

A photograph of three students sitting around a wooden table in a classroom or study area. A woman on the left is looking at a laptop screen. A woman in the center is smiling and looking at the same screen. A man on the right is also smiling and looking at the screen. There are two laptops on the table, one open and one closed. There are also some notebooks and glasses of water. The background is a chalkboard.

# UF-7 PROGRAMACIÓN AVANZADA DE ACCESO A DATOS RAÚL SALGADO VILAS



## UF-7 PROGRAMACIÓN AVANZADA DE ACCESO A DATOS

### ❏ Objetivos

- Entender la sintaxis de una bloque almacenado.
- Conocer los tipos de parámetros que recibe un bloque almacenado.
- Entender la diferencia entre parámetros In y OUT.
- Entender la relación de confirmación de actualizaciones entre un bloque llamante y un bloque llamado.
- Comprender cómo se ejecuta una función y en qué tipo de sentencias PL/SQL se pueden incorporar.





## UF-7 PROGRAMACIÓN AVANZADA DE ACCESO A DATOS

- ❑ Introducción a Bloques Almacenados: Un código PL / SQL se dice que está almacenado, porque al compilarle, el motor de Oracle lo guarda como un componente más de la Base de Datos, y por tanto se puede solicitar su ejecución desde otro PL / SQL, o desde otros entornos, siempre y cuando se haya establecido la conexión a nuestra base de datos, tenga los permisos oportunos y esté compilado correctamente.
- ❑ El mandato SQL/DDL empleado para su generación, es CREATE y los bloques almacenados que se pueden crear son:
  - Procedimientos : CREATE PROCEDURE
  - Funciones: CREATE FUNCTION
  - Paquetes: CREATE PACKAGE
  - Disparadores: CREATE TRIGGER
- ❑ Los bloques almacenados se compilan, y tengan o no tengan errores de compilación se guardan en el conjunto de objetos de mi usuario, pero:
  - Si tienen errores de compilación su STATUS estará INVALID, y no se le puede llamar a ejecutar.
  - Si no tiene errores de compilación su STATUS será VALID, y se le puede llamar a ejecutar (siempre que tengas permiso para ello).
  - La columna STATUS, se encuentra en la Vista del diccionario USER\_OBJECTS / ALL\_OBJECTS.
  - Procedimientos y Funciones almacenados son bloques PL / SQL con un nombre, a los que se pueden pasar parámetros y pueden ser invocados desde otros bloques, tanto anónimos como almacenados. Estos bloques se guardan en la base de datos una vez creados, sin(o con) errores de compilación.





## UF-7 PROGRAMACIÓN AVANZADA DE ACCESO A DATOS

❑ PL / SQL distingue entre los dos tipos de subprogramas:

- Procedimientos, que se utilizan para ejecutar una acción;
- Funciones, que siempre retornan un valor.

❑ Las partes fundamentales de un subprograma son:

- Las especificaciones: donde se define el tipo del subprograma (procedimiento o función), el nombre del mismo y los parámetros de entrada y / o salida. Los parámetros son opcionales.
- Una parte declarativa (No se usa la cláusula DECLARE): donde se definen todos los elementos que forman parte del subprograma: variables, constantes, cursores, tipos de datos, subprogramas, etc.
- Una parte de Ejecución: que comienza con BEGIN donde se realizan todas las acciones, sentencias de control y sentencia SQL.
- Una parte de excepciones o de control de errores (opcional), se especifique las acciones a tomar en caso de que se produzca un error en la ejecución del subprograma.





## UF-7 PROGRAMACIÓN AVANZADA DE ACCESO A DATOS

- ❑ Utilidad de procedimientos y Funciones almacenados: Como sabemos de SQL, todas las tablas que crea un usuario, son propiedad de ese usuario, y sólo de él; si quiero que otros usuarios puedan consultar, y/o actualizar información de cualquiera de mis tablas, les tengo que dar permisos, por ejemplo:

```
GRANT SELECT,INSERT,UPDATE ON EMPLOYEES TO ALUMNO;
```

- ❑ Autorizo al usuario ALUMNO a leer, insertar y modificar en mi tabla employees.
- ❑ Pero si no quiero (o no me fío del conocimiento que tenga el usuario ALUMNO de SQL), tengo la posibilidad de crear Bloques PL/SQL almacenados, es decir, procedimientos y funciones, para encerrar ahí las sentencias SQL, y le doy autorización a ejecutar estos bloques, desde PL o desde cualquier lenguaje de programación (como por ejemplo java):

```
GRANT EXECUTE ON NOMBRE_PROCEDIMEINTO TO ALUMNO;  
GRANT EXECUTE ON NOMBRE_FUNCION TO ALUMNO;
```







## UF-7 PROGRAMACIÓN AVANZADA DE ACCESO A DATOS

- ❑ Levantar excepciones especiales y su tratamiento: Los bloques anónimos procedimientos, Funciones y disparadores, cuando levantan excepciones, lo hacen a través del el procedimiento RAISE\_APPLICATION\_ERROR que permite, no solo levantar la excepción, sino definir mensajes de error del tipo ORA- por el usuario. La sintaxis es:
  - RAISE\_APPLICATION\_ERROR (numero\_error, mensaje\_error);
  - RAISE\_APPLICATION\_ERROR está definido en el paquete DBMS\_STANDARD por lo que se puede invocar desde cualquier programa o subprograma PL/SQL almacenado.
  - Cuando RAISE\_APPLICATION\_ERROR es invocado el subprograma acaba y devuelve el número de error y el mensaje a la aplicación o bloque que le invocó.
  - El numero\_error tiene el rango de -20000 a -20999 y mensaje puede tener una longitud máxima de 2048 bytes.





## UF-7 PROGRAMACIÓN AVANZADA DE ACCESO A DATOS

❑ Tratamiento por el bloque llamante: El bloque que llama a otro que le levanta una excepción de tipo `RAISE_APPLICATION_ERROR(-20XXX, 'mensaje_error')`, tiene dos formas de tratarle:

➤ De forma genérica, capturándole a través del `WHEN OTHERS THEN`.

```
.....  
EXCEPTION  
    WHEN OTHERS THEN  
        DBMS_OUTPUT.PUT_LINE('ERROR GENERAL : ' || SQLERRM);  
END;  
/
```

```
-- CASO DE RECIBIR UN ERROR LA SALIDA SERIA  
'ERROR GENERAL : -20XXX MENSAJE ERROR CORRESPONDIENTE'
```





## UF-7 PROGRAMACIÓN AVANZADA DE ACCESO A DATOS

❑ Tratamiento por el bloque llamante: El bloque que llama a otro que le levanta una excepción de tipo `RAISE_APPLICATION_ERROR(-20XXX, 'mensaje_error')`, tiene dos formas de tratarle:

➤ De forma específica, capturando a través de un nombre de excepción. Supongamos que un bloque almacenado detecta que algo no va bien y nos manda un código Oracle -20100, con el mensaje 'error de proceso'. Y queremos capturarlo a través de un tratamiento específico; procederemos de la siguiente forma:

```
DECLARE
    proceso_erroneo  EXCEPTION;
    .....
    PRAGMA EXCEPTION_INIT(proceso_erroneo, -20100);
BEGIN
    procedimiento1;

EXCEPTION
    WHEN proceso_erroneo THEN
        DBMS_OUTPUT.PUT_LINE('ERROR: ' || SQLERRM);

END;
```

- ❑ Declaramos la excepción `proceso_erroneo`.
- ❑ Declaramos un PRAGMA, directiva para oracle, `EXCEPTION_INIT`, asignando el código -20100 a mi excepción.
- ❑ Llamo al procedimiento "procedimiento1".
- ❑ Este me levanta un código -20100, y el sistema se lo asigna a mi excepción.
- ❑ La capturo: `when proceso_erroneo Then...`
- ❑ Y pongo las instrucciones necesarias para tratar la excepción.







## UF-7 PROGRAMACIÓN AVANZADA DE ACCESO A DATOS

❑ Procedimientos Almacenados; la sintaxis para la definición de un procedimiento es:

```
CREATE [OR REPLACE]
PROCEDURE Nombre_procedimiento [(declaración de parámetros)]
    [AUTHID {DEFINER | CURRENT_USER}]
{IS | AS}
    [PRAGMA AUTONOMOUS_TRANSACTION;]
    [Declaraciones locales de tipos, variables, etc]
BEGIN
    Sentencias ejecutables del procedimiento
[EXCEPTION
    Excepciones definidas y las acciones de estas excepciones]
END [Nombre_procedimiento];
/
```

❑ **CREATE OR REPLACE:** Permite crear un procedimiento STANDALONE (No forma parte de un paquete) y guardarlo dentro de la base de datos. Si se crea un procedimiento que ya existe dará error por ello se utiliza la cláusula OR REPLACE. Si el procedimiento no existe se creará y si ya existe se reemplazará.

**AUTHID:** La cláusula AUTHID determina si un procedimiento se ejecuta con los privilegios del usuario que lo ha creado (por defecto) o si con los privilegios del usuario que lo invoca. También determina si las referencias no cualificadas las resuelve en el esquema del propietario del procedimiento o de quién lo invoca. Para especificar que se ejecute con los permisos de quién lo invoca se utiliza la cláusula CURRENT\_USER.

❑ **Pragma AUTONOMOUS\_TRANSACTION:** Marca el procedimiento como autónomo. Un procedimiento autónomo permite realizar COMMIT o ROLLBACK de las sentencias SQL propias sin afectar a la transacción que lo haya llamado. Si en ejecución no se encuentra la sentencia COMMIT o ROLLBACK provocará una excepción





## UF-7 PROGRAMACIÓN AVANZADA DE ACCESO A DATOS

- ❑ Procedimientos Almacenados; la sintaxis para la definición de un procedimiento es:

```
CREATE [OR REPLACE]
PROCEDURE Nombre_procedimiento [(declaración de parámetros)]
    [AUTHID {DEFINER | CURRENT_USER}]
{IS | AS}
    [PRAGMA AUTONOMOUS_TRANSACTION;]
    [Declaraciones locales de tipos, variables, etc]
BEGIN
    Sentencias ejecutables del procedimiento
[EXCEPTION
    Excepciones definidas y las acciones de estas excepciones]
END [Nombre_procedimiento];
/
```

- ❑ NO DECLARE: A diferencia de los Bloques Anónimos, NO se usa la cláusula DECLARE puesto que va implícita en el IS o el AS. No existe diferencia en utilizar el IS o el AS.





## UF-7 PROGRAMACIÓN AVANZADA DE ACCESO A DATOS

- ❑ Este es el procedimiento más pequeño que se puede hacer sin que de errores de compilación, eso sí no hace nada:

```
CREATE OR REPLACE PROCEDURE no_HACE_NADA AS  
BEGIN  
    NULL;  
END IMP_LIN;
```





## UF-7 PROGRAMACIÓN AVANZADA DE ACCESO A DATOS

- ❑ Parámetros: Un procedimiento puede o no tener parámetros de entrada, si no tiene parámetros no es necesario los paréntesis, ni en la cabecera del procedimiento ni en la llamada al procedimiento. Si lleva parámetros van entre paréntesis y separados por comas.
- ❑ Para definir parámetros, se especifica el nombre de la variable y a continuación el TIPO de parámetro ORACLE y/o PL/SQL, sin precisión:

DATE, VARCHAR2, CHAR, DECIMAL , nombre\_tabla.nombre\_columna%type,  
nombre\_tabla%ROWTYPE...

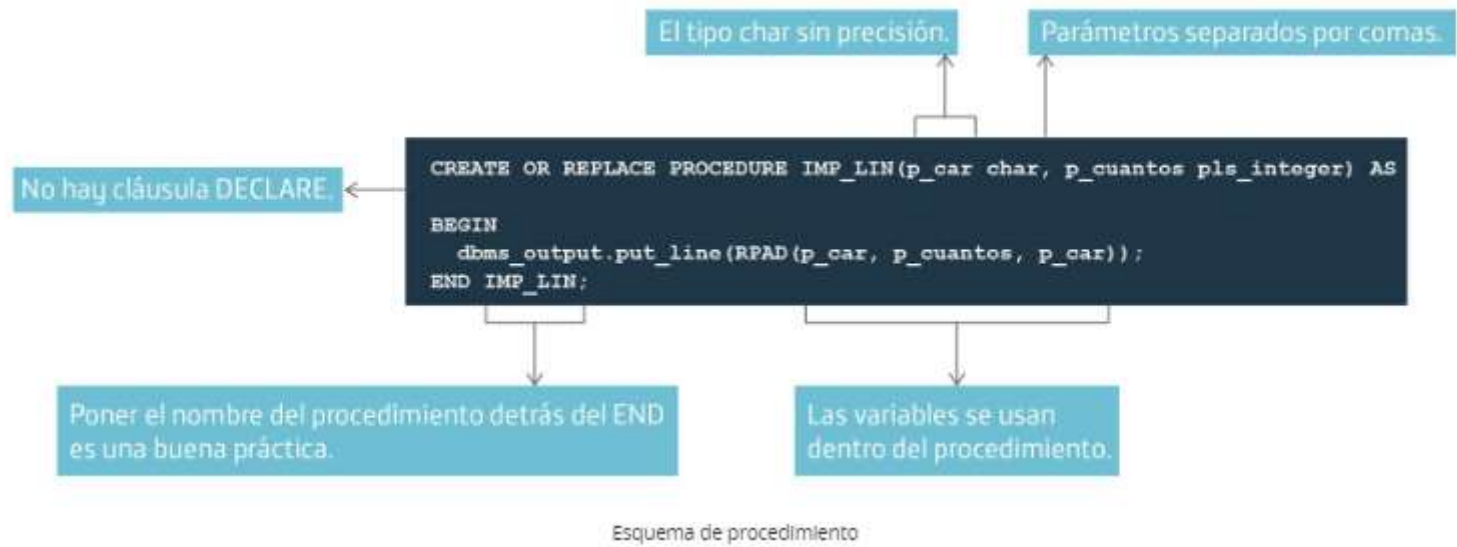
- ❑ Ejemplo: vamos a crear un procedimiento en mi usuario(recordad que estamos con el usuario HR), que imprima por consola una línea de separación, para lo cual me hace falta el carácter que forma parte de la línea, y cuantos caracteres la conforman. Le voy a llamar IMP\_LIN.

```
CREATE OR REPLACE PROCEDURE IMP_LIN(p_car char, p_cuantos pls_integer) AS  
BEGIN  
    dbms_output.put_line(RPAD(p_car, p_cuantos, p_car));  
END IMP_LIN;
```





# UF-7 PROGRAMACIÓN AVANZADA DE ACCESO A DATOS



Procedimiento PL/SQL terminado correctamente.  
Llamada a IMP\_LIN

- ❑ La ventaja que los parámetros sean sin precisión es que le puedo poner al parámetro todos los caracteres que quiera.

❑ Para probar que el procedimiento funciona creamos un bloque anónimo, con distintas llamadas:

set serveroutput on  
begin

imp\_lin('= ',60);  
imp\_lin('--FIN',40);  
imp\_lin('\* ',20);

end;  
/

=====





## UF-7 PROGRAMACIÓN AVANZADA DE ACCESO A DATOS

❑ Tipos de parámetros: Los parámetros, entre el nombre y el tipo de datos llevan un modificador, y pueden ser de tres tipos:

- IN
- OUT
- IN OUT

❑ IN: Parámetros de entrada.

- Suministra valores al procedimiento y el valor se trata como si fuera una constante. No se puede modificar en el valor dentro del procedimiento.
- Se puede especificar un valor por defecto por si no es suministrado al invocar el subprograma.
- Es la opción por defecto.
- Cuando invoco a un procedimiento, a un parámetro IN le puedo pasar una variable, o un literal del tipo correspondiente.







## UF-7 PROGRAMACIÓN AVANZADA DE ACCESO A DATOS

❑ Tipos de parámetros: Los parámetros, entre el nombre y el tipo de datos llevan un modificador, y pueden ser de tres tipos:

- IN
- OUT
- IN OUT

❑ OUT: Un parámetro OUT, en la invocación, sólo puedo pasar una variable, nunca un literal. Parámetros de salida. PL crea dentro del procedimiento una copia de esta variable pasada y la inicializa a NULL:

- Si el procedimiento termina bien, el contenido de esta variable, se copia en la variable del bloque que invocó al procedimiento
- Si el bloque termina levantando una excepción de tipo `RAISE_APPLICATION_ERROR(-20XXX,'mensaje de error')`, el contenido no se copia en la variable de origen.
- Si el bloque termina levantando una excepción de tipo `RAISE_APPLICATION_ERROR(-20XXX,'mensaje de error')`, y hemos especificado la opción `OUT NOCOPY`, el procedimiento trabaja directamente con la variable de origen, y cualquier cambio que hagamos dentro del procedimiento afecta a la variable pasada. Termine el programa bien o mal, el contenido queda cambiado.





## UF-7 PROGRAMACIÓN AVANZADA DE ACCESO A DATOS

❑ Tipos de parámetros: Los parámetros, entre el nombre y el tipo de datos llevan un modificador, y pueden ser de tres tipos:

- IN
- OUT
- IN OUT

❑ IN OUT: Parámetro de entrada y Salida. Se inicializa en el momento de invocar el subprograma y se trata como una variable dentro de él. Se comporta con las ventajas de una variable de entrada y con las de una variable de salida. Un ejemplo de una variable IN OUT es un cursor variable.





## UF-7 PROGRAMACIÓN AVANZADA DE ACCESO A DATOS

- ❑ Parámetros IN: En el ejemplo anterior todas las variables `p_car` y `p_cuantos` son de tipo IN; como es la opción por defecto, no es necesario especificarlo. Vamos a ver el uso de valores por defecto para parámetros IN, a través de un ejemplo.
- ❑ Al procedimiento `IMP_LIN` le vamos a hacer una mejora, si no nos pasan los caracteres, le ponemos por defecto el carácter '-' (guión normal), y si no me pasan cuantos caracteres pongo por defecto 40.

```
CREATE OR REPLACE PROCEDURE IMP_LIN(p_car in char default '-', p_cuantos IN pls_integer default 40) AS
BEGIN
    dbms_output.put_line(RPAD(p_car, p_cuantos, p_car));
END IMP_LIN;
```

- ❑ Probamos el procedimiento con un bloque anónimo. !!!!OJO!!!! : si paso un parámetro sólo, PL toma el literal como si fuera un(os) carácter(es), el decir el numero 20, lo toma como '20', y te imprime 40 caracteres(2020202020...).
- ❑ Si quiero pasar sólo el segundo parámetro, y que el carácter lo tome por defecto, fíjate en la llamada:  
`imp_lin(p_cuantos=> 10);`
- ❑ Si invoco sin parámetros, toma los dos por defecto.



begin

end;

/

--FIN--FIN--FIN--FIN--FIN--FIN

-----

-----



## Digital Workers



## UF-7 PROGRAMACIÓN AVANZADA DE ACCESO A DATOS

- ❑ Parámetros OUT: Vamos a ver cómo se trabaja a través de un ejemplo.
- ❑ Procedimiento llamado PAR\_IMPAR, nos pasan por IN un número, y una variable VARCHAR2 en donde le decimos si es PAR o IMPAR. pero si el número es mayor de 100, levantamos una excepción de tipo RAISE\_APPLICATION\_ERROR(-20100,'numero excede de 100'). Por motivos didácticos primero asignamos el literal y luego levantamos ala excepción.
- ❑ Primero especificamos la variable como OUT, y vemos el efecto en el bloque anónimo.

```
create or replace PROCEDURE PAR_IMPAR(P_NUMERO PLS_INTEGER,  
                                       P_LITERAL OUT VARCHAR2) AS  
  
BEGIN  
    IF MOD(P_NUMERO,2) = 0 THEN  
        P_LITERAL := ' PAR';  
    ELSE  
        P_LITERAL := ' IMPAR';  
    END IF;  
  
    IF P_NUMERO > 100 THEN  
        RAISE_APPLICATION_ERROR(-20100, 'EL NUMERO ES MAYOR DE 100');  
    END IF;  
  
END PAR_IMPAR;
```

Profesor Raúl Salgado Vilas





## UF-7 PROGRAMACIÓN AVANZADA DE ACCESO A DATOS

- ❑ Parámetros OUT: Vamos a ver cómo se trabaja a través de un ejemplo.
- ❑ Procedimiento llamado PAR\_IMPAR, nos pasan por IN un número, y una variable VARCHAR2 en donde le decimos si es PAR o IMPAR. pero si el número es mayor de 100, levantamos una excepción de tipo RAISE\_APPLICATION\_ERROR(-20100,'numero excede de 100'). Por motivos didácticos primero asignamos el literal y luego levantamos ala excepción.
- ❑ Primero especificamos la variable como OUT, y vemos el efecto en el bloque anónimo.

```
create or replace PROCEDURE PAR_IMPAR(P_NUMERO PLS_INTEGER,  
                                       P_LITERAL OUT VARCHAR2) AS  
  
BEGIN  
    IF MOD(P_NUMERO,2) = 0 THEN  
        P_LITERAL := ' PAR';  
    ELSE  
        P_LITERAL := ' IMPAR';  
    END IF;  
  
    IF P_NUMERO > 100 THEN  
        RAISE_APPLICATION_ERROR(-20100, 'EL NUMERO ES MAYOR DE 100');  
    END IF;  
  
END PAR_IMPAR;
```







## UF-7 PROGRAMACIÓN AVANZADA DE ACCESO A DATOS

- ☐ Para probar el Procedimiento PAR\_IMPAR, montamos un bucle desde el numero 98 al 101.
- ☐ En cada interacción ponemos V\_LITERAL := NULL, a nulo, para ver el efecto.
- ☐ Los números 98,99,100, se ejecutan bien.
- ☐ Al llegar a 101, el procedimiento levanta la excepción -20100, el bloque anónimo la captura en WHEN OTHERS THEN, saca el mensaje y al escribir V\_LITERAL, está a nulo, porque el procedimiento no ha volcado el literal 'IMPAR' almacenado en su variable P\_LITERAL.
- ☐ La función escalar NVL aplicado a una variable, si esta tiene el nulo activado, muestra el literal asociado, si tiene contenido, muestra el contenido.





## UF-7 PROGRAMACIÓN AVANZADA DE ACCESO A DATOS

```
SET SERVEROUTPUT ON
DECLARE
    V_LITERAL VARCHAR2(40);

BEGIN
    FOR NUMERO IN 98 .. 101 LOOP
        PAR_IMPAR(NUMERO, V_LITERAL);
        DBMS_OUTPUT.PUT_LINE('EL NUMERO ' || NUMERO || ' ES ' || v_LITERAL);
        V_LITERAL := NULL;
    END LOOP;
EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('ERROR EN BLOQUE PPAL : ' || SQLERRM);
        DBMS_OUTPUT.PUT_LINE('LITERAL : ' ||
            nvl(v_LITERAL, 'ESTA A NULO EL P_LITERAL NO SE HA COPIADO'));

END;
/
```

← Bloque anónimo probar PAR\_IMPAR

→

- ☐ EL NUMERO 98 ES PAR
- ☐ EL NUMERO 99 ES IMPAR
- ☐ EL NUMERO 100 ES PAR
- ☐ ERROR EN BLOQUE PPAL : ORA-20100: EL NUMERO ES MAYOR DE 100
- ☐ LITERAL : ESTA A NULO EL P\_LITERAL NO SE HA COPIADO
- ☐ Procedimiento PL/SQL terminado correctamente.





## UF-7 PROGRAMACIÓN AVANZADA DE ACCESO A DATOS

OUT NO COPY: Modificamos el Procedimiento PAR\_IMPAR, e incrustamos NOCOPY detrás de OUT en P\_LITERAL.

```
create or replace PROCEDURE PAR_IMPAR(P_NUMERO PLS_INTEGER,  
                                       P_LITERAL OUT NOCOPY VARCHAR2) AS
```

```
BEGIN
```

```
  IF MOD(P_NUMERO,2) = 0 THEN
```

```
    P_LITERAL := ' PAR';
```

```
  ELSE
```

```
    P_LITERAL := ' IMPAR';
```

```
  END IF;
```

```
IF P_NUMERO > 100 THEN
```

```
  RAISE_APPLICATION_ERROR(-20100, 'EL NUMERO ES MAYOR DE 100');
```

```
END IF;
```

```
END PAR_IMPAR;
```





## UF-7 PROGRAMACIÓN AVANZADA DE ACCESO A DATOS

❑ Ejecutamos el mismo bloque anónimo anterior (Bloque anónimo probar PAR\_IMPAR). El resultado es:

Procedimiento PL/SQL terminado correctamente.

EL NUMERO 98 ES PAR

EL NUMERO 99 ES IMPAR

EL NUMERO 100 ES PAR

ERROR EN BLOQUE PPAL : ORA-20100: EL NUMERO ES MAYOR  
DE 100

LITERAL : IMPAR

❑ Es decir con la opción OUT NOCOPY, el procedimiento trabaja directamente con la variable que le paso: V\_LITERAL, la modifica y pase lo que pase en el procedimiento, cuando nos cede el control, mi variable tiene el contenido (en este caso 'IMPAR').

Procedimiento PL/SQL terminado correctamente.

Date cuenta en la última línea, ha salido impar (era el numero 101).





# UF-7 PROGRAMACIÓN AVANZADA DE ACCESO A DATOS

```
.....  
FOR NUMERO IN 98 .. 101 LOOP  
.....  
END LOOP;  
  
EXCEPTION  
  WHEN OTHERS THEN  
    DBMS_OUTPUT.PUT_LINE('ERROR EN BLOQUE PPAL : ' ||  
SQLERRM);  
    DBMS_OUTPUT.PUT_LINE('LITERAL : ' ||  
nvl(v_LITERAL,'ESTA A NULO EL P_LITERAL NO SE HA  
COPIADO'));  
    DBMS_OUTPUT.PUT_LINE('EL NUMERO ' || NUMERO || '  
ES ' || v_LITERAL);  
  
END;  
/
```

❑ En el bloque anónimo que hemos empleado para este ejemplo (Bloque anónimo probar PAR\_IMPAR) de variables OUT, ni se te ocurra en el DBMS de la excepción referenciar la variable "numero" del FOR, porque recuerda que esa variable es "local" a FOR, y te daría error de interpretación.

Informe de error -  
ORA-06550: line 14, column 46:  
PLS-00201: identifier 'NUMERO' must be declared  
ORA-06550: line 14, column 9:  
PL/SQL: Statement ignored  
06550. 00000 - "line %s, column %s:\n%s"  
\*Cause: Usually a PL/SQL compilation error.  
\*Action:





## UF-7 PROGRAMACIÓN AVANZADA DE ACCESO A DATOS

- ❑ Cláusula PRAGMA AUTONOMOUS\_TRANSACTION: Un procedimiento autónomo permite realizar COMMIT o ROLLBACK de las sentencias SQL propias sin afectar a la transacción que lo haya llamado. Si en ejecución no se encuentra la sentencia COMMIT o ROLLBACK provocará una excepción; en el ejemplo siguiente, el COMMIT sólo afecta al INSERT del procedimiento llamado.

```
CREATE OR REPLACE PROCEDURE Llamador
IS
BEGIN
```

```
    UPDATE departments
```

```
    SET location_id=1700;
```

```
-- Se inicia una transacción
```

```
--Se invoca un procedimiento autónomo
```

```
    Procedimiento_autonomo;
```

```
    ROLLBACK;
```

```
-- Se deshará el UPDATE no el INSERT.
```

```
END Llamador;
```

```
/
```







## UF-7 PROGRAMACIÓN AVANZADA DE ACCESO A DATOS

```
CREATE OR REPLACE PROCEDURE Procedimiento_autonomo  
IS
```

```
    PRAGMA AUTONOMOUS_TRANSACTION;
```

```
BEGIN
```

```
    INSERT INTO locations(location_id,city)  
    VALUES (locations_seq.nextval,'Madrid');
```

```
    COMMIT; /* Sólo afectará al INSERT del procedimiento autónomo*/
```

```
END Procedimiento_autonomo;
```

```
/
```





## UF-7 PROGRAMACIÓN AVANZADA DE ACCESO A DATOS

- ❑ **Funciones Almacenadas:** Una función es un subprograma que calcula un valor. Las funciones difieren principalmente de los procedimientos en que siempre retornan un valor mediante la instrucción RETURN. La sintaxis para crear una función es:

```
CREATE [OR REPLACE]
    FUNCTION Nombre_Función
        [(declaracion_parámetro
        [, declaracion_parámetro]...)]
    RETURN Tipo_dato
        [AUTHID {DEFINER | CURRENT_USER}]
    {IS | AS}
        [DETERMINISTIC]
        [PRAGMA AUTONOMOUS_TRANSACTION;]
        [Declaraciones locales de tipos, variables, etc]
    BEGIN
        /*Sentencias ejecutables*/
        .....
    return literal/variable;
    [EXCEPTION
        --Excepciones definidas y las acciones de estas excepciones]
        .....
    return literal/variable;
END [Nombre_Función]
```



## UF-7 PROGRAMACIÓN AVANZADA DE ACCESO A DATOS

❑ Su estructura es exactamente igual a los procedimientos, con estas dos diferencias:

- En la cabecera de la función y antes del IS/AS, incorporan la cláusula RETURN tipo de dato. El tipo de datos es : tipos de Oracle, más pls\_inteber, más boolean, mas variables %Type, más registros%Rowtype.
- La cláusula DETERMINISTIC: ayuda al optimizador de Oracle a evitar llamadas redundantes. Si una función almacenada ha sido anteriormente invocada con los mismos parámetros el optimizador puede escoger devolver el mismo valor, sin volver a ejecutar la función.

❑ Toda Función finaliza con una sentencia RETURN, en donde se devuelve el contenido de una variable y/o un literal correspondiente. La Función más pequeña que podemos hacer sin errores de compilación es:

```
CREATE OR REPLACE FUNCTION NOMBRE_FUNCION RETURN VARCHAR2 AS
BEGIN
    RETURN NULL;
END NOMBRE_FUNCION;
```





## UF-7 PROGRAMACIÓN AVANZADA DE ACCESO A DATOS

- ❑ Definición y ejecución de funciones: Lo vemos con un ejemplo. Vamos a hacer una función denominada EXISTE\_DEP, que recibe como parámetro de entrada un código de departamento, y nos informa si ese departamento existe o no. Además, si el departamento existe, en un parámetro de tipo OUT nos vuelca un registro con todos los datos de este departamento.

```
CREATE OR REPLACE FUNCTION EXISTE_DEP(P_DEP DEPARTMENTS.DEPARTMENT_ID%TYPE,  
F_DEP OUT DEPARTMENTS%ROWTYPE) RETURN BOOLEAN AS  
BEGIN  
    SELECT *  
    INTO F_DEP  
    FROM DEPARTMENTS  
    WHERE DEPARTMENT_ID = P_DEP;  
    RETURN TRUE;  
EXCEPTION  
    WHEN NO_DATA_FOUND THEN  
        RETURN FALSE;  
END EXISTE_DEP;
```





## UF-7 PROGRAMACIÓN AVANZADA DE ACCESO A DATOS

❑ Y ahora vamos a hacer un bloque anónimo para probar. Haremos dos invocaciones:

- Departamento 30 : existe, y mostramos el nombre del departamento
- Departamento 430 : mostramos el mensaje no existe

```
set serveroutput on
declare
    f_dep departments%rowtype;
begin
    -- el primer parametro de la función lo cogemos de la consola
    if existe_dep(&dep, f_dep) then
        dbms_output.put_line('departamento : ' || f_dep.department_name);
    else
        dbms_output.put_line('departamento no existe');
    end if;
exception
    when others then
        dbms_output.put_line('error en bloque anónimo : ' || sqlerrm);

end;
/
```

- ❑ con 30
- ❑ departamento : Purchasing
- ❑ Procedimiento PL/SQL terminado correctamente.
- ❑ con 430
- ❑ departamento no existe
- ❑ Procedimiento PL/SQL terminado correctamente





## UF-7 PROGRAMACIÓN AVANZADA DE ACCESO A DATOS

- ☐ La sentencia RETURN (No la de la parte de la especificación de la función, donde especificamos el tipo de dato que se devuelve) finaliza la ejecución de la función y devuelve el control.
- ☐ Una función puede contener varias sentencias RETURN.
- ☐ La sentencia RETURN no tiene porque ser la última sentencia del subprograma o función.
- ☐ En las funciones, la sentencia debe devolver un valor. Este valor se evalúa en el momento de devolverlo, por lo que puede ser una expresión.
- ☐ Las funciones son invocadas como parte de una expresión y pueden ser invocadas desde múltiples sitios:
  - Un IF, para evaluar la condición.
  - Un bucle, para evaluar su finalización.
  - Para asignar lo que devuelve la función a una variable.







## UF-7 PROGRAMACIÓN AVANZADA DE ACCESO A DATOS

### ☐ Paquetes: NO ENTRA

- ☐ Un paquete sirve para encerrar la lógica del negocio, es decir las consultas y/o actualizaciones de mis tablas para evitar accesos indebidos.
- ☐ Objetivos: Que os suene pero no entra en el examen por lo que iremos rápido
  - Conocer las partes en que se divide la creación de un paquete.
  - Qué se puede escribir en la especificación del paquete.
  - El cuerpo del paquete subprogramas públicos y privados.
  - Desarrollo de un ejemplo de Paquete.
  - Trabajar con cursores variables en un paquete.
  - Ver las ventajas de trabajar con paquetes.



- ❏ Los paquetes se dividen en:



## UF-7 PROGRAMACIÓN AVANZADA DE ACCESO A DATOS

- ❑ Crear la especificación y el cuerpo: La forma genérica de crear una especificación de Paquete es:

```
CREATE [OR REPLACE] PACKAGE Nombre_paquete
    [AUTHID {CURRENT_USER | DEFINER}]
{IS | AS}
    [PRAGMA SERIALLY_REUSABLE;]
    [Definición_Tipo_Colección ...]
    [Definición_tipo_Registro ...]
    [Definición_Subtipos ...]
    [Declaración_Colección ...]
    [Declaración_constante ...]
    [Declaracion_Excepción ...]
    [Declaración_Objeto ...]
    [Declaración_Registro ...]
    [Declaración_Variable ...]
    [Especificación_Cursor ...]
    [Especificación_Función...]
    [Especificación_Procedimiento ...]
    [Especificación_Llamada ...]
    [PRAGMA RESTRICT_REFERENCES(Tipos_Comportamiento) ...]
END [Nombre_paquete];
/
```

- ❑ La especificación contiene la parte pública del paquete la cual es visible desde otras aplicaciones. Los procedimientos tienen que ser declarados al final de la zona de especificación, excepto las PRAGMAS que hacen referencia a alguna función.





## UF-7 PROGRAMACIÓN AVANZADA DE ACCESO A DATOS

❑ Una vez creada la cabecera del paquete, posteriormente se crearía el cuerpo:

```
CREATE [OR REPLACE] PACKAGE BODY Nombre_paquete {IS | AS}
    [PRAGMA SERIALLY_REUSABLE;]
    [Definición_Tipo_Colección ...]
    [Definición_tipo_Registro ...]
    [Definición_Subtipos ...]
    [Declaración_Colección ...]
    [Declaración_constante ...]
    [Declaracion_Excepción ...]
    [Declaración_Objeto ...]
    [Declaración_Registro ...]
    [Declaración_Variable ...]
    [Especificación_Cursor ...]
    [Especificación_Función...]
    [Especificación_Procedimiento ...]
    [Especificación_Llamada ...]
[BEGIN
    Sentencias procedurales
[EXCEPTION
    Tratamiento de excepciones]]
END [Nombre_paquete];
/
```



- ❑ Los procedimientos y/o funciones de un paquete se pueden sobrecargar, es decir, crear el mismo nombre, pero admitiendo distintos tipos de parámetros y/o distinto número de parámetros.
- ❑ El cuerpo del paquete contiene la implementación de cada cursor y subprograma declarado en la parte de las especificaciones. Todos los subprogramas que estén declarados en las especificaciones serán públicos, el resto serán privados y no podrán ser accedidos fuera del paquete.
- ❑ Para concordar los procedimientos declarados en la zona de especificaciones y el cuerpo se hace una comparación carácter a carácter. La única excepción es el espacio en blanco. En caso de que no coincidiera Oracle levantaría una excepción.
- ❑ El begin de un paquete es opcional y éste se ejecuta una sola vez por sesión.
- ❑ La zona de excepciones del paquete trata las excepciones originadas sólo por instrucciones del begin del paquete.



## UF-7 PROGRAMACIÓN AVANZADA DE ACCESO A DATOS

- ❑ La invocación a un tipo de datos avanzado, subprograma de un paquete desde otro paquete, procedimiento o función se puede realizar, siempre y cuando estos subprogramas sean públicos:

```
DECLARE
  variable  nommbre_paquete.tipo_avanzado;

BEGIN
  Nombre_paquete.nombre_procedimiento(parametros);

  Variable:= Nombre_paquete.nombre_funcion(parametros);

  IF Nombre_paquete.nombre_funcion(parámetros) < 10 THEN
    ...
  END IF;
```





## UF-7 PROGRAMACIÓN AVANZADA DE ACCESO A DATOS

❑ Ejemplo de paquete: Para la gestión de las tablas del esquema HR, vamos a crear un paquete llamado: PKT\_GESTION\_EMPL. Y vamos a crear los siguientes Subprogramas y tipos necesarios:

❑ Procedimientos:

- PROCEDURE IMP\_EMPLEADOS(P\_DEP EMPLOYEES.DEPARTMENT\_ID%TYPE, P\_CUANTOS OUT PLS\_INTEGER);
- PROCEDURE CUR\_ESPECIFICO( P\_CURSOR IN OUT CUR\_2V2, OPCION PLS\_INTEGER);

❑ Funciones:

- FUNCTION EXISTE\_DEP(P\_DEP DEPARTMENTS.DEPARTMENT\_ID%TYPE, P\_FILA OUT DEPARTMENTS%ROWTYPE) RETURN BOOLEAN;
- FUNCTION EXISTE\_DEP(P\_DEP DEPARTMENTS.DEPARTMENT\_ID%TYPE) RETURN BOOLEAN;

❑ Y los siguientes Tipos avanzados:

- TYPE REG\_2V2 IS RECORD (CAMPO\_1 VARCHAR2(4000), CAMPO\_2 VARCHAR2(4000));
- TYPE CUR\_2V2 IS REF CURSOR RETURN REG\_2V2;





## UF-7 PROGRAMACIÓN AVANZADA DE ACCESO A DATOS

```
CREATE OR REPLACE PACKAGE PKT_GESTION_EMPL AS
```

```
TYPE REG_2V2 IS RECORD (CAMPO_1 VARCHAR2(4000), CAMPO_2 VARCHAR2(4000));
```

```
TYPE CUR_2V2 IS REF CURSOR RETURN REG_2V2;
```

```
PROCEDURE IMP_EMPLEADOS(P_DEP EMPLOYEES.DEPARTMENT_ID%TYPE, P_CUANTOS OUT PLS_INTEGER);
```

```
FUNCTION EXISTE_DEP(P_DEP DEPARTMENTS.DEPARTMENT_ID%TYPE, P_FILA OUT DEPARTMENTS%ROWTYPE) RETURN  
BOOLEAN;
```

```
FUNCTION EXISTE_DEP(P_DEP DEPARTMENTS.DEPARTMENT_ID%TYPE) RETURN BOOLEAN;
```

```
PROCEDURE CUR_ESPECIFICO( P_CURSOR IN OUT CUR_2V2, OPCION PLS_INTEGER);
```

```
END PKT_GESTION_EMPL;
```

```
/
```

```
Package created.
```

```
PKT_GESTION_EMPL
```







## UF-7 PROGRAMACIÓN AVANZADA DE ACCESO A DATOS

- ❑ Una vez compilada la especificación del paquete, procedemos a crear el body con la implementación de cada procedimiento y cada función, si lo hacemos con el sqldeveloper, este es el cuerpo que genera:

```
CREATE OR REPLACE PACKAGE BODY PKT_GESTION_EMPL AS
```

```
PROCEDURE IMP_EMPLEADOS(P_DEP EMPLOYEES.DEPARTMENT_ID%TYPE, P_CUANTOS OUT PLS_INTEGER) AS  
BEGIN
```

```
-- TAREA: Se necesita implantación para PROCEDURE PKT_GESTION_EMPL.IMP_EMPLEADOS  
NULL;  
END IMP_EMPLEADOS;
```

```
FUNCTION EXISTE_DEP(P_DEP DEPARTMENTS.DEPARTMENT_ID%TYPE, P_FILA OUT DEPARTMENTS%ROWTYPE) RETURN  
BOOLEAN AS  
BEGIN
```

```
-- TAREA: Se necesita implantación para FUNCTION PKT_GESTION_EMPL.EXISTE_DEP  
RETURN NULL;  
END EXISTE_DEP;
```





## UF-7 PROGRAMACIÓN AVANZADA DE ACCESO A DATOS

```
FUNCTION EXISTE_DEP(P_DEP DEPARTMENTS.DEPARTMENT_ID%TYPE) RETURN BOOLEAN AS  
BEGIN
```

```
-- TAREA: Se necesita implantación para FUNCTION PKT_GESTION_EMPL.EXISTE_DEP  
RETURN NULL;  
END EXISTE_DEP;
```

```
PROCEDURE CUR_ESPECIFICO( P_CURSOR IN OUT CUR_2V2, OPCION PLS_INTEGER) AS  
BEGIN
```

```
-- TAREA: Se necesita implantación para PROCEDURE PKT_GESTION_EMPL.CUR_ESPECIFICO  
NULL;  
END CUR_ESPECIFICO;
```

```
END PKT_GESTION_EMPL;
```





## UF-7 PROGRAMACIÓN AVANZADA DE ACCESO A DATOS

- ☐ Es una gran ayuda, porque nos crea un cuerpo, totalmente compilable, sin errores y a partir de ahí podemos ir dando forma a cada uno de los subprogramas.
- ☐ Veamos el procedimiento empleados que recibe por parámetro un código de departamento, e imprime los empleados pertenecientes a este departamento. En una variable OUT vuelca el número de empleados se han procesado.
- ☐ El procedimiento no verifica si el departamento existe o no existe.





## UF-7 PROGRAMACIÓN AVANZADA DE ACCESO A DATOS

CREATE OR REPLACE PACKAGE BODY PKT\_GESTION\_EMPL AS

```
PROCEDURE IMP_EMPLEADOS(P_DEP EMPLOYEES.DEPARTMENT_ID%TYPE, P_CUANTOS OUT PLS_INTEGER) AS
CURSOR CUR_EMPL IS
  SELECT LAST_NAME, SALARY, HIRE_DATE
  FROM EMPLOYEES
  WHERE DEPARTMENT_ID = P_DEP;
```

BEGIN

```
  P_CUANTOS := 0;
  FOR REG_EMPL IN CUR_EMPL LOOP
    DBMS_OUTPUT.PUT('APELLIDO : ' || RPAD(REG_EMPL.LAST_NAME,15));
    DBMS_OUTPUT.PUT('SALARIO : ' || LPAD(REG_EMPL.SALARY,8));
    DBMS_OUTPUT.PUT_LINE(' FECHA : ' || TO_CHAR(REG_EMPL.HIRE_DATE,'DD-MM-YYYY'));
  --  P_CUANTOS := CUR_EMPL%ROWCOUNT;
    P_CUANTOS := P_CUANTOS + 1;
```

END LOOP;

END IMP\_EMPLEADOS;





## UF-7 PROGRAMACIÓN AVANZADA DE ACCESO A DATOS

```
FUNCTION EXISTE_DEP(P_DEP DEPARTMENTS.DEPARTMENT_ID%TYPE, P_FILA OUT DEPARTMENTS%ROWTYPE) RETURN  
BOOLEAN AS
```

```
BEGIN  
  -- TAREA: Se necesita implantación para FUNCTION PKT_GESTION_EMPL.EXISTE_DEP  
  RETURN NULL;  
END EXISTE_DEP;
```

```
FUNCTION EXISTE_DEP(P_DEP DEPARTMENTS.DEPARTMENT_ID%TYPE) RETURN BOOLEAN AS  
BEGIN
```

```
  -- TAREA: Se necesita implantación para FUNCTION PKT_GESTION_EMPL.EXISTE_DEP  
  RETURN NULL;  
END EXISTE_DEP;
```

```
FUNCTION GET_EMPLEADOS(P_DEP DEPARTMENTS.DEPARTMENT_ID%TYPE) RETURN TAB_EMPL AS  
BEGIN
```

```
  -- TAREA: Se necesita implantación para FUNCTION PKT_GESTION_EMPL.GET_EMPLEADOS  
  RETURN NULL;  
END GET_EMPLEADOS;
```





## UF-7 PROGRAMACIÓN AVANZADA DE ACCESO A DATOS

```
PROCEDURE CUR_ESPECIFICO( P_CURSOR IN OUT CUR_2V2, OPCION PLS_INTEGER) AS
BEGIN
    -- TAREA: Se necesita implantación para PROCEDURE PKT_GESTION_EMPL.CUR_ESPECIFICO
    NULL;
END CUR_ESPECIFICO;

PROCEDURE INSERT_DEPAR(P_FILA_DEP DEPARTMENTS%ROWTYPE) AS
BEGIN
    -- TAREA: Se necesita implantación para PROCEDURE PKT_GESTION_EMPL.INSERT_DEPAR
    NULL;
END INSERT_DEPAR;

END PKT_GESTION_EMPL;
```





## UF-7 PROGRAMACIÓN AVANZADA DE ACCESO A DATOS

❑ Y hacemos un bloque anónimo para probar el procedimiento:

```
SET SERVEROUTPUT ON
```

```
DECLARE
```

```
  V_CUANTOS PLS_INTEGER;
```

```
  V_DEP EMPLOYEES.DEPARTMENT_ID%TYPE;
```

```
  F_DEP DEPARTMENTS%ROWTYPE;
```

```
BEGIN
```

```
  V_DEP := &DEP;
```

```
  PKT_GESTION_EMPL.IMP_EMPLEADOS(V_DEP,V_CUANTOS);
```

```
  IF V_CUANTOS = 0 THEN
```

```
    DBMS_OUTPUT.PUT_LINE('DEPARTAMENTO SIN EMPLEADOS');
```

```
  ELSE
```

```
    DBMS_OUTPUT.PUT_LINE('EMPLEADOS LEIDOS : ' || V_CUANTOS);
```

```
  END IF;
```

```
EXCEPTION
```

```
  WHEN OTHERS THEN
```

```
    DBMS_OUTPUT.PUT_LINE('ERROR GENERAL TEST_imp_empls : ' || SQLERRM);
```

```
END;
```





## UF-7 PROGRAMACIÓN AVANZADA DE ACCESO A DATOS

Enter value for dep: 30

old 6: V\_DEP := &DEP;

new 6: V\_DEP := 30;

APELLIDO : Raphaely      SALARIO : 11000 FECHA : 07-12-2002

APELLIDO : Khoo          SALARIO : 3100 FECHA : 18-05-2003

APELLIDO : Baida        SALARIO : 2900 FECHA : 24-12-2005

APELLIDO : Tobias        SALARIO : 2800 FECHA : 24-07-2005

APELLIDO : Himuro       SALARIO : 2600 FECHA : 15-11-2006

APELLIDO : Colmenares   SALARIO : 2500 FECHA : 10-08-2007

EMPLEADOS LEIDOS : 6







## UF-7 PROGRAMACIÓN AVANZADA DE ACCESO A DATOS

- ☐ `FUNCTION EXISTE_DEP(P_DEP DEPARTMENTS.DEPARTMENT_ID%TYPE, P_FILA OUT DEPARTMENTS%ROWTYPE)`  
`RETURN BOOLEAN AS`
- ☐ Función que pide un código de departamento y si existe en la tabla devuelve true y además vuelca la fila obtenida en un parámetro de departments%rowtype.
- ☐ Si no existe el departamento devuelve FALSE.





## UF-7 PROGRAMACIÓN AVANZADA DE ACCESO A DATOS

CREATE OR REPLACE

PACKAGE BODY PKT\_GESTION\_EMPL AS

PROCEDURE IMP\_EMPLEADOS(P\_DEP EMPLOYEES.DEPARTMENT\_ID%TYPE, P\_CUANTOS OUT PLS\_INTEGER) AS

CURSOR CUR\_EMPL IS

SELECT LAST\_NAME, SALARY, HIRE\_DATE

FROM EMPLOYEES

WHERE DEPARTMENT\_ID = P\_DEP;

BEGIN

P\_CUANTOS := 0;

FOR REG\_EMPL IN CUR\_EMPL LOOP

DBMS\_OUTPUT.PUT('APELLIDO : ' || RPAD(REG\_EMPL.LAST\_NAME,15));

DBMS\_OUTPUT.PUT('SALARIO : ' || LPAD(REG\_EMPL.SALARY,8));

DBMS\_OUTPUT.PUT\_LINE(' FECHA : ' || TO\_CHAR(REG\_EMPL.HIRE\_DATE,'DD-MM-YYYY'));

-- P\_CUANTOS := CUR\_EMPL%ROWCOUNT;

P\_CUANTOS := P\_CUANTOS + 1;

END LOOP;

END IMP\_EMPLEADOS;





## UF-7 PROGRAMACIÓN AVANZADA DE ACCESO A DATOS

```
FUNCTION EXISTE_DEP(P_DEP DEPARTMENTS.DEPARTMENT_ID%TYPE, P_FILA OUT DEPARTMENTS%ROWTYPE) RETURN  
BOOLEAN AS  
BEGIN  
    SELECT *  
    INTO P_FILA  
    FROM DEPARTMENTS  
    WHERE DEPARTMENT_ID = P_DEP;  
    RETURN TRUE;  
EXCEPTION  
    WHEN NO_DATA_FOUND THEN  
        RETURN FALSE;  
END EXISTE_DEP;
```





## UF-7 PROGRAMACIÓN AVANZADA DE ACCESO A DATOS

```
FUNCTION EXISTE_DEP(P_DEP DEPARTMENTS.DEPARTMENT_ID%TYPE) RETURN BOOLEAN AS
BEGIN
    RETURN NULL;
END EXISTE_DEP;
```

```
PROCEDURE CUR_ESPECIFICO( P_CURSOR IN OUT CUR_2V2, OPCION PLS_INTEGER) AS
BEGIN
    -- TAREA: Se necesita implantación para PROCEDURE PKT_GESTION_EMPL.CUR_ESPECIFICO
    NULL;
END CUR_ESPECIFICO;

END PKT_GESTION_EMPL;
```





## UF-7 PROGRAMACIÓN AVANZADA DE ACCESO A DATOS

Y el correspondiente bloque anónimo para probar la función EXISTE\_DEP:

```
SET SERVEROUTPUT ON
DECLARE
    FILA_DEP DEPARTMENTS%ROWTYPE;
BEGIN
    IF PKT_GESTION_EMPL.EXISTE_DEP(30, FILA_DEP) THEN
        DBMS_OUTPUT.PUT_LINE('DEPARTAMENTO : ' || FILA_DEP.DEPARTMENT_NAME);
    ELSE
        DBMS_OUTPUT.PUT_LINE('DEPARTAMENTO NO ENCONTRADO');
    END IF;

    EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('ERROR GENERAL TEST_FUN_EXISTE_DEP : ' || SQLERRM);
END;
/
DEPARTAMENTO : Purchasing
PL/SQL procedure successfully completed.
```





## UF-7 PROGRAMACIÓN AVANZADA DE ACCESO A DATOS

- ❑ **Sobrecargar la Función EXISTE\_DEP:** Definimos el mismo nombre de función EXISTE\_DEP, pero solo admitimos como parámetro de entrada el código de departamento, si existe devolvemos TRUE y si no existe devolvemos FALSE.
- ❑ Para probar vamos a usar el mismo de antes, para saber el mensaje de qué función es, en los DBMS especificamos fun 2 param, fun 1 param.





## UF-7 PROGRAMACIÓN AVANZADA DE ACCESO A DATOS

CREATE OR REPLACE

PACKAGE BODY PKT\_GESTION\_EMPL AS

PROCEDURE IMP\_EMPLEADOS(P\_DEP EMPLOYEES.DEPARTMENT\_ID%TYPE, P\_CUANTOS OUT PLS\_INTEGER) AS

CURSOR CUR\_EMPL IS

SELECT LAST\_NAME, SALARY, HIRE\_DATE

FROM EMPLOYEES

WHERE DEPARTMENT\_ID = P\_DEP;

BEGIN

P\_CUANTOS := 0;

FOR REG\_EMPL IN CUR\_EMPL LOOP

DBMS\_OUTPUT.PUT('APELLIDO : ' || RPAD(REG\_EMPL.LAST\_NAME,15));

DBMS\_OUTPUT.PUT('SALARIO : ' || LPAD(REG\_EMPL.SALARY,8));

DBMS\_OUTPUT.PUT\_LINE(' FECHA : ' || TO\_CHAR(REG\_EMPL.HIRE\_DATE,'DD-MM-YYYY'));

-- P\_CUANTOS := CUR\_EMPL%ROWCOUNT;

P\_CUANTOS := P\_CUANTOS + 1;

END LOOP;

END IMP\_EMPLEADOS;





## UF-7 PROGRAMACIÓN AVANZADA DE ACCESO A DATOS

```
FUNCTION EXISTE_DEP(P_DEP DEPARTMENTS.DEPARTMENT_ID%TYPE, P_FILA OUT DEPARTMENTS%ROWTYPE) RETURN  
BOOLEAN AS  
BEGIN  
    SELECT *  
    INTO P_FILA  
    FROM DEPARTMENTS  
    WHERE DEPARTMENT_ID = P_DEP;  
    RETURN TRUE;  
EXCEPTION  
    WHEN NO_DATA_FOUND THEN  
        RETURN FALSE;  
END EXISTE_DEP;
```







## UF-7 PROGRAMACIÓN AVANZADA DE ACCESO A DATOS

FUNCTION EXISTE\_DEP(P\_DEP DEPARTMENTS.DEPARTMENT\_ID%TYPE) RETURN BOOLEAN AS

  v\_cuantos pls\_integer := 0;

BEGIN

  SELECT COUNT(\*)

  INTO V\_CUANTOS

  FROM EMPLOYEES

  WHERE DEPARTMENT\_ID = P\_DEP;

  IF V\_CUANTOS = 0 THEN

    DBMS\_OUTPUT.PUT\_LINE('EXISTE\_DEP 1 PARAM');

    RETURN FALSE;

  ELSE

    DBMS\_OUTPUT.PUT\_LINE('EXISTE\_DEP 1 PARAM');

    RETURN TRUE;

  END IF;

END EXISTE\_DEP;





## UF-7 PROGRAMACIÓN AVANZADA DE ACCESO A DATOS

```
PROCEDURE CUR_ESPECIFICO( P_CURSOR IN OUT CUR_2V2, OPCION PLS_INTEGER) AS
BEGIN
  -- TAREA: Se necesita implantación para PROCEDURE PKT_GESTION_EMPL.CUR_ESPECIFICO
  NULL;
END CUR_ESPECIFICO;

END PKT_GESTION_EMPL;
```





## UF-7 PROGRAMACIÓN AVANZADA DE ACCESO A DATOS

- ☐ Trabajando con cursores variables: Vamos a generar el procedimiento llamado:
- ☐ PROCEDURE CUR\_ESPECIFICO( P\_CURSOR IN OUT CUR\_2V2, OPCION PLS\_INTEGER) AS
- ☐ Este procedimiento trabaja con un cursor variable IN OUT, definido como TYPE en la especificación del paquete, que devuelve consultas formadas por 2 variables de tipo VARCHAR2, definidas en un registro del paquete.
- ☐ El procedimiento recibe además de una variable REF CURSOR del paquete, un número natural de forma que si recibimos en la OPCION los valores:
  - Seleccionamos el last\_name y el first\_name de employees para los empleados del departamento 30.
  - Seleccionamos el job\_id y job\_title de la tabla jobs, de aquellos cuyo job\_id comience con la letra 'S'.
  - Cualquier otro número levantamos la excepción -20100, 'OPCION INCORRECTA, SOLO 1 o 2'. El cursor se abre, y se pasa por IN OUT al proceso que lo solicita, y la lectura y cierre lo hace el Test.





## UF-7 PROGRAMACIÓN AVANZADA DE ACCESO A DATOS

- ❑ Este es el tratamiento típico de cursores variables. La consulta se hace en el Bloque anónimo, y el tratamiento en el bloque o programa que solicita la información (p.ej, Java):

```
CREATE OR REPLACE PACKAGE BODY PKT_GESTION_EMPL AS
```

```
  PROCEDURE IMP_EMPLEADOS(P_DEP EMPLOYEES.DEPARTMENT_ID%TYPE, P_CUANTOS OUT PLS_INTEGER) AS
```

```
  CURSOR CUR_EMPL IS
```

```
    SELECT LAST_NAME, SALARY, HIRE_DATE
```

```
    FROM EMPLOYEES
```

```
    WHERE DEPARTMENT_ID = P_DEP;
```

```
BEGIN
```

```
  P_CUANTOS := 0;
```

```
  FOR REG_EMPL IN CUR_EMPL LOOP
```

```
    DBMS_OUTPUT.PUT('APELLIDO : ' || RPAD(REG_EMPL.LAST_NAME,15));
```

```
    DBMS_OUTPUT.PUT('SALARIO : ' || LPAD(REG_EMPL.SALARY,8));
```

```
    DBMS_OUTPUT.PUT_LINE(' FECHA : ' || TO_CHAR(REG_EMPL.HIRE_DATE,'DD-MM-YYYY'));
```

```
--  P_CUANTOS := CUR_EMPL%ROWCOUNT;
```

```
  P_CUANTOS := P_CUANTOS + 1;
```

```
  END LOOP;
```

```
END IMP_EMPLEADOS;
```





## UF-7 PROGRAMACIÓN AVANZADA DE ACCESO A DATOS

```
FUNCTION EXISTE_DEP(P_DEP DEPARTMENTS.DEPARTMENT_ID%TYPE, P_FILA OUT DEPARTMENTS%ROWTYPE) RETURN  
BOOLEAN AS  
BEGIN  
    SELECT *  
    INTO P_FILA  
    FROM DEPARTMENTS  
    WHERE DEPARTMENT_ID = P_DEP;  
    RETURN TRUE;  
EXCEPTION  
    WHEN NO_DATA_FOUND THEN  
        RETURN FALSE;  
END EXISTE_DEP;
```





## UF-7 PROGRAMACIÓN AVANZADA DE ACCESO A DATOS

```
FUNCTION EXISTE_DEP(P_DEP DEPARTMENTS.DEPARTMENT_ID%TYPE) RETURN BOOLEAN AS
    v_cuantos pls_integer := 0;
BEGIN
    SELECT COUNT(*)
    INTO V_CUANTOS
    FROM EMPLOYEES
    WHERE DEPARTMENT_ID = P_DEP;
    IF V_CUANTOS = 0 THEN
        DBMS_OUTPUT.PUT_LINE('EXISTE_DEP 1 PARAM');
        RETURN FALSE;
    ELSE
        DBMS_OUTPUT.PUT_LINE('EXISTE_DEP 1 PARAM');
        RETURN TRUE;
    END IF;
END EXISTE_DEP;
```





## UF-7 PROGRAMACIÓN AVANZADA DE ACCESO A DATOS

```
PROCEDURE CUR_ESPECIFICO( P_CURSOR IN OUT CUR_2V2, OPCION PLS_INTEGER) AS

BEGIN
CASE OPCION
WHEN 1 THEN
    OPEN P_CURSOR FOR SELECT LAST_NAME, FIRST_NAME FROM EMPLOYEES WHERE DEPARTMENT_ID = 30;
WHEN 2 THEN
    OPEN P_CURSOR FOR SELECT JOB_ID, JOB_TITLE FROM JOBS WHERE JOB_TITLE LIKE 'S%';
ELSE
    RAISE_APPLICATION_ERROR(-20100, 'OPCION INCORRECTA, SOLO 1 o 2');
END CASE;
END CUR_ESPECIFICO;

END PKT_GESTION_EMPL;
```





## UF-7 PROGRAMACIÓN AVANZADA DE ACCESO A DATOS

- ❑ Este es el bloque anónimo para probar el procedimiento, con las tres salidas correspondientes a las opciones:
  - 4 (excepción opción incorrecta).
  - 2 (lectura de trabajos que empiecen por 'S').
  - 1 (lectura de empleados del departamento 30).
  - Fíjate que las dos variables declaradas son de los tipos definidos en el paquete, tanto el cursor como el registro. Es una forma cómoda de trabajar, no me tengo que inventar tipos nuevos, y el paquete trabaja con sus tipos de forma óptima.
  
- ❑ Al llamar al procedimiento
  
- ❑ `PKT_GESTION_EMPL.CUR_ESPECIFICO(MI_CURSOR, 3);` El procedimiento me vuelca, en la variable `mi_cursor`, la dirección de memoria del fichero que ha generado. Y ahora con un bucle `loop end loop;` trato la información.







## UF-7 PROGRAMACIÓN AVANZADA DE ACCESO A DATOS

SET SERVEROUTPUT ON

DECLARE

MI\_CURSOR PKT\_GESTION\_EMPL.CUR\_2V2;

MI\_REG PKT\_GESTION\_EMPL.REG\_2V2;

BEGIN

PKT\_GESTION\_EMPL.CUR\_ESPECIFICO(MI\_CURSOR, 3);

LOOP

FETCH MI\_CURSOR INTO MI\_REG;

EXIT WHEN MI\_CURSOR%NOTFOUND;

DBMS\_OUTPUT.PUT\_LINE('APELLIDO O JOB\_ID : ' || MI\_REG.CAMPO\_1);

END LOOP;

IF MI\_CURSOR%ROWCOUNT = 0 THEN

DBMS\_OUTPUT.PUT\_LINE('CURSOR VACÍO');

END IF;

CLOSE MI\_CURSOR;





## UF-7 PROGRAMACIÓN AVANZADA DE ACCESO A DATOS

```
EXCEPTION
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('ERROR GENERAL TEST_CUR_ESPECIFICO : ' || SQLERRM);
END;
/
```

Enter value for opcion: 4

ERROR GENERAL TEST\_CUR\_ESPECIFICO : ORA-20100: OPCION INCORRECTA, SOLO 1 o 2

PL/SQL procedure successfully completed.

Enter value for opcion: 2

APELLIDO O JOB\_ID : SA\_MAN

APELLIDO O JOB\_ID : SA\_REP

APELLIDO O JOB\_ID : ST\_MAN

APELLIDO O JOB\_ID : ST\_CLERK

APELLIDO O JOB\_ID : SH\_CLERK



PL/SQL procedure successfully completed.

Enter value for opcion: 1

APELLIDO O JOB\_ID : Raphaely

APELLIDO O JOB\_ID : Khoo

APELLIDO O JOB\_ID : Baida

APELLIDO O JOB\_ID : Tobias

APELLIDO O JOB\_ID : Himuro

APELLIDO O JOB\_ID : Colmenares

PL/SQL procedure successfully completed.



## UF-7 PROGRAMACIÓN AVANZADA DE ACCESO A DATOS

## ❑ Ventajas de los Paquetes PL/SQL:

- **Modularidad:** Los paquetes pueden encapsular lógicamente tipos de datos y subprogramas en un módulo PL/SQL con nombre. Cada paquete es fácil de entender, y las interfaces con los paquetes son simples, claras y bien definidas, esto facilita el desarrollo de la aplicación.
- **Facilidad en el Diseño de la Aplicación:** Cuando diseñamos una aplicación, todo lo que inicialmente se necesita es la información de la interfaz en la especificación del paquete. No se necesita definir completamente el cuerpo del paquete hasta que no se complete la definición de la aplicación.
- **Ocultamiento de la Información:** En los paquetes, se pueden especificar tipos de datos y subprogramas para que sean públicos (visibles y accesibles) o privados (invisibles e inaccesibles). Por ejemplo, si tenemos un paquete que contiene cuatro subprogramas, tres públicos y uno privado. El paquete oculta la especificación del subprograma privado y solo implementa su código en el cuerpo del paquete.
- **Funcionalidad Agregada:** Las variables son persistentes para la sesión. Es decir, su valor se mantiene para toda la sesión del usuario que ejecuta ese paquete. Si otro usuario invoca el paquete, las variables contendrán el valor que inicialice el paquete, no los valores modificados por otro usuario.



## UF-7 PROGRAMACIÓN AVANZADA DE ACCESO A DATOS

- Mejora Ejecución: En la parte declarativa del cuerpo del paquete, opcionalmente, se puede inicializar las variables globales del cuerpo del paquete. Esta inicialización se ejecutará la primera vez que el paquete se coloque en memoria, es decir, la primera vez que un procedimiento del paquete sea invocado
- Sobrecarga de subprogramas en paquetes: PL/SQL permite dos o más subprogramas con el mismo nombre dentro del mismo paquete. Esta opción es usada cuando se necesita un subprograma igual que acepte parámetros que tienen diferentes tipos de datos.





## UF-7 TRIGGERS

- ☐ Los **disparadores** en base de datos permiten poder realizar acciones programáticas en el caso de que se produzcan determinados eventos en una base de datos.
- ☐ Es una herramienta que permite de alguna forma controlar todo el funcionamiento de la base de datos para que sea estable, bien construida y mantenida. Hay que tener en cuenta que muchos usuarios y programas accederán a los datos inmersos en la base de datos y los TRIGGER nos permiten controlarla de alguna forma.

### Objetivos

- ☐ Conocer cómo se crean los disparadores.
- ☐ Conocer los usos de los disparadores en bases de datos.
- ☐ Conocer y manejar los Disparadores DML.
- ☐ Usar los nombres de correlación y pseudorecords.
- ☐ Usar los Disparadores de Sistema.
- ☐ Manejar los errores y las excepciones en los disparadores.
- ☐ Conocer las buenas prácticas para usar disparadores.
- ☐ Manejar el orden de ejecución de disparadores.
- ☐ Conocer la habilitación y deshabilitación de disparadores.





## UF-7 TRIGGERS

- ❑ Un **disparador** o **TRIGGER** es un elemento de código que se parece a un procedimiento almacenado, es decir es un elemento de la estructura que se almacena en la base de datos que se ha creado al efecto.
- ❑ Por lo tanto un disparador o TRIGGER es un código específico que puede invocarse o lanzarse más de una vez. Un disparador se puede activar y desactivar de forma que pueda ser llamado o no por un elemento programático.
- ❑ Hay tres tipos de disparadores de bases 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 de filas).
  - **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, etcétera) o una instrucción de definición de datos (creación, modificación o eliminación de una tabla u otro objeto).





## UF-7 TRIGGERS

❑ **Usos de los disparadores.** Se pueden utilizar para:

- Implementar restricciones complejas de seguridad o integridad.
- Posibilitar la realización de operaciones de manipulación sobre vistas.
- Prevenir transacciones erróneas.
- Implementar reglas administrativas complejas.
- Generar automáticamente valores derivados.
- Auditar las actualizaciones e, incluso, enviar alertas.
- Gestionar réplicas remotas de la tabla.





## UF-7 TRIGGERS

- ❑ **Los disparadores de tablas son disparadores asociados a una determinada tabla de la base de datos:** Se disparan cuando se produce un determinado suceso o evento de manipulación que afecta a la tabla (inserción, borrado o modificación de filas).
- ❑ El siguiente ejemplo crea el trigger audit\_subida\_salario, que se disparará después de cada modificación de la columna salario de la tabla employees de la base de datos HR de ejemplo de Oracle. En la tabla auditaremples tendremos auditadas todas las subidas de salarios realizadas:

```
CREATE OR REPLACE TRIGGER audit_subida_salario
  AFTER UPDATE OF salary ON employees
  FOR EACH ROW
  BEGIN
    INSERT INTO auditaremples
    VALUES ('SUBIDA SALARIO EMPLEADO ' || :old.employee_id );
  END;
/
```

- ❑ Este disparador requiere de la tabla auditaremples, que habrá sido creada:

```
CREATE TABLE auditaremples (log VARCHAR2(200));
```

- ❑ De esta manera si ejecutamos la sentencia:

```
UPDATE employees SET salary=18000 WHERE employee_id=100;
```







## UF-7 TRIGGERS

- ❑ **Los disparadores de tablas son disparadores asociados a una determinada tabla de la base de datos:** Se disparan cuando se produce un determinado suceso o evento de manipulación que afecta a la tabla (inserción, borrado o modificación de filas).
- ❑ El siguiente ejemplo crea el trigger audit\_subida\_salario, que se disparará después de cada modificación de la columna salario de la tabla employees de la base de datos HR de ejemplo de Oracle. En la tabla auditaremples tendremos auditadas todas las subidas de salarios realizadas:

```
CREATE OR REPLACE TRIGGER audit_subida_salario
  AFTER UPDATE OF salary ON employees
  FOR EACH ROW
  BEGIN
    INSERT INTO auditaremples
    VALUES ('SUBIDA SALARIO EMPLEADO ' || :old.employee_id );
  END;
/
```

- ❑ Este disparador requiere de la tabla auditaremples, que habrá sido creada:

```
CREATE TABLE auditaremples (log VARCHAR2(200));
```

- ❑ De esta manera si ejecutamos la sentencia:

```
UPDATE employees SET salary=18000 WHERE employee_id=100;
```





## UF-7 TRIGGERS

- ❑ Crearemos registros del siguiente tipo en la tabla auditaremples:

SUBIDA SALARIO EMPLEADO 100

- ❑ El formato es el siguiente:

```
CREATE [OR REPLACE] TRIGGER nombretigger
  {BEFORE | AFTER}
  {DELETE | INSERT | UPDATE [OF <lista_columnas>]}
  ON nombretabla
  [FOR EACH {STATEMENT | ROW [WHEN (condicion)]}]
  < CUERPO DEL TRIGGER (BLOQUE PL/SQL)>
```

- ❑ Hay que destacar las siguientes cuestiones:

- El evento de disparo será una orden de manipulación: INSERT, DELETE o UPDATE. En el caso de esta última, se podrán especificar opcionalmente las columnas cuya modificación producirá el disparo.
- El momento en que se ejecuta el trigger puede ser antes (BEFORE) o después (AFTER) de que se ejecute la orden de manipulación.
- El nivel de disparo del trigger puede ser a nivel de orden o a nivel de fila.
- A nivel de orden (STATEMENT). El trigger se activará una sola vez para cada orden, independientemente del número de filas afectadas por ella. Se puede incluir la cláusula FOR EACH STATEMENT, aunque no es necesario, ya que se asume por omisión.
- A nivel de fila ( ROW ): el disparador se activará una vez para cada fila afectada por la orden.





## UF-7 TRIGGERS

- La restricción del trigger. La cláusula WHEN seguida de una condición restringe la ejecución del trigger al cumplimiento de la condición especificada. Esta condición tiene algunas limitaciones:
- Solamente se puede utilizar con triggers a nivel de fila (FOR EACH ROW).
- Se trata de una condición SQL, no PL/SQL.
- No puede incluir una consulta a la misma o a otras tablas o vistas.





## UF-7 TRIGGERS

- ❑ Ejemplo: El siguiente ejemplo crea un trigger que se disparará cada vez que se borre un empleado, guardando su número de empleado, apellido y departamento en una fila de la tabla auditareemple:

```
CREATE OR REPLACE TRIGGER audit_borrado_emple
BEFORE DELETE ON employees
FOR EACH ROW
BEGIN
    INSERT INTO auditareemple
    VALUES ('BORRADO EMPLEADO' || '*' || :old.employee_id || '*' || :old.first_name
    || '*Dpto.' || :old.department_id);
END;
/
```





## UF-7 TRIGGERS

- ❑ 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 respectivamente.
- ❑ Por ejemplo:

... IF :new.salary < :old.salary ...

- ❑ Al utilizar los valores old y new deberemos tener en cuenta el evento de disparo:
  - Cuando el evento que dispara el trigger es DELETE, deberemos hacer referencia a :old.nombrecolumna, ya que el valor de new es NULL.
  - Paralelamente, cuando el evento de disparo es INSERT, deberemos referirnos siempre a :new.nombrecolumna, puesto que el valor de old no existe (es NULL).
  - Para los triggers cuyo evento de disparo es UPDATE, tienen sentido los dos valores.
  - En el caso de que queramos hacer referencia a los valores new y old, al indicar la restricción del trigger (en la cláusula WHEN), lo haremos sin poner los dos puntos.

- ❑ Por ejemplo:

WHEN new.salary < old.salary





## UF-7 TRIGGERS

- ❑ Múltiples eventos de disparo y predicados condicionales
- ❑ Un mismo trigger puede ser disparado por distintas operaciones o eventos de disparo. Para indicarlo, se utilizará el operador OR.

```
CREATE TRIGGER ...  
BEFORE INSERT OR UPDATE OR DELETE ON empleados ...  
BEGIN  
    IF INSERTING THEN  
        ...  
    ELSIF DELETING THEN  
        ...  
    ELSIF UPDATING('salary') THEN  
        ...  
    END IF  
    ...  
END;
```





## UF-7 TRIGGERS

- ❑ Los disparadores de sustitución están asociados a las vistas.
- ❑ Este tipo de disparadores arrancan al ejecutarse una instrucción de actualización sobre la vista a la que están asociados. Se ejecutan en lugar de (INSTEAD OF) la orden de manipulación que produce el disparo del disparador; por eso se denominan disparadores de sustitución.
- ❑ El formato genérico para la creación de estos disparadores de sustitución es:

```
CREATE [OR REPLACE] TRIGGER nombrettrigger
INSTEAD OF
{DELETE | INSERT | UPDATE [OF <lista_columnas>]}
ON nombrevista
[FOR EACH ROW] [WHEN (condicion)]
<CUERPO DEL TRIGGER (BLOQUE PL/SQL)>.
```





## UF-7 TRIGGERS

### Activar y desactivar disparadores

- ☐ Un disparador puede estar activado o desactivado. Cuando se crea está activado, pero podemos variar esta situación mediante: `ALTER TRIGGER nombretrigger DISABLE`.
- ☐ Para volver a activarlo utilizamos: `ALTER TRIGGER nombretrigger ENABLE`.
- ☐ Para volver a compilar emplearemos: `ALTER TRIGGER nombretrigger COMPILE`.
- ☐ Para eliminar un trigger escribiremos: `DROP TRIGGER nombretrigger`.







## UF-7 TRIGGERS

- ☐ Un disparador si no se ejecuta correctamente puede generar una excepción.
- ☐ En la mayoría de los casos, si un disparador ejecuta alguna instrucción que genera un error y por lo tanto una excepción, si la excepción no se maneja dentro del disparador con un controlador de excepciones, entonces la base de datos revertirá los efectos tanto de las sentencias ejecutadas por el TRIGGER como la sentencia disparadora que lanzó el evento.
- ☐ En los siguientes casos, la base de datos revierte sólo los efectos del disparador y no los efectos de la sentencia activadora (además de registrar el error en el archivos de trace y el log de alertas):
  - ☐ El evento desencadenante es o bien AFTER STARTUP ON DATABASE o BEFORE SHUTDOWN ON DATABASE.
  - ☐ El evento desencadenante es AFTER LOGON ON DATABASE y el usuario tiene los permisos de ADMINISTER DATABASE TRIGGER.
  - ☐ El evento desencadenante es AFTER LOGON ON SCHEMA y el usuario, es propietario del esquema o tiene el privilegio ALTER ANY TRIGGER.





## UF-7 TRIGGERS

### ☐ Utilización de RAISE\_APPLICATION\_ERROR

☐ En el paquete DBMS\_STANDARD se incluye un procedimiento llamado RAISE\_APPLICATION\_ERROR que nos sirve para levantar errores y definir mensajes de error personalizados.

☐ Su formato es el siguiente:

```
RAISE_APPLICATION_ERROR(numero_error,mensaje_error);
```

☐ Es importante saber que el número de error está comprendido entre -20000 y -20999 y el mensaje es una cadena de caracteres de hasta 512 bytes.





## UF-7 TRIGGERS

Ejemplo 1: En este ejemplo se crea un TRIGGER ante un esquema llamado HR. Cuando un usuario se conecta en su intento de borrar un objeto de la base de datos, la base de datos dispara un error indicando que no se puede borrar el objeto.

```
CREATE OR REPLACE TRIGGER triggerBorrado
```

```
BEFORE DROP ON hr.SCHEMA
```

```
BEGIN
```

```
    RAISE_APPLICATION_ERROR ( num => -20000, msg => 'No se puede borrar el objeto' );
```

```
END;
```

```
/
```





## UF-7 TRIGGERS

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

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





## UF-7 TRIGGERS

❑ Para ver el funcionamiento introducimos los siguientes valores en la tabla:

```
INSERT INTO empleados VALUES ('D1','Director',null);
INSERT INTO empleados VALUES ('D2','D.Comercial','D1');
INSERT INTO empleados VALUES ('D3','D.Producción','D1');
INSERT INTO empleados VALUES ('D4','Jefe Ventas','D2');
INSERT INTO empleados VALUES ('D5','Jefe Marketing','D2');
INSERT INTO empleados VALUES ('D6','Vendedor 1','D4');
INSERT INTO empleados VALUES ('D7','Vendedor 2','D4');
INSERT INTO empleados VALUES ('D8','Vendedor 3','D4');
INSERT INTO empleados VALUES ('D9','Vendedor 4','D4');
INSERT INTO empleados VALUES ('D10','Obrero 1','D3');
INSERT INTO empleados VALUES ('D11','Obrero 2','D3');
INSERT INTO empleados VALUES ('D12','Obrero 3','D3');
INSERT INTO empleados VALUES ('D13','Secretaria','D5');
INSERT INTO empleados values ( 'D14','Obrero4','D4');
```





## UF-7 TRIGGERS

- ❑ El trigger para controlar que no haya más de cinco empleados con el mismo jefe sería el siguiente:

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

- ❑ Para contar los empleados a los que supervisa cada jefe:

```
SELECT cojefe, count(*) FROM empleados GROUP BY cojefe;
```





# UF-7 TRIGGERS

- ❑ Ejemplo de salida: Una vez que tenemos los datos introducidos si ejecutamos el siguiente script tendríamos la salida que se muestra:

```
INSERT INTO empleados values ( 'D15','Obrero5','D4');  
SELECT COJEFE,COUNT(*) FROM EMPLEADOS GROUP BY COJEFE;
```

Salida de Script x

Tarea terminada en 0,01 segundos

Error que empieza en la línea: 18 del comando :  
INSERT INTO empleados values ( 'D15','Obrero5','D4')  
Informe de error -  
Error SQL: ORA-20600: D4 no se puede supervisar mas  
de 5  
ORA-06512: at "TEL.JEFES", line 8  
ORA-04088: error during execution of trigger 'TEL.JEFES'

COJEFE	COUNT (*)
	1
D1	2
D4	5
D5	1
D2	2
D3	3

6 filas seleccionadas

