

10.1 DISPARADORES (TRIGGERS)

El usuario podrá especificar una serie de acciones distintas ante una determinada condición. El usuario escribe el procedimiento a aplicar dependiendo del resultado de la condición.

Hay tres tipos de disparadores de base de datos:

- Disparadores de tablas: asociados a una tabla. Se disparan cuando se produce un determinado suceso o evento de manipulación que afecta a la tabla (inserción, borrado o modificación).
- Disparadores de sustitución: asociados a vistas. Se disparan cuando se intenta ejecutar un comando de manipulación que afecta a la vista (inserción, borrado o modificación de filas).
- Disparadores del sistema: se disparan cuando ocurre un evento del sistema (arranque o parada de la base de datos, entrada o salida de un usuario..) o una instrucción de definición de datos (creación, modificación o eliminación de una tabla u otro objeto).

Sintaxis

La sintaxis para la creación de disparadores en SQL 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 nombre del disparador se usará si se desea eliminar el disparador, concretamente, se usa como argumento del comando DROP TRIGGER. El resto de la sintaxis se describe a continuación:

* La palabra siguiente (momento) determina si la función debe ser llamada antes (BEFORE) o después (AFTER) del evento.

* El siguiente elemento del comando determina qué eventos dispararán la ejecución (INSERT, DELETE, UPDATE). Es posible especificar múltiples eventos utilizando el operador OR.

* El nombre de la relación (nombre_tabla) determinará la tabla afectada por el evento. La instrucción FOR EACH determina si el disparador se ejecutará para cada fila afectada o bien antes (o después) de que la secuencia se haya completado.

Statement: el trigger se activará una sola vez para cada orden, independientemente del número de filas afectadas por ella.

Row: el trigger se activará una vez por cada fila afectada por la orden.

* La cláusula when seguida de una condición restringe la ejecución del trigger al cumplimiento de la condición especificada.

* El cuerpo del trigger es la definición del código que se invocará. BEGIN END.

Una característica importante de los disparadores es que no se puede tener más de uno para una determinada combinación de tabla, evento y momento. Es decir, no se pueden definir dos disparadores sobre la tabla T siendo ambos AFTER INSERT. Esto hace que si hay diferentes acciones a realizar para esa combinación, habrá que programar el disparador para que ejecute condicionalmente unas acciones u otras.

Ejemplo:

El siguiente trigger se disparará cada vez que se actualice un empleado guardando su número de empleado, nombre y oficina en una fila de la tabla auditareemple:

```
CREATE or REPLACE TRIGGER dis_ejemplo1
BEFORE UPDATE
ON empleados
FOR EACH ROW
BEGIN
insert into auditareemple values (:old.apellido || ' ' ||:new.apellido);
END dis_ejemplo1;
```

Este disparador requiere la tabla auditareemple , que habra sido creada.

```
CREATE TABLE auditareemple(col1 varchar(200));
```

Para probar si funciona el disparador haremos una actualización en la tabla empleados:

```
UPDATE empleados SET apellido=' Ernest' where apellido=' Ernst;
```

Vemos que la tabla empleados se ha actualizado y en la tabla auditareemple aparecerá una fila con el apellido viejo y apellido nuevo.

Dentro BEGIN ... END se puede definir un disparador que ejecute distintas sentencias como condicionales y bucles.

- **Valores NEW y OLD**

Se puede hacer referencia a los valores anterior y posterior a una actualización a nivel de fila. Lo haremos como: **:old.nombrecolumna y :new.nombrecolumna**

Nota: en mysql se pone old.nombrecolumna y new.nombrecolumna

Al utilizar los valores old y new deberemos tener en cuenta el evento de disparador:

Cuando el evento que dispara el trigger es DELETE, deberemos hacer referencia a old.nombrecolumna, ya que el valor de new es NULL.

Cuando el evento de disparo es INSERT, deberemos referirnos siempre a new.nombrecolumna, puesto que el valor old no existe (es NULL).

Para los triggers cuyo evento de disparo es UPDATE se puede utilizar los dos.

- **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.

Ejemplo:

```
CREATE OR REPLACE TRIGGER dis_ejemplo2
BEFORE INSERT OR UPDATE OR DELETE ON empleados
FOR EACH ROW
BEGIN
  IF UPDATING THEN
    INSERT INTO auditaremples VALUES (:old.apellido || ' ' || :new.apellido);
  ELSIF DELETING THEN
    INSERT INTO auditaremples VALUES (:old.apellido || ' ' || current_date || ' BAJA ');
  ELSIF INSERTING THEN
    INSERT INTO auditaremples VALUES ('ALTA ' || current_date || ' ' || :new.apellido);
  END IF;
END dis_ejemplo2;
```

Si insertamos:

```
insert into empleados values (220,'Ines','Garcia', 9999, null, current_date, 'AD_VP', 23000,
null, 100, 90);
```

Vemos que habrá insertado en la table empleados y además en la table auditaremples aparecerá la palabra ALTA , fecha de alta y el apellido.

```
Delete from empleados where apellido='Garcia';
```

Borrara de la table empleados y en la table auditaremples aparecerá el apellido , fecha actual y la palabra BAJA.

DROP TRIGGER

DROP TRIGGER *nombre_disp.*

Ejemplo

```
DROP TRIGGER dis_ejemplo2;
```

10.2 PROGRAMAS.

Para crear nuestros programas podemos utilizar alguna de las herramientas específicas de desarrollo de Oracle, como **JDeveloper**, **Oracle SqlDeveloper**, que incluyen facilidades gráficas de edición, depuración y compilación.

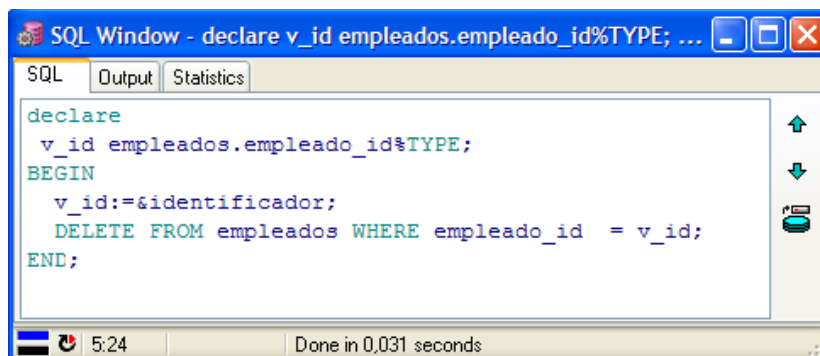
En oracle se puede distinguir dos tipos de bloques:

- **Bloques anónimos:** bloques PL/SQL que no tienen ningún nombre asociado. Este comenzará con un DECLARE o BEGIN.
- **Subprogramas:** bloques PL/SQL que tienen un nombre y pueden recibir y devolver valores. Normalmente se guardan en la base de datos y podemos ejecutarlos invocándolos desde otros subprogramas o herramientas.

Bloques anónimos

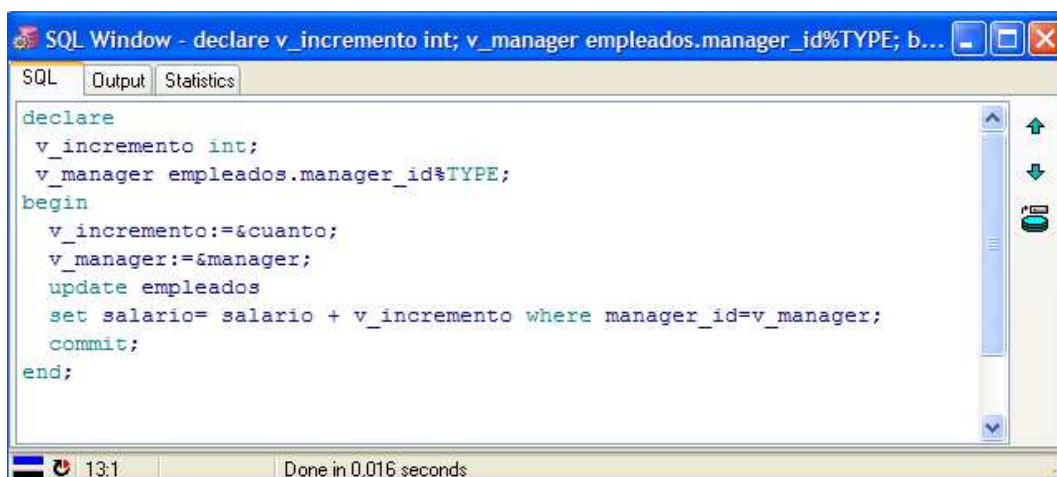
Dentro de un bloque anónimo podemos utilizar cualquier **orden de manipulación** de datos.

Ejemplos:



```
SQL Window - declare v_id empleados.empleado_id%TYPE; ...
SQL Output Statistics
declare
  v_id empleados.empleado_id%TYPE;
BEGIN
  v_id:=&identificador;
  DELETE FROM empleados WHERE empleado_id = v_id;
END;
5:24 Done in 0,031 seconds
```

Borra de la tabla *empleados* la fila correspondiente al cliente cuyo *empleado_id* se especifica en la variable *v_id*.



```
SQL Window - declare v_incremento int; v_manager empleados.manager_id%TYPE; b...
SQL Output Statistics
declare
  v_incremento int;
  v_manager empleados.manager_id%TYPE;
begin
  v_incremento:=&cuanto;
  v_manager:=&manager;
  update empleados
  set salario= salario + v_incremento where manager_id=v_manager;
  commit;
end;
13:1 Done in 0,016 seconds
```

Actualiza el salario de aquellos empleados cuyo manager y salario se introducen desde el teclado.

Sin embargo, cuando queremos realizar una consulta SELECT dentro de un bloque nos da error. Esto es debido, a que el resultado queda en un área de memoria denominada **cursor implícito** a la que accederemos utilizando variables.

Por ejemplo, para obtener en PL/SQL el número total de empleados de la tabla del mismo nombre no podemos utilizar directamente la instrucción SQL correspondiente pues dará error:

```
BEGIN
Select count(*) from emleados;  error!!!!
END;
```

Utilizaremos variables (declaradas previamente) junto a la cláusula INTO para acceder a los datos devueltos en la consulta.

Por ejemplo:

```
BEGIN

Select count(*) INTO total_emple from empleados.

END;
```

El formato básico es:

SELECT <columna/s> INTO <variable/s> FROM <tabla> WHERE...];

Debe de haber coincidencia en el tipo entre las variables con las columnas especificadas en la cláusula SELECT.

La consulta deberá devolver **una única fila**, pues en caso contrario se producirá un error como TOO_MANY_ROWS.

Por ejemplo:

```
DECLARE
  v_ape varchar2(10);
  v_salario number (8,2);
BEGIN
  SELECT apellido, salario INTO v_ape, v_salario
  FROM empleados WHERE empleado_id=106;
  DBMS_OUTPUT.PUT_LINE(v_ape || ' ' || v_salario);
  COMMIT;
END;
```

10.3 GESTION DE EXCEPCIONES

Las excepciones sirven para tratar errores y mensajes de aviso. En Oracle estan disponibles excepciones predefinidas correspondientes a errores mas comunes como NO_DATA_FOUND, TOO_MANY_ROWS.

NO_DATA_FOUND: no ha devuelto ningún valor.

TOO_MANY_ROWS: ha devuelto más de una fila.

Las excepciones se disparan automáticamente al producirse un error asociado. La sección EXCEPTION es la encargada de gestionar los errores que podrían darse durante la ejecución mediante la cláusula WHEN .

El ejemplo anterior con la clausula EXCEPTION

```
DECLARE
  v_ape varchar2(10);
  v_salario number (8,2);
BEGIN
  SELECT apellido, salario INTO v_ape, v_salario
  FROM empleados WHERE empleado_id=106;
  DBMS_OUTPUT.PUT_LINE(v_ape || ' ' || v_salario);
  COMMIT;

EXCEPTION

  WHEN NO_DATA_FOUND THEN
    RAISE_APPLICATION_ERROR(-20000,' ERROR no hay datos ');
  WHEN TOO_MANY_ROWS THEN
    RAISE_APPLICATION_ERROR(-20000,' ERROR demasiados datos');
  WHEN OTHERS THEN
    RAISE_APPLICATION_ERROR(-20000,'Error en la aplicación');

END;
```

PL/SQL cuando detecta un error automáticamente va a la zona de exception. Allí buscará el error o uno genérico y realizará el tratamiento correspondiente. Al finalizar el tratamiento, sale del bloque actual y devuelve el control al programa o herramienta que realizó la llamada.

Subprogramas: procedimientos y funciones

- Son bloques que tienen nombre por el que son invocados desde otros programas.
- Se compilan, almacenan y ejecutan en la base de datos oracle.
- Tienen una cabecera que incluye el nombre del subprograma (indicando si se trata de procedimiento o función), los parámetros y el tipo de valor de retorno.
- La zona de declaraciones y el bloque o cuerpo del programa comienza con la palabra IS o AS.
- Pueden ser de dos tipos: procedimientos o funciones.

Procedimiento	Función
PROCEDURE <nombre_procedimiento> [<lista de parámetros>] IS [<declar objetos locales>;] BEGIN <instrucciones>; [EXCEPTION <excepciones>;] END [<nombre_procedimiento>;]	FUNCTION <nombre_función> [<lista de parámetros>] IS [<declar objetos locales>;] BEGIN <instrucciones>; RETURN <expresión>; [EXCEPTION <excepciones>;] END [<nombre_función>;]

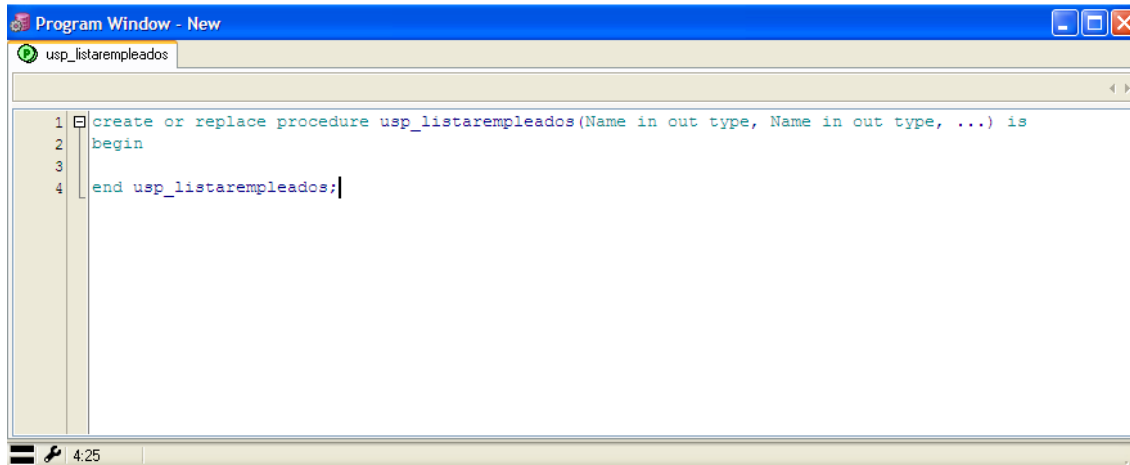
❖ Procedimientos

Para crear un procedimiento seguiremos los siguientes pasos:

FILE/NEW/PROGRAM WINDOW /PROCEDURE

Indicamos el nombre del procedimiento : **usp_listareempleados**

El prefijo usp (user stored procedure) se suele utilizar para indicar que es un procedimiento almacenado.



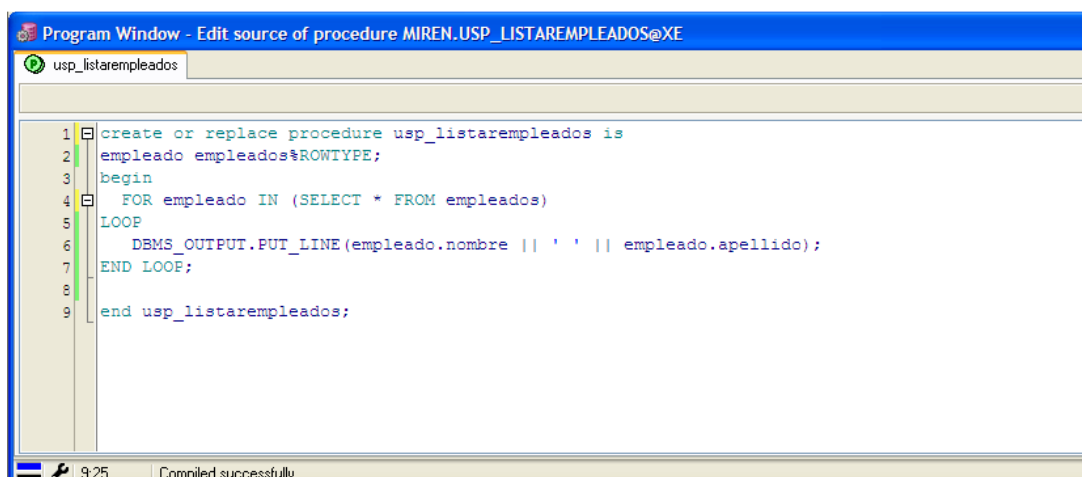
Tras el comando create procedure aparece el nombre del procedimiento, y opcionalmente los parámetros entre paréntesis. En nuestro ejemplo no lo necesitamos. Seguidamente aparecerá la palabra IS (o también AS) y el código del procedimiento que lo pondremos entre el BEGIN ... END. Nuestro procedimiento solamente listará los empleados de la base de datos, por lo que no necesita ningún tipo de parámetros.

Si necesitamos variables éstas deberán ser declaradas previamente, es decir, tras la palabra IS y antes de BEGIN.

En nuestro caso vamos a definir la variable empleado empleados%ROWTYPE;

A continuación el código sql entre BEGIN .. END.

```
FOR empleado IN (SELECT * FROM empleados)
LOOP
  DBMS_OUTPUT.PUT_LINE(empleado.nombre || ' ' || empleado.apellido);
END LOOP;
```



```


1 create or replace procedure usp_listareempleados is
2 empleado empleados%ROWTYPE;
3 begin
4   FOR empleado IN (SELECT * FROM empleados)
5   LOOP
6     DBMS_OUTPUT.PUT_LINE(empleado.nombre || ' ' || empleado.apellido);
7   END LOOP;
8
9 end usp_listareempleados;

```

El lenguaje PL/SQL no es sensible a la combinación de mayúsculas y minúsculas que utilicemos, excepto los literales que van entre comillas simples.

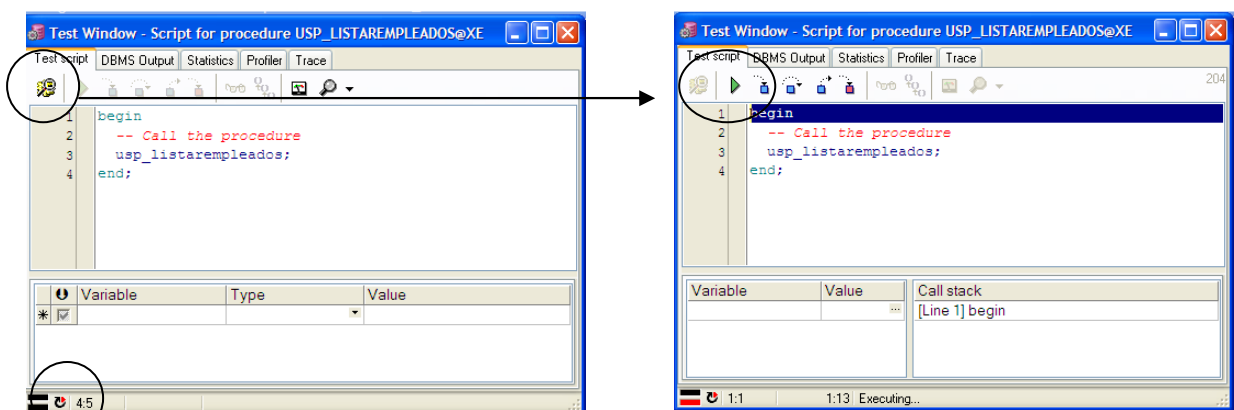
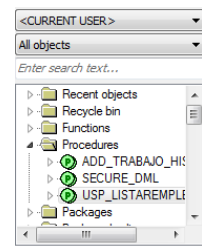
La parte de código que va desde CREATE PROCEDURE hasta IS se conoce como **especificación** del procedimiento, donde pueden aparecer parámetros.

El resto del código es el **cuerpo** del procedimiento; con una sección de declaración y después un bloque BEGIN..END.

Ejecutamos el procedimiento  y si da todo correcto aparecerá compiled successfully.

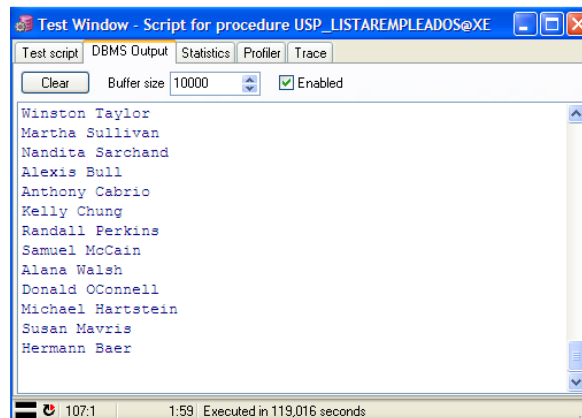
Para ver el resultado será:

Botón derecho sobre el procedimiento usp_listareempleados y elegimos test



Desde aquí también

Obtenemos el resultado



Las aplicaciones podrían utilizar el identificador `usp_listareempleados` en su programación para obtener este resultado.

Por defecto los procedimientos almacenados se ejecutan con la identidad y los privilegios del usuario que los ha creado.

Los procedimientos almacenados tienen ventajas respecto a las consultas directas contra las base de datos, pero no serían de mucha utilidad si no se utilizan parámetros.

Estos sirven para particularizar el comportamiento del procedimiento almacenado.

En el ejemplo anterior hemos obtenido todos los empleados pero que pasa si queremos solamente aquellos cuyo salario es superior a 4000.

Existen tres tipos de parámetros:

Parámetros de entrada (IN): que sirven para comunicar algún detalle al procedimiento almacenado.

Parámetros de salida (OUT): sirve para que el procedimiento devuelva información al código que lo ha ejecutado.

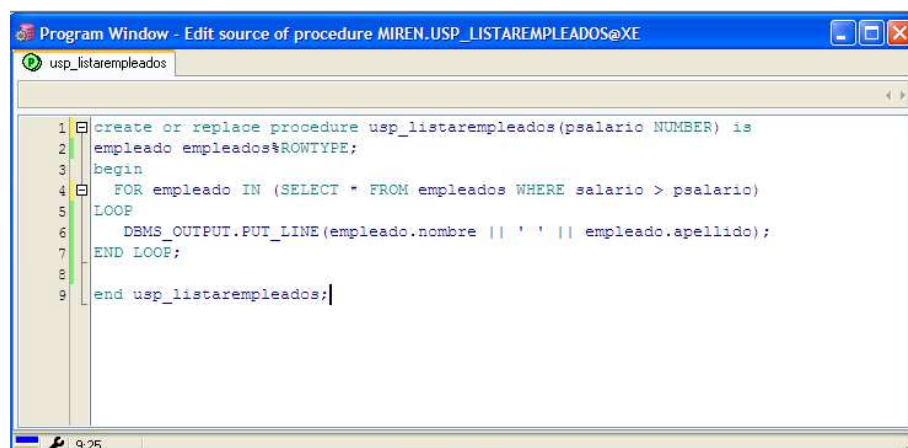
Parámetros de entrada/salida (IN/OUT): que se puede utilizar con ambos propósitos

Tal y como hemos dicho antes se declaran entre paréntesis detrás del nombre.

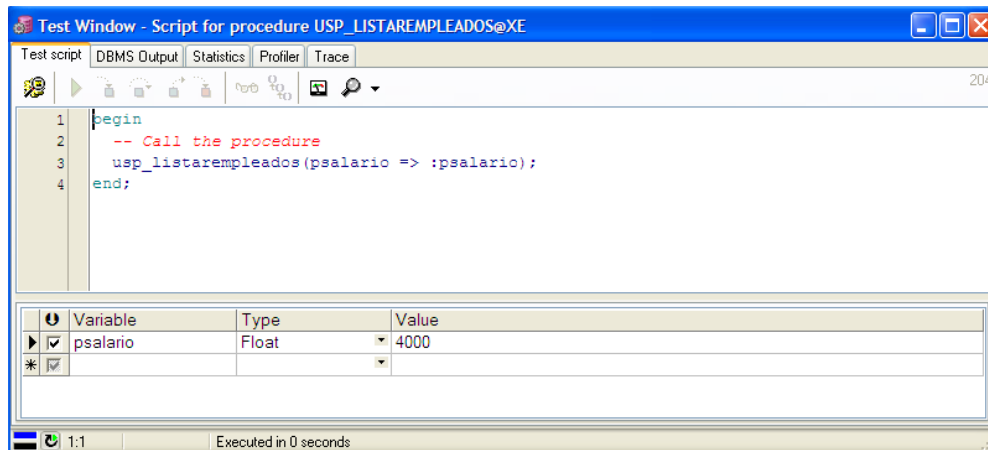
Los parámetros solamente hay que definir el nombre y el tipo, no el tamaño ni la precisión. Si no se pone nada toma por defecto que es de entrada (IN) en caso contrario se debería especificar el tipo de parámetro.

Ahora se puede utilizar ese parámetro para particularizar el SELECT

En nuestro ejemplo



Siguiendo los pasos anteriores llegaremos a esta ventana, pero en este caso nos pide el valor de parámetro.



También se puede realizar la llamada de la siguiente forma:

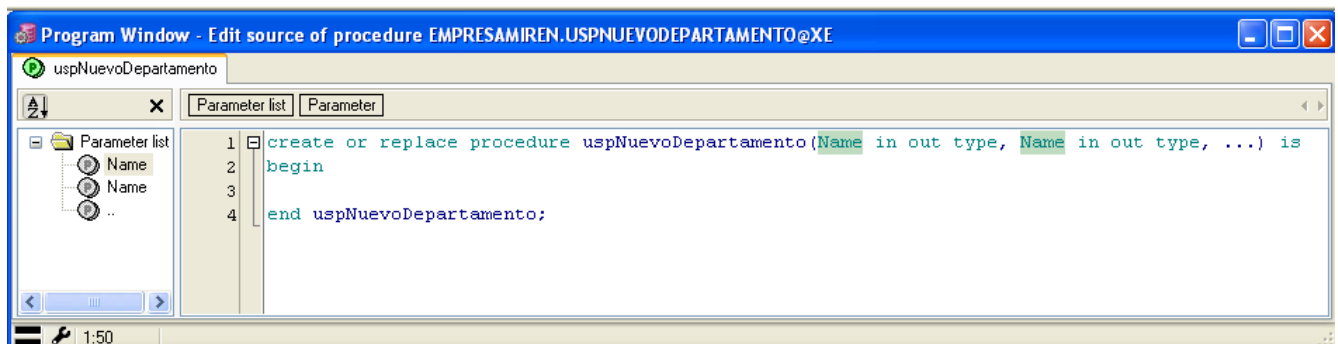
```

begin
    usp_listaremplados(15000);
end;

```

Vamos a crear un nuevo ejemplo y dar de alta a un nuevo departamento. Al nuevo procedimiento le llamaremos: **uspNuevoDepartamento**

La introducción de los parámetros se puede realizar directamente en código o a través de:

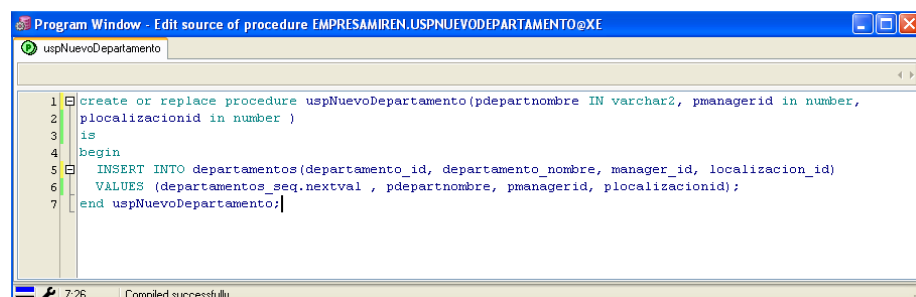


Creamos los siguientes parámetros:

pdepartnombre IN VARCHAR
pmanagerid IN NUMBER
plocalizacionid IN NUMBER

A continuación escribimos el cuerpo del procedimiento.

INSERT INTO departamentos(departamento_id, departamento_nombre, manager_id, localizacion_id) VALUES (departamentos_seq.nextval , pdepartnombre, pmanagerid, plocalizacionid);



Para obtener el siguiente identificador de departamentos utilizamos departamentos_seq. Esto es posible porque a la hora de creación la base de datos hemos realizado un create sequence departamentos_seq.

¿Qué pasa si el código que utiliza el procedimiento quiere saber el identificador del departamento?. Utilizaremos un parametro de salida. Añadimos al final de los parámetros

pdepartaid OUT number.

Después escribiremos la siguiente orden:

SELECT departamentos_seq.currval INTO pdepartaid FROM dual;

Si con la propiedad NEXTVAL obtenemos el siguiente valor libre, con CURRVAL obtenemos el último que se ha utilizado.

Se ha utilizado la tabla DUAL como comodín para realizar la consulta.

Ahora vamos a realizar una llamada a este procedimiento, escribiendo su código sql. Abrimos una ventana SQL Window y escribimos el código siguiente.

DECLARE

Departamentoid NUMBER(4);

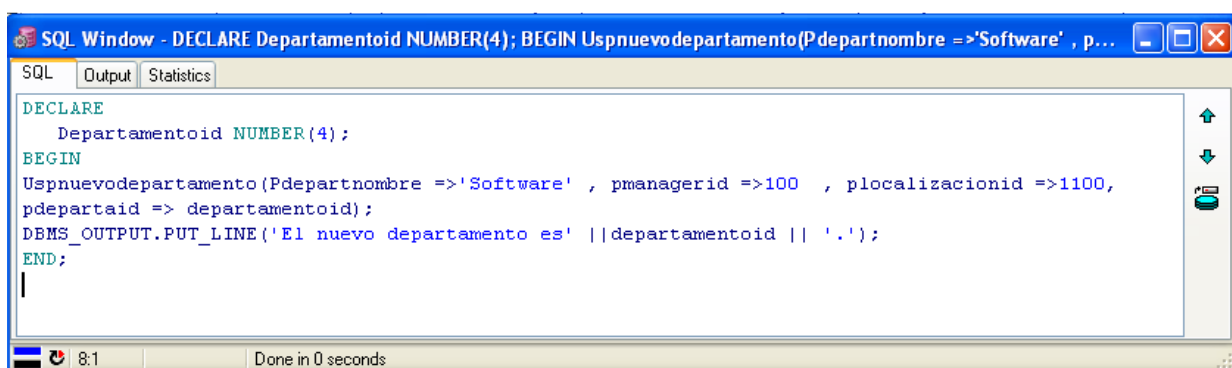
BEGIN

```
Uspnuevodepartamento(Pdepartnombre => 'Software', pmanagerid =>100,
plocalizacionid =>1100, pdepartaid => departamentoid);
DBMS_OUTPUT.PUT_LINE('El nuevo departamento es ' || departamentoid || '.');
COMMIT;
END;
```

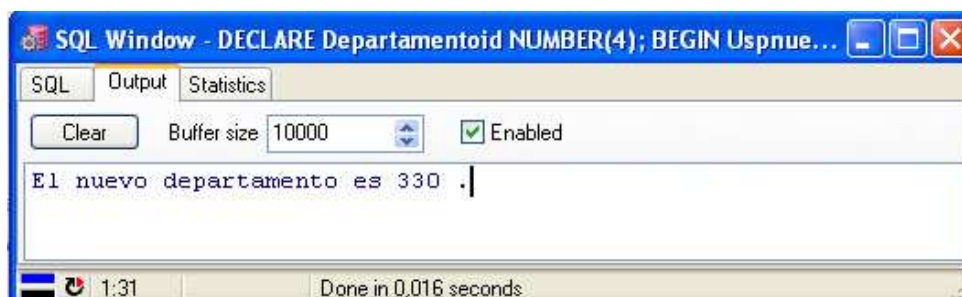
o también **uspnewodepartamento('Software', 100, 1100, pdepartaid);**

o si queremos mediante variables de sustitución

uspnewodepartamento(&vnombre, &vmanager, &vlocal, pdepartid);



Si ejecutamos vemos que nos da el siguiente resultado:



Vemos como se ha creado un nuevo departamento.

Con los procedimientos almacenados podemos implementar la forma en que las aplicaciones interactúan con la base de datos, otorgando el permiso de ejecutar los procedimientos pero no de acceder directamente a las tablas. Al parecido pasa con las funciones.

❖ Funciones

Las funciones a diferencia con los procedimientos devuelven un valor al código que las ha llamado, así como la forma en que pueden ser utilizadas.

Para utilizar un procedimiento almacenado se debe ejecutar a través de una instrucción, sin embargo las funciones se utilizan formando parte de una expresión.

Las funciones que nosotros vamos a crear se comportan igualmente que las funciones integradas en PL/SQL.

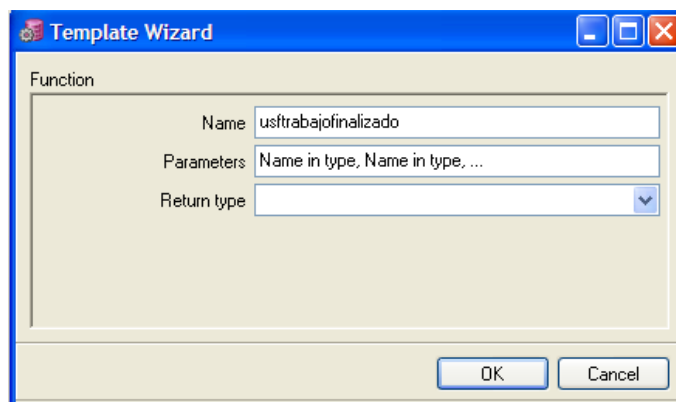
Toda función devuelve un **único valor**.

Para crear una función nos posicionaremos en FUNCTION- NEW e introducimos el nombre de la función.

Ejemplo:

Vamos a crear una función que nos indique si un trabajador ha finalizado su trabajo. Consideramos que ha finalizado en función de la fecha actual. Llamaremos a esta función **usftrabajofinalizado**.

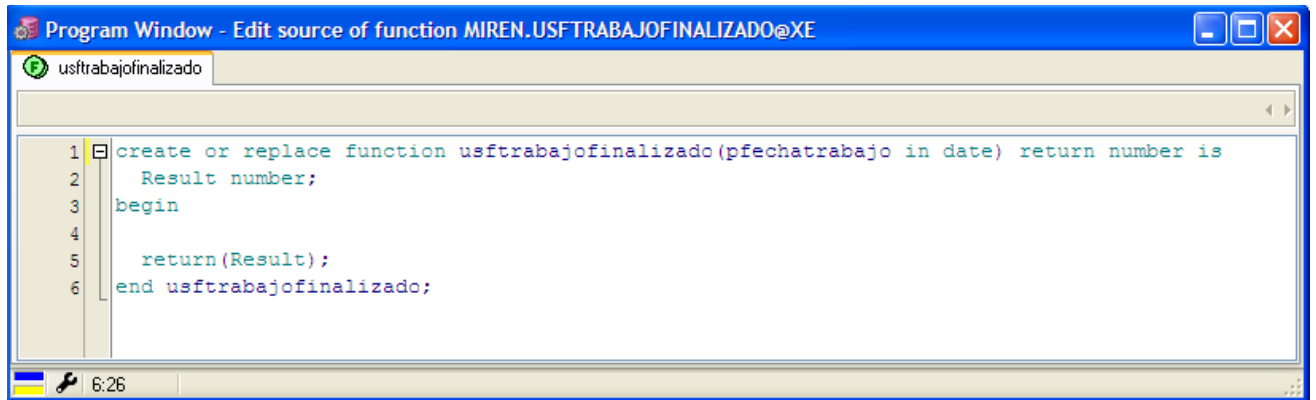
Vemos a diferencia que el procedimiento que nos pide que indiquemos de que tipo es el valor que vamos a devolver.



Seleccionaremos el tipo **Number** para indicar si ha finalizado o no.

Adicionalmente las funciones pueden tener parámetros, tanto de entrada (IN) , salida(OUT) como entrada y salida (IN/OUT) aunque normalmente se utilizarán datos de entrada porque el valor que devuelve la función viene asociado al nombre de la misma, sin necesidad de un parámetro adicional de salida.

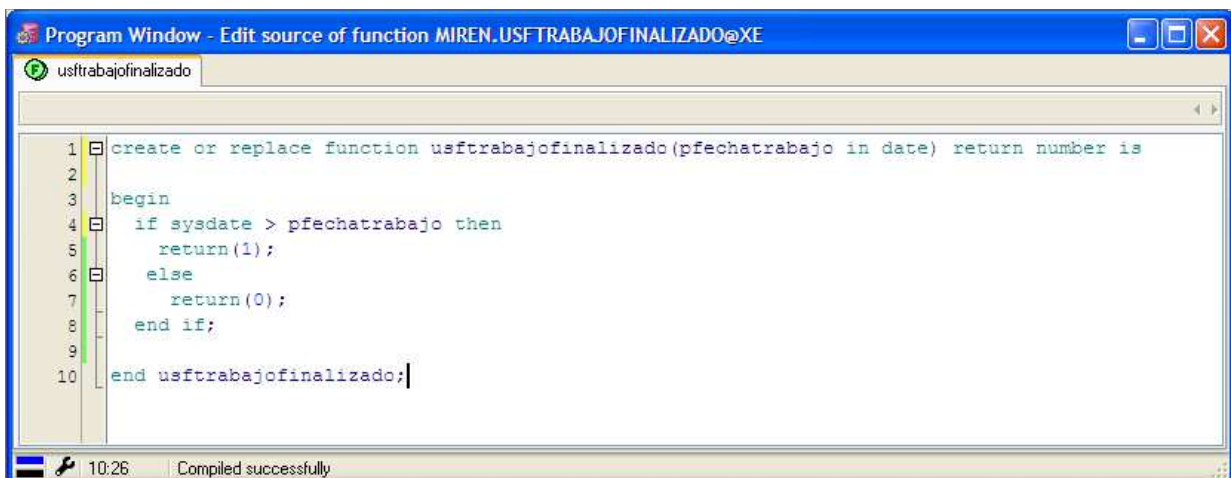
Añadimos un parámetro de entrada pfechatrabajo que será la fecha a comprobar si ha finalizado o no.



Program Window - Edit source of function MIREN.USFTRABAJOFINALIZADO@XE

```
1 create or replace function usftrabajofinalizado(pfechatrabajo in date) return number is
2   Result number;
3 begin
4
5   return(Result);
6 end usftrabajofinalizado;
```

6:26



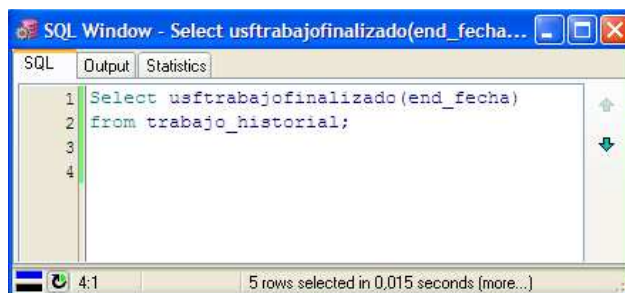
Program Window - Edit source of function MIREN.USFTRABAJOFINALIZADO@XE

```
1 create or replace function usftrabajofinalizado(pfechatrabajo in date) return number is
2
3 begin
4   if sysdate > pfechatrabajo then
5     return(1);
6   else
7     return(0);
8   end if;
9
10 end usftrabajofinalizado;
```

10:26 Compiled successfully

Una vez realizada la función vamos a ver como se utiliza.

Vamos a file – new – sql window



SQL Window - Select usftrabajofinalizado(end_fecha...

```
1 Select usftrabajofinalizado(end_fecha)
2 from trabajo_historial;
```

4:1 5 rows selected in 0,015 seconds (more...)

El resultado será 0 o 1 por cada fila existente en la tabla trabajo_historial.

10.4 Cursores Explícitos.

Se utilizan para trabajar con consultas que pueden devolver más de una fila.

Hay 4 operaciones básicas para trabajar con un cursor:

- **Declaración** del cursor en la zona de declaraciones

CURSOR <nombrecursor> **IS** <sentencia **SELECT**>;

- **Apertura** del cursor en la zona de instrucciones

OPEN <nombrecursor>;

La instrucción open ejecuta automáticamente la sentencia Select asociada y sus resultados se almacenan en las estructuras internas de memoria manejadas por el cursor.

- **Recogida de información almacenada** en el cursor.

FETCH <nombrecursor> **INTO** {<variable>|<listavARIABLES>};

Después de into figurará una variable o una lista de variables que recogerá la información de todas las columnas correspondientes de la cláusula select.

Puede declararse de la forma:

<variable> <nombrecursor>%ROWTYPE;

Cada Fetch recupera una fila y el cursor avanza automáticamente a la fila siguiente en cada nueva instrucción fetch.

- **Cierre del cursor** , cuando el cursor no se va a utilizar hay que cerrarlo.

CLOSE <nombrecursor>;

Ejemplo: se utiliza el cursor para visualizar el nombre y la localidad de todos los departamentos.

DECLARE

CURSOR curl **IS**

SELECT departamento_nombre, ciudad, provincia
FROM departamentos INNER JOIN localizaciones ON
departamentos.localizacion_id= localizaciones.localizacion_id;

v_nom varchar2(30);

v_ciu varchar2(30);

v_pro varchar2(25);

BEGIN

OPEN curl;

LOOP

FETCH curl **INTO** v_nom, v_ciu, v_pro;

EXIT WHEN curl%NOTFOUND;

DBMS_OUTPUT.PUT_LINE(v_nom || ' ' || v_ciu || ' ' || v_pro);

END LOOP;

CLOSE curl;

END;

A diferencia de los cursores implícitos la sentencia select no tiene into.

▪ Atributos del cursor

%FOUND: devuelve verdadero si el último FETCH ha recuperado algún valor; en caso contrario, devuelve falso. Si el cursor no está abierto devuelve error, y si está abierto pero no se ha ejecutado aún el FETCH devuelve null.

En el ejemplo anterior se podía haber puesto en la condición de salida:

```
...
FETCH curl INTO v_nom, v_ciu, v_pro;
WHILE curl%FOUND LOOP
    DBMS_OUTPUT.PUT_LINE(v_nom || ' ' || v_ciu || ' ' || v_pro);
    FETCH curl INTO v_nom, v_ciu, v_pro;
END LOOP;
%NOTFOUND: hace lo contrario que el atributo anterior. Se suele utilizar como condición de salida e bucles:
```

```
...
EXIT WHEN curl%NOTFOUND;
```

```
...
%ROWCOUNT: devuelve el número de filas recuperadas hasta el momento por el cursor (número de fetch realizados satisfactoriamente);
```

%ISOPEN: devuelve verdadero si el cursor está abierto.

Ejemplo: Visualizar los apellidos de los empleados pertenecientes al departamento 100 numerándolos secuencialmente.

DECLARE

```
CURSOR cure IS
  SELECT nombre, apellido FROM empleados WHERE departamento_id=100;
  v_ape VARCHAR2(25);
  v_nom VARCHAR2(20);
```

BEGIN

```
  OPEN cure;
  LOOP
    FETCH cure INTO v_nom, v_ape;
    DBMS_OUTPUT.PUT_LINE(cure%ROWCOUNT || '. ' || v_nom || ' ' || v_ape);
    EXIT WHEN cure%NOTFOUND;
  END LOOP;
```

```
CLOSE cure;
END;
```

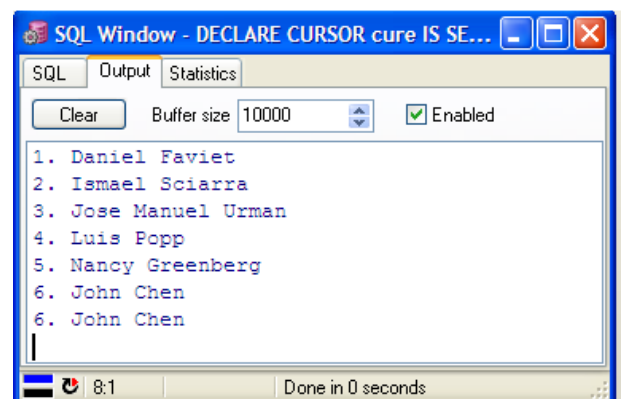
Vemos que la última fila se repite. Esto es debido a que fetch cuando no recupera una fila :

- no incrementa el valor de %rowcount
- no se sobrescribe el valor de las variables del cursor.

Luego no es correcto el código. Se deberá controlar de otra forma.

DECLARE

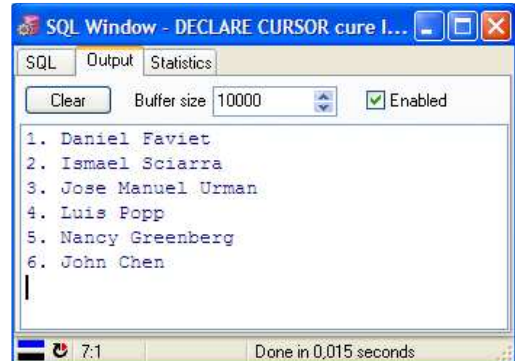
```
CURSOR cure IS
  SELECT nombre, apellido FROM empleados WHERE departamento_id=100;
  v_ape VARCHAR2(25);
  v_nom VARCHAR2(20);
```



```

BEGIN
  OPEN cure;
  FETCH cure INTO v_nom, v_ape;
  WHILE cure%found LOOP
    DBMS_OUTPUT.PUT_LINE(TO_CHAR(cure%ROWCOUNT || ' ' || v_nom || ' ' || v_ape));
    FETCH cure INTO v_nom, v_ape;
  END LOOP;
CLOSE cure;
END;

```



▪ Variables de acoplamiento en el manejo de cursores.

Su función es la misma que las variables de sustitución, es decir, permite crear una código más abierto, de forma que, en la condición del select permite introducir valores diferentes en vez de ser uno fijo.

Ejemplo: El caso anterior en vez de visualizar los empleados del departamento 100 , vamos a pedir los empleados del departamento que el usuario quiera.

```

DECLARE
  v_depart number(4,0);
  CURSOR cure IS
    SELECT nombre, apellido FROM empleados WHERE departamento_id=&v_depart;
  v_ape VARCHAR2(25);
  v_nom VARCHAR2(20);
BEGIN
  OPEN cure;
  FETCH cure INTO v_nom, v_ape;
  WHILE cure%found LOOP
    DBMS_OUTPUT.PUT_LINE(TO_CHAR(cure%ROWCOUNT || ' ' || v_nom || ' ' || v_ape));
    FETCH cure INTO v_nom, v_ape;
  END LOOP;
CLOSE cure;
END;

```

**/* también se puede declarar v_reg cure%ROWTYPE;
en vez de las dos variables y hacer referencia a ellas
de la forma siguiente v_reg.nombre, v_reg.apellido/**

Recordar que la variable de sustitución en un procedimiento/función no se puede utilizar, luego esta variable de acoplamiento será a través de un parámetro.

▪ Cursores FOR .. LOOP

Es otra forma de trabajar con los cursores donde simplifica las tareas de apertura, cierre... En este caso solamente hay que realizar dos pasos:

- Declarar el cursor en la sección declarativa

CURSOR <nombrecursor> IS <sentencia SELECT>;

- Procesar el cursor

```
FOR <nombrevar> IN <nombrercursor> LOOP
...
END LOOP;
```

Donde nombrevar es el nombre de la variable que creará el bucle para recoger los datos del cursor.

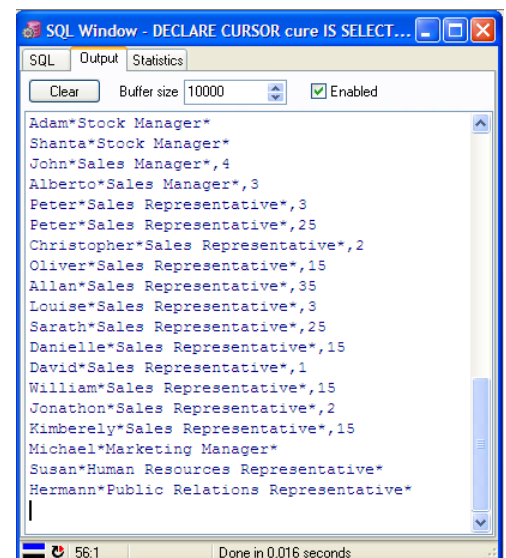
Al entrar al bucle:

- se abre el cursor de manera automática.
- se declara implícitamente la variable nombrevar de tipo nombrevar%ROWTYPE y se ejecuta un FETCH implícito, cuyo resultado quedará en nombrevar.
- se ejecutara el bucle y cuando se termine se cerrará el cursor de manera automática.

Ejemplo: Vamos a visualizar el apellido , trabajo y comisión de los empleados salario es superior a 5000.

```
DECLARE
  CURSOR cure IS
    SELECT nombre, trabajo_nombre, comision FROM
empleados INNER JOIN trabajos on
  empleados.trabajo_id=trabajos.trabajo_id
  WHERE salario>5000;

BEGIN
  FOR v_reg IN cure LOOP
    DBMS_OUTPUT.PUT_LINE(v_reg.nombre || '*' ||
v_reg.trabajo_nombre || '*' || v_reg.comision);
  END LOOP;
END;
```



Nota: la v_reg es local al bucle, es decir, al salir del bucle la variable no estará disponible.

▪ Atributos en cursores implícitos

Oracle abre implícitamente un cursor cuando procesa un comando SQL que no esta asociado a un cursor explícito. El cursor implícito se llama **SQL** y dispone también de los cuatro atributos anteriormente mencionados, y que nos dan información sobre la ejecución de los comandos SELECT INTO, INSERT, UPDATE y DELETE.

El valor de los atributos del cursor SQL se refiere, en cada momento, a la última orden SQL.

SQL%FOUND: dará TRUE si el último SELECT INTO, INSERT, UPDATE o DELETE ha afectado a una o varias filas.

SQL%NOTFOUND: dará TRUE si el último SELECT INTO, INSERT, UPDATE o DELETE ha fallado(no ha afectado a ninguna fila).

SQL%ROWCOUNT: devuelve el número de filas afectadas por el último SELECT INTO, INSERT, UPDATE o DELETE.

SQL%ISOPEN: siempre dará FALSO, ya que oracle cierra automáticamente el cursor después de cada orden SQL.

Ejemplo

DECLARE

v_dpto departamentos.departamento_nombre%TYPE:='MARKETING'; -- NO EXISTE

v_loc localizaciones.localizacion_id%TYPE;

BEGIN

SELECT localizacion_id INTO v_loc FROM localizaciones WHERE ciudad='Roma';

UPDATE departamentos SET localizacion_id=v_loc

WHERE departamento_nombre=v_dpto;

IF SQL%NOTFOUND THEN

DBMS_OUTPUT.put_line('Error en la actualizacion');

END IF;

DBMS_OUTPUT.put_line('Continua el programa');

SELECT localizacion_id INTO v_loc FROM departamentos WHERE
departamento_nombre=v_dpto;

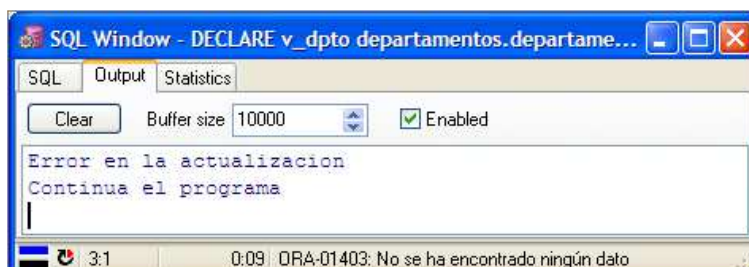
IF SQL%NOTFOUND THEN

DBMS_OUTPUT.put_line('Nunca pasará por aqui');

END IF;

END;

Resultado de ejecución



Tendría que haber impreso 'Nunca pasara por aquí' sin embargo un select .. into nunca dará no_data_found y por lo tanto SQL%Found es siempre verdadero. Luego el if no se cumple.

▪ Excepciones

Las excepciones sirven para tratar errores en tiempo de ejecución, así como errores y situaciones definidas por el usuario.

Cuando se produce un error PL/SQL levanta una excepción y pasa el control a la sección EXCEPTION, donde buscará un manejador WHEN para la excepción o uno genérico (WHEN OTHERS) y dará por finalizada la ejecución del bloque actual.

El formato de la sección EXCEPTION es:

```
...
EXCEPTION
  WHEN <nombreexcepcion1> THEN
    <instrucciones1>;
  WHEN <nombreexcepcion2> THEN
    <instrucciones2>;
...
[WHEN OTHERS THEN
  <instrucciones>;]
```

Excepciones internas predefinidas

Están predefinidas por Oracle. Se disparan automáticamente al producirse determinados errores. En la siguiente tabla se incluyen las excepciones más frecuentes con los códigos de error correspondientes:

Código error Oracle	Valor de SQL CODE	Excepción	Se disparan cuando...
ORA-06530	-6530	ACCESS_INTO_NULL	Se intenta acceder a los atributos de un objeto no inicializado.
ORA-06531	-6531	COLLECTION_IS_NULL	Se intenta acceder a elementos de una colección que no ha sido inicializada.
ORA-06511	-6511	CURSOR_ALREADY_OPEN	Intentamos abrir un cursor que ya se encuentra abierto.
ORA-00001	-1	DUP_VAL_ON_INDEX	Se intenta almacenar un valor que crearía duplicados en la clave primaria o en una columna con la restricción UNIQUE.
ORA-01001	-1001	INVALID_CURSOR	Se intenta realizar una operación no permitida sobre un cursor (por ejemplo, cerrar un cursor que no estaba abierto).
ORA-01722	-1722	INVALID_NUMBER	Fallo al intentar convertir una cadena a un valor numérico.
ORA-01017	-1017	LOGIN_DENIED	Se intenta conectar a ORACLE con un usuario o una clave no válidos.
ORA-01012	-1012	NOT_LOGGED_ON	Se intenta acceder a la base de datos sin estar conectado a Oracle.
ORA-01403	+100	NO_DATA_FOUND	Una sentencia SELECT ... INTO ... no devuelve ninguna fila.
ORA-06501	-6501	PROGRAM_ERROR	Hay un problema interno en la ejecución del programa.
ORA-06504	-6504	ROWTYPE_MISMATCH	La variable del cursor del HOST y la variable del cursor PL/SQL pertenecen a tipos incompatibles.
ORA-06533	-6533	SUBSCRIPT_OUTSIDE_LIMIT	Se intenta acceder a una tabla anidada o a un <i>array</i> con un valor de índice ilegal (por ejemplo, negativo).
ORA-06500	-6500	STORAGE_ERROR	El bloque PL/SQL se ejecuta fuera de memoria (o hay algún otro error de memoria).
ORA-00051	-51	TIMEOUT_ON_RESOURCE	Se excede el tiempo de espera para un recurso.
ORA-01422	-1422	TOO_MANY_ROWS	Una sentencia SELECT ... INTO ... devuelve más de una fila.
ORA-06502	-6502	VALUE_ERROR	Un error de tipo aritmético, de conversión, de truncamiento, etcétera.
ORA-01476	-1476	ZERO_DIVIDE	Se intenta la división entre cero.

No hay que declararlas en la sección DECLARE. Únicamente debemos incluir los manejadores WHEN con el tratamiento para cada excepción y/o un manejador genérico.

DECLARE

.....
BEGIN

.....
EXCEPTION

WHEN **NO_DATA_FOUND** THEN

DBMS_OUTPUT.PUT_LINE('Error datos no encontrados');

WHEN **TOO_MANY_ROWS** THEN

DBMS_OUTPUT.PUT_LINE('Error demasiadas filas');

WHEN OTHERS THEN

DBMS_OUTPUT.PUT_LINE('Error ');

END;

Excepciones definidas por el usuario

Las excepciones definidas por el usuario se usan para tratar condiciones de error definidas por el programador.

Para su utilización hay que seguir tres pasos:

1. Se declaran en la sección DECLARE de la forma siguiente:

<nombreexcepcion> EXCEPTION;

2. Se disparan o levantan en la sección ejecutable del programa con la orden RAISE:

RAISE <nombreexcepcion>;

3. Se tratan en la sección EXCEPTION según el formato ya conocido:

WHEN <nombreexcepcion> THEN <tratamiento>;

Ejemplo

DECLARE

 Importe_erroneo EXCEPTION;

BEGIN

 IF precio NOT BETWEEN precio_min AND precio_max

THEN

 RAISE importe_erroneo;

END IF;

...

EXCEPTION

...

 WHEN importe_erroneo THEN

 DBMS_OUTPUT.PUT_LINE('Importe erróneo.Venta cancelada.');

...

END;

La instrucción RAISE se puede usar varias veces en el mismo bloque con la misma o con distintas excepciones, pero solamente puede haber un manejador WHEN para cada excepción.

10.5 Paquetes.

Oracle permite utilizar el concepto de paquete (package) para agrupar objetos de programación relacionados, como variables, cursores, procedimientos y funciones.

Agrupando todas estas unidades en un paquete, se obtiene muchas ventajas, pero sobretodo destaca la posibilidad de dividir lógicamente la programación que se almacena en el servidor de la base de datos.

Un paquete se divide en dos partes:

- Especificación: se encuentra la declaración de variables, cursores, procedimientos y funciones que el paquete hace disponibles. Es la parte **pública** del paquete.
- Cuerpo: donde se encuentra los detalles concretos de implantación de los cursores, procedimientos y funciones además de otras unidades de código que se utilizan ahí y que no son accesibles fuera del paquete. Es la parte **privada** del paquete.

La implantación queda oculta y puede modificarse sin necesidad de que las aplicaciones que utilizan el paquete tengan que hacerlo.

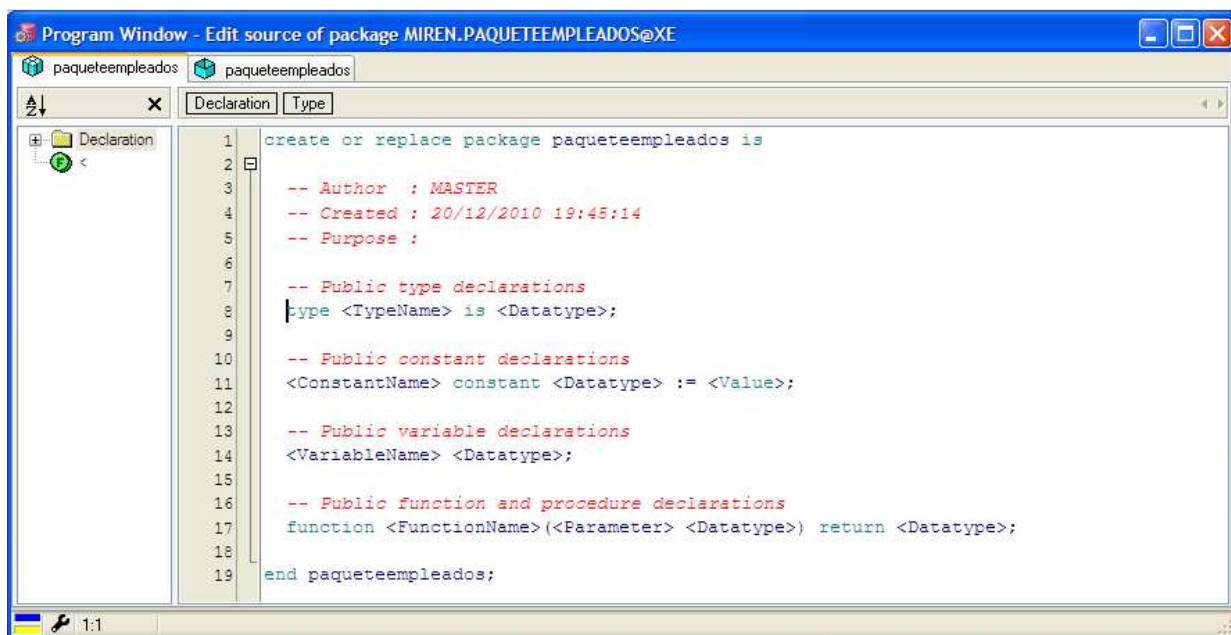
Para ello seleccionamos packages botón derecho new.

Cuando se solicita cualquier componente de un paquete, el código de todas las unidades incluidas en él es cargado en memoria de forma que la primera vez puede que cueste más en hacerlo, pero llamadas posteriores son más rápidas al ya estar disponible en memoria.

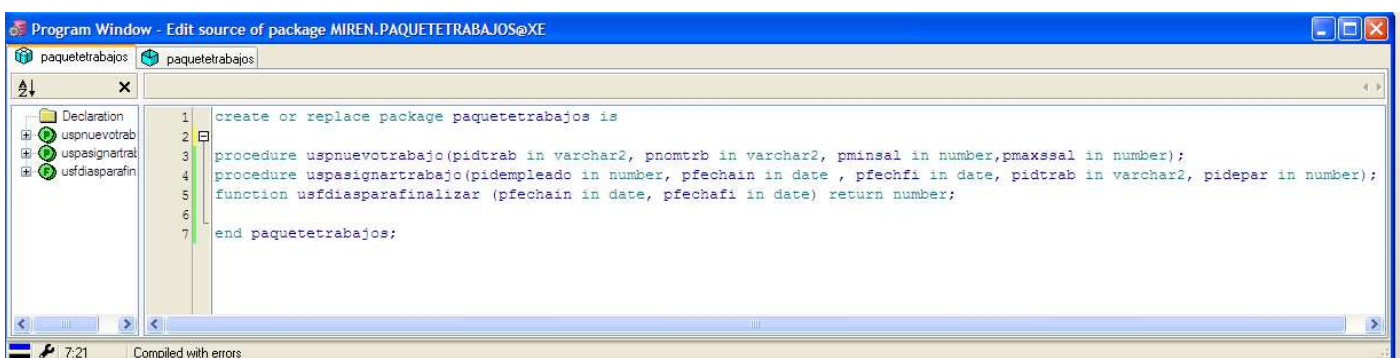
El diseño del paquete deber agrupar unidades que tengan relación. Por lo tanto, el nombre del paquete debe hacer referencia a la naturaleza de la funcionalidad que ofrece.

Ejemplo: **PaqueteTrabajos**.

Sobre packages pulsamos botón derecho y seleccionamos new. Ponemos como nombre **PaqueteTrabajos**, ya que todo va a estar relacionado con trabajos.




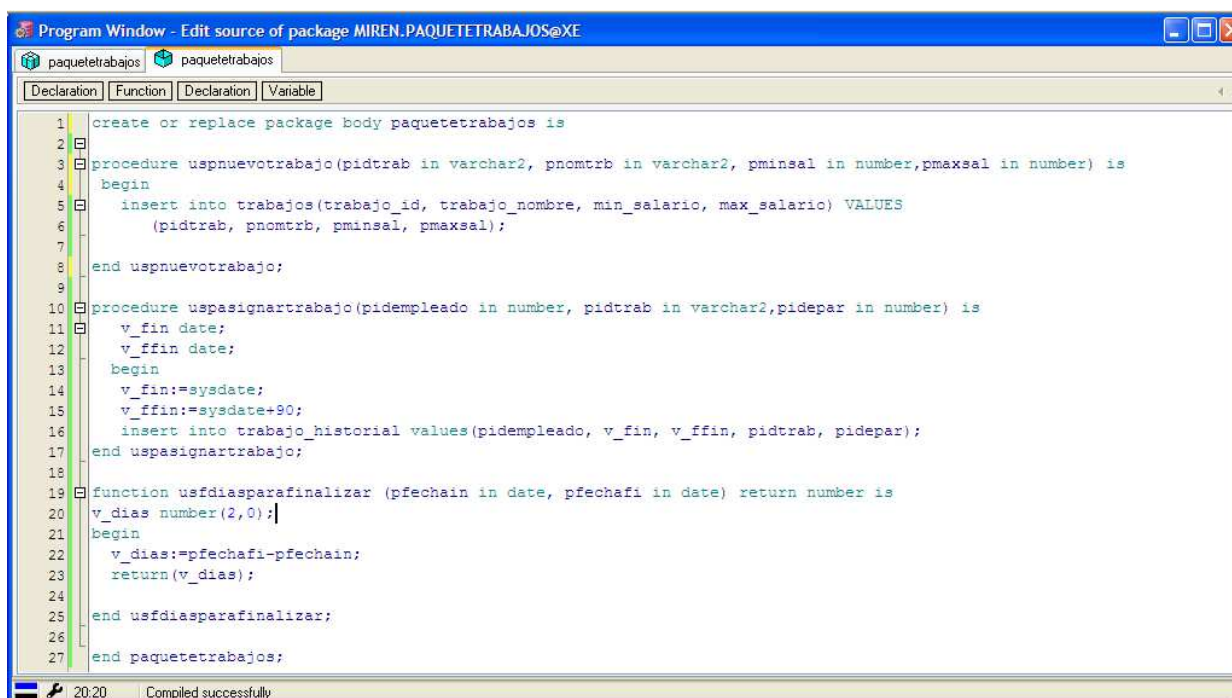
Vemos que nos da información sobre como declarar constantes, variables, funciones y procedimientos. Nosotros vamos a borrar todo y vamos a crear nuestros objetos.



Los que vean el paquete podrán utilizar dos procedimientos y una función. Hay que fijarse que sólo necesitan esto y no los detalles concretos de implementación. Saben que si quieren saber los días que faltan para finalizar el trabajo solamente hace falta utilizar la función **usfdiasparafinalizar**; cuando quieran dar de alta a un trabajo el procedimiento **usp nuevotrabajo**, etc.

En la especificación del paquete vemos los parámetros a utilizar , el tipo que devuelve la función.. pero ningún detalle de su código, que queda "oculto" en el cuerpo del paquete. Ahora hay que crear el cuerpo del paquete

A continuación creamos el cuerpo del paquete y para ello seleccionamos el botón  **paquetetrabajos** . En él completaremos el código de los procedimientos, funciones y cursores que hayamos incluido en la especificación del paquete e incluso otras unidades de código que necesitemos para ello, pero que sólo estarán disponibles en el interior del cuerpo del paquete.



```

1 create or replace package body paquetetrabajos is
2
3 procedure usp nuevotrabajo(pidtrab in varchar2, pnomtrb in varchar2, pminsal in number, pmaxsal in number) is
4 begin
5 insert into trabajos(trabajo_id, trabajo_nombre, min_salario, max_salario) VALUES
6 (pidtrab, pnomtrb, pminsal, pmaxsal);
7
8 end usp nuevotrabajo;
9
10 procedure usp asignartrabajo(pidempleado in number, pidtrab in varchar2, pidepar in number) is
11 v_fin date;
12 v_ffin date;
13 begin
14 v_fin:=sysdate;
15 v_ffin:=sysdate+90;
16 insert into trabajo_historial values(pidempleado, v_fin, v_ffin, pidtrab, pidepar);
17 end usp asignartrabajo;
18
19 function usfdiasparafinalizar (pfechain in date, pfechafi in date) return number is
20 v_dias number(2,0);
21 begin
22 v_dias:=pfechafi-pfechain;
23 return(v_dias);
24 end usfdiasparafinalizar;
25
26
27 end paquetetrabajos;

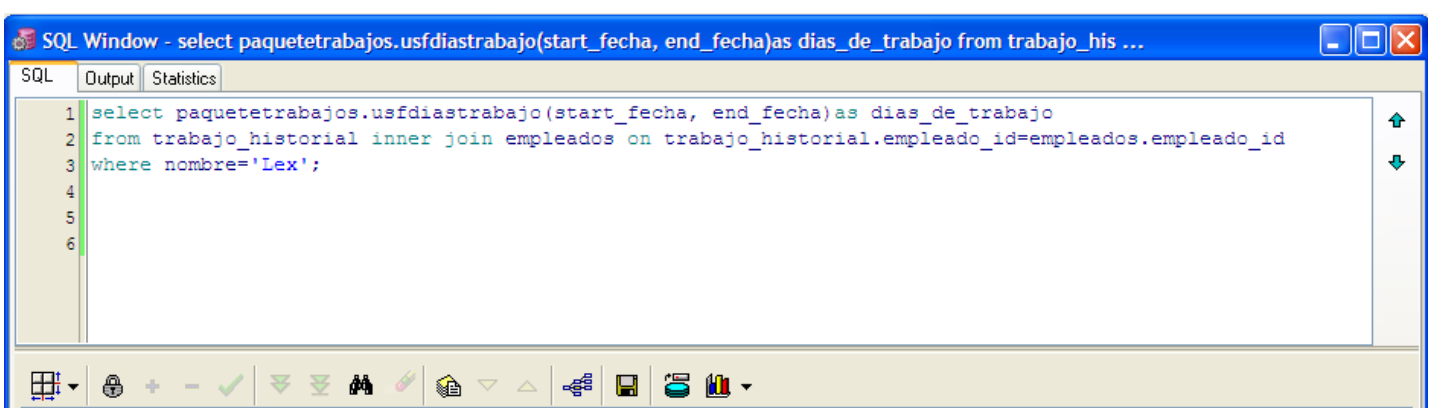
```

20:20 Compiled successfully

Aquí no se ha más que incluir ahora en el cuerpo del paquete la implementación o código de cada uno de los procedimientos almacenados y de la función, exactamente igual que hicimos al crear estos subprogramas de forma independiente. Una vez creado el paquete lo compilamos para ver si hay algún error.

Una vez creado vamos a ver como se utiliza. Es parecido a cómo se hace con un procedimiento independiente, pero ahora tendremos que indicar el nombre del paquete.

Queremos saber cuantos días de trabajo ha realizado el empleado cuyo nombre es **Lex**. Para ello utilizamos la función creada en el paquete **trabajos.usfdiastrabajo**.



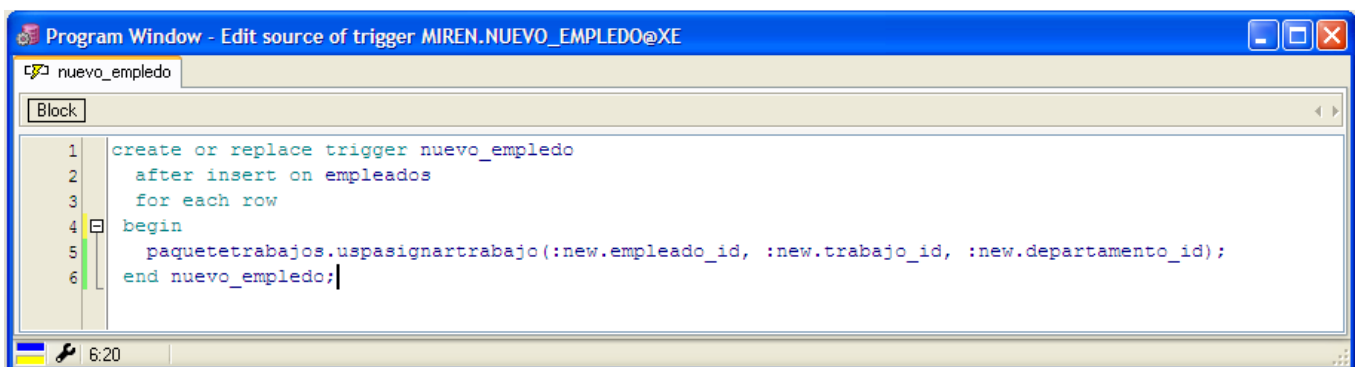
```

SQL Window - select paquete_trabajos.usfdiastrabajo(start_fecha, end_fecha)as dias_de_trabajo from trabajo_his ...
SQL Output Statistics
1 select paquete_trabajos.usfdiastrabajo(start_fecha, end_fecha)as dias_de_trabajo
2 from trabajo_historial inner join empleados on trabajo_historial.empleado_id=empleados.empleado_id
3 where nombre='Lex';
4
5
6

```

Vamos a crear un trigger de forma que cuando se inserte un nuevo empleado se le dé de alta en el historial de trabajo.

Para ello desde sqldeveloper seleccionamos trigger , botón derecho new



Como se puede ver el trigger utiliza el procedimiento uspassignartrabajo del paquete de trabajos.

Observar que en oracle la sintaxis de new y old cambia. Hay que poner : delante.

Ahora insertamos un nuevo empleado y comprobaremos como se dispara el trigger que ejecutará el procedimiento creado en el paquetetrabajos.





```

insert into empleados values(empleados_seq.nextval, 'Miren', 'Etxeguren',
'xxx@gmail.com', null,
sysdate, 'AD_PRES', 25000, NULL, 101, 40);
commit;

```

EDITAR	EMPLEADO_ID	NOMBRE	APELLIDO	EMAIL	TELEFONO	CONTRATO_DATE	TRABAJO_ID	SALARIO	COMISION	MANAGER_ID
	211	Miren	Etxeguren	zubiri@gmail.com	-	20/12/10	AD_PRES	25000	-	101
	100	Steven	King	SKING	515.123.4567	17/06/87	AD_PRES	24000	-	-
	102	Lex	De Haan	LDEHAAN	515.123.4569	13/06/93	AD_VP	17000	-	100
	105	David	Austin	DAUSTIN	590.423.4569	25/06/97	IT_PROG	4800	-	103

y automáticamente se ha creado

EDITAR	EMPLEADO_ID	START_FECHA	END_FECHA	TRABAJO_ID	DEPARTAMENTO_ID
	101	21/09/89	27/10/93	AC_ACCOUNT	110
	101	28/10/93	15/03/97	AC_MGR	110
	176	24/03/98	31/12/98	SA_REP	80
	211	20/12/10	20/03/11	AD_PRES	40