palabra RETURN.
- Sin embargo, en el caso de las funciones, esta sentencia debe contener un valor, que es aquel que se va a devolver al programa que la llamó.
- La expresión que sigue a la sentencia puede ser tan compleja como se desee pero siempre debe respetar el tipo de datos que está definido en la cabecera (especificación) de la función.
- Una función debe contener como mínimo una sentencia RETURN, de otra manera, al no encontrarla, PL/SQL generará una excepción.

- Para borrar una función:

DROP FUNCTION nombre_función;

Recursividad

- La recursividad es una poderosa técnica que simplifica el diseño de algoritmos.
- Básicamente, la recursividad significa autoreferencia.
- Se aplica cuando un mismo algoritmo debe ser utilizado varias veces dentro de la solución a un problema determinado, cambiando cada vez las condiciones iniciales de cada ejecución (del algoritmo).
- Un programa recursivo es aquel que se llama a si mismo.
- Piense en una llamada recursiva como una llamada a otro subprograma que hace lo mismo que el inicial.
- Cada llamada crea una nueva instancia de todos los ítems declarados en el subprograma, incluyendo parámetros, variables, cursores y excepciones.
- Se recomienda ser muy cuidadoso con las llamadas recursivas.
 - Entre otras cosas, existe un máximo de veces que un mismo cursor puede ser abierto y eso se define en una variable de Oracle llamada OPEN_CURSORS. Al menos alguna vez la recursividad se debe revertir, es decir, las autoreferencias deben darse un número finito de veces

Recursividad versus Iteración

- La recursividad no es una herramienta considerada fundamental en la programación PL/SQL.
- Cualquier problema que requiera su utilización también puede ser resuelto utilizando iteración.
- Una versión iterativa de un programa es usualmente más fácil de diseñar y de entender. Sin embargo, la versión recursiva es más simple, pequeña y más fácil de depurar
- Ejemplos Clásicos:
 - Factorial
 - Serie de Fibonacci

- La versión recursiva de una función, en general, es mucho más elegante que la iterativa.
- Sin embargo, la iterativa, suele ser más eficiente; corre más rápido y utiliza menos memoria del computador.
- Si las llamadas son demasiadas se podrá advertir la diferencia en eficiencia. Considere esto para futuras implementaciones de una u otra alternativa.

Polimorfismo

- El polimorfismo es una característica del manejo de objetos.
- Significa que es posible definir más de un objeto con los mismos nombres, pero diferenciados únicamente por la cantidad o tipo de los parámetros que reciben o devuelven.
- En el caso de los subprogramas, es posible declarar más de uno con el mismo nombre, pero se deberá tener la precaución de diferenciarlos en cuanto al tipo de parámetros que utilizan.

Ejemplo

```
PROCEDURE initialize
(tab OUT DateTabTyp,
n INTEGER)
IS
BEGIN
FOR i IN 1..n LOOP
tab(i) := SYSDATE;
END LOOP;
END initialize;
```

```
PROCEDURE initialize
(tab OUT RealTabTyp,
n INTEGER)
IS
BEGIN
FOR i IN 1..n LOOP
tab(i) := 0.0;
END LOOP;
END initialize;
```

- Estos procedimientos sólo difieren en el tipo de dato del primer parámetro.
- Para efectuar una llamada a cualquiera de ellos, se puede implementar lo siguiente:

DECLARE

```
TYPE DateTabTyp IS TABLE OF DATE INDEX BY BINARY_INTEGER;
TYPE RealTabTyp IS TABLE OF REAL INDEX BY BINARY_INTEGER;
hiredate_tab    DateTabTyp;
comm_tab        RealTabTyp;
indx            BINARY_INTEGER;
```

...

BEGIN

```
indx := 50;
initialize(hiredate_tab, indx);    -- llama a la primera versión
initialize(comm_tab, indx); -- llama a la segunda versión
```

...
END;

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>;
```

- Ej:

```
AFTER DELETE ON nombre_tabla
AFTER DELETE OF nombre_columna ON nombre_tabla
```

Activar/desactivar de disparadores:

- Todos los disparadores asociados a una tabla:
ALTER TABLE nombre_tabla ENABLE ALL TRIGGERS
ALTER TABLE nombre_tabla DISABLE ALL TRIGGERS
- Un disparador específico:
ALTER TRIGGER nombre_disparador ENABLE
ALTER TRIGGER nombre_disparador DISABLE

OTRAS FUNCIONES

- Eliminar un disparador
DROP TRIGGER nombre_disparador
- Ver todos los disparadores y su estado
SELECT TRIGGER_NAME , STATUS FROM USER_TRIGGERS;
- Ver el cuerpo de un disparador
SELECT TRIGGER_BODY FROM USER_TRIGGERS WHERE
TRIGGER_NAME='nombre_disparador';
- Ver la descripción de un disparador
SELECT DESCRIPTION FROM USER_TRIGGERS WHERE
TRIGGER_NAME= 'nombre_disparador';

- 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 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 sólo 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 <u>qué tipo de orden DML</u> provoca la activación del disparador.
BEFORE , AFTER	Define si el disparador <u>se activa antes o después de que se ejecute la orden</u> .
FOR EACH ROW	<p>Los disparadores con <u>nivel de fila se activan una vez por cada fila afectada</u> por la orden que provocó el disparo.</p> <p>Los disparadores con <u>nivel de orden se activan sólo una vez, antes o después de la orden</u>. Los disparadores con nivel de fila se identifican por la cláusula FOR EACH ROW en la definición del disparador.</p> <p>La cláusula WHEN sólo es válida para los disparadores con nivel de fila</p>

•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 PRODUCTOS:

```
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 con respecto al Orden

•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.

Utilización de predicados

- Su comportamiento es el siguiente:

```
CREATE OR REPLACE TRIGGER ejemplo
BEFORE INSERT OR UPDATE OR DELETE ON tabla
BEGIN
IF DELETING THEN
Acciones asociadas al borrado;
ELSIF INSERTING THEN
Acciones asociadas a la inserción;
ELSE
Acciones asociadas a la modificación;
END IF;
END ejemplo;
/
```

RAISE_APPLICATION_ERROR

- RAISE_APPLICATION_ERROR (nro_error, mensaje); [-20000 y -20999]

•EJEMPLO:

- Escribir un trigger que impida insertar un empleado con una comisión superior al 10% del promedio de las comisiones de todos los empleados. Validar con el número de mensaje - 20001, cuando la comisión sea muy alta:

```
Create or replace trigger emp_comision
before insert on empleado
For each row
Declare
    V_comision empleado.comision%type;
Begin
    Select avg(comision)*1.1 into v_comision from empleado;

    If :new.comision > v_comision Then
RAISE_APPLICATION_ERROR (-20001, 'La comisión es muy alta') ;
    End If;
End;
```

- Al probar...

```
SQL> insert into empleado values (9000, 'Karlo', 'analista','7839','17/11/81',5000,600,40,1);
insert into empleado values (9000, 'Karlo', 'analista','7839','17/11/81',5000,600,40)
```

*

ERROR en línea 1:

ORA-20001: La comisión es muy alta

ORA-06512: en "CERTAMEN_PL.EMP_COMISION", línea 6

ORA-04088: error durante la ejecución del disparador

'CERTAMEN_PL.EMP_COMISION'

- Si tienen más de 60 líneas de código conviene hacer un procedimiento, pues se compila cada vez que se va a ejecutar
- No se deben utilizar para hacer cumplir las restricciones de integridad del schema (de la BD por las claves primarias y foráneas)

Restricciones de los disparadores

- Tabla mutante. La que está siendo modificada por una operación DML o una tabla que se verá afectada por los efectos de un DELETE CASCADE debido a la integridad referencial
- Tabla de restricción. Tabla de la que un disparador puede necesitar leer debido a una restricción de integridad referencial

Restricciones de los disparadores

- Las órdenes del cuerpo de un disparador no pueden:
 - Leer o modificar una tabla mutante
 - Leer o modificar claves primarias o ajenas de una tabla de restricción
- Para evitar estos problemas es necesaria la utilización de paquetes o package

Tabla mutante

- Dos tablas: empleado y departamento. En la tabla empleado tenemos el número de departamento como clave ajena.

Tabla Mutante en SQL*PLUS:

```
SQL> CREATE or replace TRIGGER ejemplo
2 AFTER UPDATE OF cod_Depto On depto
3 FOR EACH ROW
4 BEGIN
5 UPDATE empleado
6 SET empleado.cod_depto = :NEW.cod_depto
7 WHERE empleado.cod_depto= :OLD.cod_depto;
8 END ejemplo;
9 /

Disparador creado.

SQL> UPDATE depto
2 SET cod_depto= 50
3 WHERE cod_depto=10;
UPDATE depto
*
ERROR en línea 1:
ORA-04091: la tabla A.DEPTO está mutando, puede que el disparador/la función no puedan verla
ORA-06512: en " A.EJEMPLO", línea 2
ORA-04088: error durante la ejecución del disparador ' A.EJEMPLO'
```

Subprogramas 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 pueden ser enormemente útiles.

El siguiente ejemplo declara, ejecuta y 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

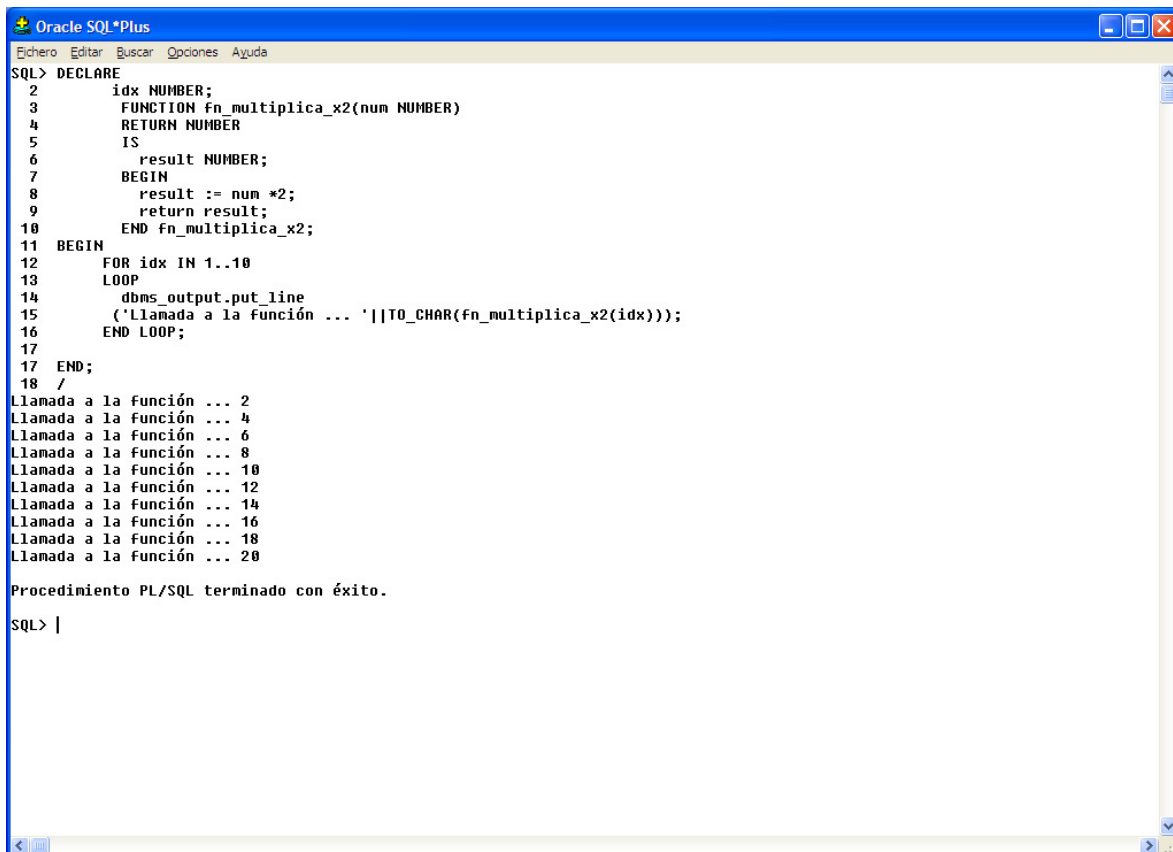
```
FOR idx IN 1..10
LOOP
dbms_output.put_line
('Llamada a la función ...'||TO_CHAR(fn_multiplica_x2(idx)));
```

END LOOP;
END;

•El resultado sería:

Llamada a la función ... 2
Llamada a la función ... 4
Llamada a la función ... 6
Llamada a la función ... 8
Llamada a la función ... 10
Llamada a la función ... 12
Llamada a la función ... 14
Llamada a la función ... 16
Llamada a la función ... 18
Llamada a la función ... 20

.
.
.



```
SQL> DECLARE
  2     idx NUMBER;
  3     FUNCTION fn_multiplica_x2(num NUMBER)
  4     RETURN NUMBER
  5     IS
  6         result NUMBER;
  7     BEGIN
  8         result := num *2;
  9         return result;
 10     END fn_multiplica_x2;
 11 BEGIN
 12     FOR idx IN 1..10
 13     LOOP
 14         dbms_output.put_line
 15         ('Llamada a la función ... '||TO_CHAR(fn_multiplica_x2(idx)));
 16     END LOOP;
 17 END;
 18 /
Llamada a la función ... 2
Llamada a la función ... 4
Llamada a la función ... 6
Llamada a la función ... 8
Llamada a la función ... 10
Llamada a la función ... 12
Llamada a la función ... 14
Llamada a la función ... 16
Llamada a la función ... 18
Llamada a la función ... 20

Procedimiento PL/SQL terminado con éxito.
SQL> |
```

Nótese que se utiliza la función TO_CHAR para convertir el resultado de la función fn_multiplica_x2 (numérico) en alfanumérico (es opcional).