

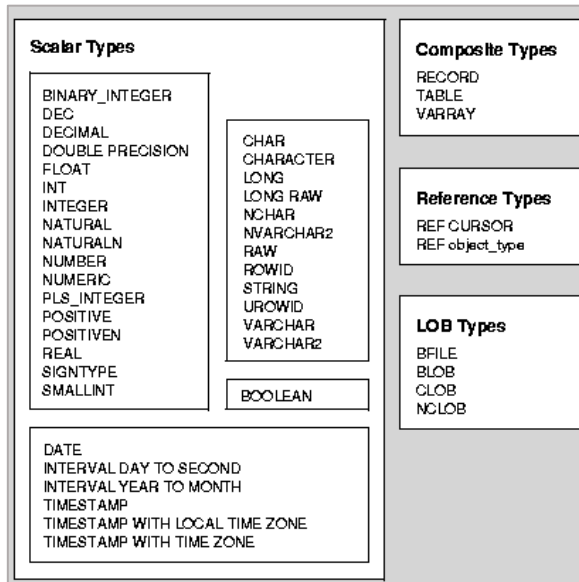
PL/SQL

Ventajas PL/SQL

- **Integración de construcciones de procedimientos con SQL:** Al emitir un comando de SQL este indica al servidor de bases de datos qué debe de hacer, sin embargo no especifica cómo debe de hacerlo. PL/SQL integra sentencias de SQL y su ejecución
- **Rendimiento mejorado:** Se pueden combinar sentencias SQL en una sola unidad del programa.
- **Desarrollo de programas basado en módulos:** La unidad básica de los programas de PL/SQL son los bloques . Los bloques pueden estar en una secuencia o anidados en otros bloques.
- **Integración con herramientas:** Esta integrado en herramientas de Oracle como Oracle Forms y Oracle Reports. Lo que quiere decir que procesa sentencias y procedimientos y solo se transfieren las sentencias SQL a la base de datos.
- **Portabilidad:** Se pueden escribir paquetes de programas portátiles y crear bibliotecas que se puedan volver aa utilizar en distintos entornos.
- **Manejo de excepciones:** PL/SQL comparte el mismo sistema de tipo de dato que SQL (solo con algunas extensiones) y utiliza la misma sintaxis de expresiones.

Tipos de datos

De la misma manera que en SQL se utilizan los mismos tipos de datos. Estos tipos de datos se pueden usar para DECLARE variables o en cursores para definir el tipo de dato que tiene cada variable dentro de los parámetros establecidos.



Existen los de tipo estacalar que dependiendo del uso qu tenga las variables y los datos que vamos a ingresar vamos a establecer sus longitud, tamaño o bien su escala (en caso de numeros)

Tenemos a los de tipo **CONPOSITE** que son tipos de datos compuestos que con ayuda de palabras clave podemos manipular dentro de la base de datos. El ejemplo más común es **TABLE** que con ayuda de sentencias DDL nos permite modificación de los datos que se alojan dentro, se pueden **CREAR**, **ALTERAR** Y **BORRAR** datos que se encuentran alojados en este tipo de dato, además de que se pueden consultar directamente de la interfaz gráfica de el server

Después se tiene a los de tipo **DATE** que nos permiten usar datos del tipo **FECHA**, con todas las variantes que ofrece la base de datos como indicador de zona horaria, tiempo, hora local, segundos, minutos, etc.

Construyendo bloques en PL/SQL

PL/SQL es un lenguaje estructurado con bloques. Cada bloque PL/SQL está definido por las palabras clave **DECLARE**, **BEGIN**, **EXCEPTION** y **END** que dividen el bloque en tres secciones:

1. **Declarativa:** Sentencias que declaran variables, constantes y distintos elementos de código. Que después pueden ser usados dentro del bloque.
2. **Ejecutable:** Sentencias ejecutables
3. **Manejo de excepciones:** Sección estructurada para manejar cualquier excepción que se produzca durante la ejecución de la sección ejecutable. No es necesario que se declare en el bloque, ni que se manejen las excepciones que se puedan lanzar

```
BEGIN
DBMS_OUTPUT.PUT_LINE ('Hola');
END;
```

Ejemplo en SQL*Plus. Se muestra: "Hola Mundo!" en la consola de SQL*Plus y se muestra que el procedimiento se completó exitosamente.
En SQL*Plus se pueden ejecutar perfectamente las sentencias de SQL y PL/SQL.

```
SQL> set serveroutput on
SQL> begin
  2  dbms_output.put_line('¡Hola Mundo!');
  3  end;
  4  /
¡Hola Mundo!
PL/SQL procedure successfully completed.
```

Variables: Se utilizan para almacenar datos y manipular valores almacenados.

- Pueden almacenar cualquier valor PL/SQL por ejemplo variables, types, cursores o subprogramas.
- Se permiten los caracteres \$ y #, no deben de tener más de 30 caracteres, pueden incluir letras o números.
- **Manejo de variables:**
 - Se pueden declarar en cualquier parte del bloque
 - Se pueden asignar un valor inicial e imponer una restricción **NOT NULL**
 - Se puede sustituir sus valor

- El operador de asignación es ':= '
- Si se desea utilizar un delimitador (! !, [], ' ') entonces se utiliza para inicializar la cadena.

Estructura de bloque:

DECLARE: Variables, cursores, excepciones definidas

BEGIN: Sentencias SQL y PL/SQL

EXCEPTION: Acciones que realizar cuando se producen excepciones

END; Obligatoria

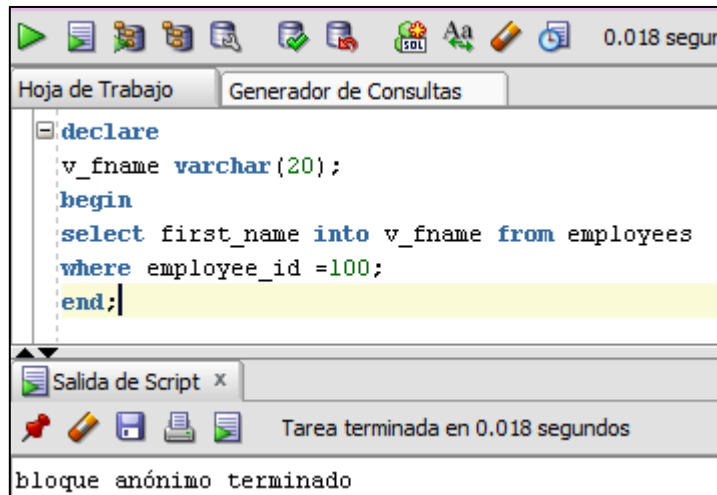
Tipos de bloques: Existen tres tipos de bloques que forman un programa PL/SQL.


- **Procedimientos:** Son objetos con nombre que contienen sentencias SQL y/o PL/SQL
- Los procedimientos pueden incluir parámetros de entrada(in).
- Para poder ejecutarlos o extraerlos se usa la función EXTRACT o también una sentencia BEGIN. En el caso de los parámetros entonces le damos a a base un parámetro de entrada y la tabla se actualiza en caso se que se agreguen datos a la tabla nueva.
- **Funciones:** Son objetos con nombre que contienen sentencias SQL y/o PL/SQL. A diferencia de los procedimientos las funciones devuelven un valor del tipo de dato específico.
- **Bloques anónimos:** Son bloques sin nombre. Se declaran en el punto de una aplicación donde se van a ejecutar y se compilan cada vez que una aplicación se ejecuta. Estos bloques no están almacenados en la base de datos
 - **Subprogramas:** Los subprogramas complementan a los bloques anónimos. Son bloques PL/SQL con nombre almacenados en la base de datos

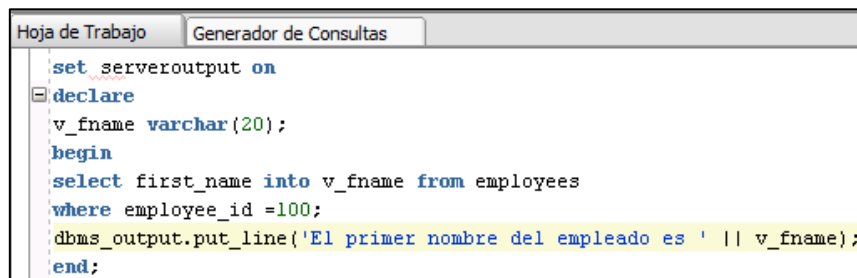
Construcciones de Programa

- Construcciones de herramienta:
 - **Bloques anónimos:** Bloques SQL/PLSQL sin nombre que están embebidos en una aplicación o se emiten de forma interactiva.

Ejemplo: El bloque anónimo obtiene first_name del empleado cuyo employee_id es 100 y lo almacena en una variable denominada v_fname



- Para ejecutar nuestro bloque anónimo dar clic en el botón Run Script() o pulsar F5
- Para activar la salida en SQL Developer, ejecutar el siguiente comando antes de ejecutar el bloque en PL/SQL: `SET SERVEROUTPUT ON`
- Utilice un paquete predefinido por Oracle y su procedimiento en bloque anónimo:
`dbms_output.put_line('El primer nombre del empleado es ' || v_fname);`
- Lo anterior usarlo como se muestra a continuación:



Y el resultado del script será:

```
bloque anónimo terminado
El primer nombre del empleado es Steven
```

Lo que se muestra es la ejecución de el comando SET SERVEROUTPUT ON y la ejecución del bloque anónimo.

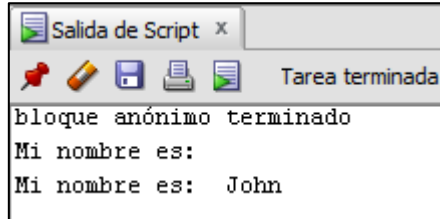
Variables: Se utilizan para almacenar datos y manipular valores almacenados. Pueden almacenar cualquier objeto PL/SQL por ejemplo, variables, tipos, cursores o subprogramas.

En las variables se admiten los caracteres '#' y '\$' así como pueden incluir números y letras. Las variables no deben contener mas de 30 caracteres.

Ejemplo de inicialización de variables:

```
DECLARE
v_miNombre VARCHAR2(20);
BEGIN
DBMS_OUTPUT.PUT_LINE('Mi nombre es: ' || v_miNombre);
v_miNombre := 'John';
DBMS_OUTPUT.PUT_LINE('Mi nombre es: ' || v_miNombre);
END;
```

- Se declara `v_miNombre` pero no se inicializa, se le asigna el valor John a la variable en la sección ejecutable. Por lo tanto la salida del bloque es:



Declaración de variables: Las variables deben de declararse dentro de la sección DECLARE, de la siguiente manera

```
Nombre_variable [CONSTANT] TIPO [NOT NULL] [:= inicialización];
```

Cualquier variable que se declare y no se inicialice tiene por defecto el valor NULL . En el tipo(TYPE) se incorpora el tipo de dato de la variable en esta parte se incluyen todos los tipos de datos existentes en SQL y algunos existentes de PL/SQL. Algunos ejemplos de la declaración de variables :

```
Interes NUMBER(5,3);
Descripcion VARCHAR2(50) := 'inicial';
Fecha_max DATE;
Contabilizado BOOLEAN := TRUE;
PI CONSTANT REAL := 3.14159;
```

Otra forma de asignarle valor a una variable es mediante la clausula INTO de la sentencia SELECT, por ejemplo:

```
SELECT COUNT(*) INTO xNumFac FROM FACTURAS;
```

DBMS_OUTPUT.PUT_LINE : Se utiliza para devolver una cadena de texto en el resultado de script

%ROWTYPE: Representa una fila parcial, tabla o view

- No se puede utilizar los nombres de columna como identificadores
- NOT NULL si la variable debe de tener un valor

%TYPE: Se declara una constante o variable, elemento, parámetro para que sea del mismo tipo de datos que un parámetro declarado anteriormente.

- No se necesita conocer el tipo exacto de la columna de la tabla

- Si se cambia la definición y/o tipo de columna de la tabla, el tipo de variable cambia automáticamente en tiempo de ejecución.

En la declaración, si tenemos la variable 'y' por ejemplo y esta declarada del tipo CHAR podemos declarar otra variable 'j' de la siguiente forma:

```
J y%type;
```

Y lo mismo sucede para declarar una estructura que ya esta declarada, por ejemplo una tabla que existe y tenemos previamente declarada:

```
J employee%rowtype; j tendrá la misma estructura que la tabla employee
```

Delimitadores de una cadena: Si la cadena contiene un apóstrofe, se puede especificar cualquier carácter que no este presente en la cadena como delimitador. Se utiliza q' para especificar el delimitador.

En el siguiente ejemplo se utiliza ¡ y [como delimitadores :

```
DECLARE
v_evento VARCHAR2(20);
BEGIN
v_evento := q'!Día del padre!';
DBMS_OUTPUT.PUT_LINE ('3er Domingo de Junio es: '||v_evento);
v_evento := q'[Día de la madre]';
DBMS_OUTPUT.PUT_LINE ('2do Domingo de mayo es: ' || v_evento);
END;
```

Y como resultado del script se obtiene:

```
bloque anónimo terminado
3er Domingo de Junio es: Día del padre
2do Domingo de mayo es: Día de la madre
```


%TYPE: Se utiliza cuando el valor almacenado de una variable en cuestión se debe derivar de una tabla de la base de datos sin saber que tipo de dato es. El atributo **%TYPE** permite declarar una constante, variable, elemento de colección, registro o parámetro de subprograma para que sea del mismo tipo de datos que un parámetro declarado anteriormente (aun sin saber cual es su tipo de dato).

Sintaxis para asignar el mismo tipo de dato de una columna de una tabla en específico:

```
nombre_variable tabla.nombre_columna%TYPE;
```

Por ejemplo, se quiere utilizar el mismo tipo de dato en la variable `nombre` para `subnombre` donde el dato es de tipo VARCHAR, no excede los 25 caracteres y contiene la restricción NOT NULL:

```
DECLARE
nombre VARCHAR(25) NOT NULL := 'Smith';
subnombre nombre%TYPE := 'Jones';
BEGIN
DBMS_OUTPUT.PUT_LINE ('nombre='||nombre);
DBMS_OUTPUT.PUT_LINE ('subnombre='||subnombre);
END;
```



```
bloque anónimo terminado
nombre=Smith
subnombre=Jones
```

Declaración de variables booleanas: PL/SQL permite comparar variables tanto en sentencias SQL como en sentencias de procedimiento. Estas comparaciones son denominadas expresiones booleanas. Las expresiones booleanas constan de expresiones simples o complejas separadas por operadores relacionales.

- A estas variables se les puede asignar los valores de TRUE, FALSE y las expresiones condicionales utilizan los operadores lógicos AND, OR y NOT

Ejemplo:

```
DECLARE
flag BOOLEAN := FALSE;
BEGIN
flag := TRUE;
END;
```

Variables LOB: Están concebidas para almacenar grandes cantidades de datos. Con la categoría LOB de los tipos de datos, puede almacenar bloques de datos no estructurados (como texto, imágenes gráficas, videoclips o sonidos) con tamaño de hasta 128TB.

- **CLOB:** Almacena grandes bloques de datos de caracteres en la DB.
- **BLOB:** Almacena grandes objetos binarios en la DB.
- **BFILE:** Almacena archivos binarios de gran tamaño.
- **NCLOB:** Almacena grandes bloques de datos Unicode NCHAR de un solo byte o multibyte de ancho fijo en la DB.

Variable de enlace: Se crean en el entorno de un host. Se pueden utilizar y manipular varios subprogramas. Para crear una variable de enlace se utiliza el comando `VARIABLE`, se puede hacer referencia de la variable y ver su valor con el comando `PRINT`

Ejemplo 1. En el siguiente bloque se utiliza la variable `b_resultado`. La salida resultante del comando `PRINT` se muestra debajo del código

```
VARIABLE b_resultado NUMBER
BEGIN
SELECT (SALARY *12) + NVL(COMMISSION_PCT,0) INTO :b_resultado
FROM employees WHERE employee_id = 144;
END;
/
PRINT b_resultado
```

Salida de Script x

Tarea terminada en 0.012 segundos

bloque anónimo terminado
B_RESULTADO

30000

Ejemplo 2. En el siguiente bloque se utiliza la variable b_emp_salary. La salida resultante del PRINT especifica en la condición `WHERE salary =:b_emp_salary;` donde asigna el salary a la variable de enlace b_emp_salary.

```
VARIABLE b_emp_salary NUMBER
BEGIN
SELECT salary INTO :b_emp_salary
FROM employees WHERE employee_id = 200;
END;
/
PRINT b_emp_salary
SELECT first_name, last_name
FROM employees
WHERE salary =:b_emp_salary;
```

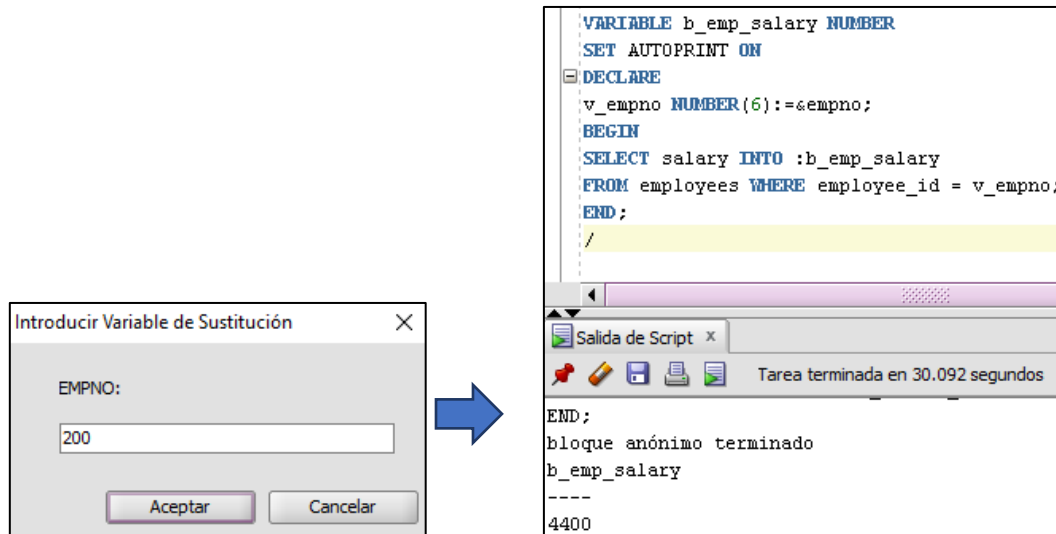
Salida de Script x

Tarea terminada en 0.011 segundos

FIRST_NAME	LAST_NAME
Jennifer	Whalen

AUTOPRINT: Permite mostrar automáticamente variables de enlace que se utilizan en los bloques de PL/SQL correctos con el comando `SET AUTOPRINT ON`

Por ejemplo, en el siguiente scrip se crea una variable de enlace `b_emp_salary` y se activa AUTOPRINT, se declara `v_empno` y se utiliza una variable de sustitución para recibir una entrada de usuario con &.



Donde:

	EMPLOYEE_ID	FIRST_NAME	LAST_NAME	SALARY
1	200	Jennifer	Whalen	4400

Conversión del tipo de dato: PL/SQL maneja las conversiones de datos con tipo de dato escalar. Las conversiones de tipo de dato pueden ser de dos tipos:

- **Conversión implícita:** PL/SQL intenta convertir los tipos de datos de forma dinámica si aparecen mezclados en una sentencia. Las conversiones implícitas se producen entre caracteres y números o entre caracteres y fechas.

Por ejemplo, la variable `v_sal_hike` es del tipo VARCHAR2. Al calcular el salario total convierte primero `v_sal_hike` a NUMBER y a continuación procede a realizar la operación. El resultado final es de tipo NUMBER.

```
DECLARE
v_salary NUMBER (6) := 6000;
v_sal_hike VARCHAR2(5) := '1000';
v_total_salary v_salary%TYPE;
BEGIN
v_total_salary :=v_salary + v_sal_hike;
```

- **Conversión explícita:** Para convertir valores de un tipo de dato a otro, se deben utilizar las funciones incorporadas(TO_DATE, TO_CHAR, TO_NUMBER)

Bloques anidados: Se pueden anidar bloques. Lo que quiere decir que se puede integrar un bloque dentro de otro.

Los bloques anidados son útiles para incluir **SELECT** u otras declaraciones que pueden generar una excepción. Si ocurre una excepción manejada por el bloque interno no afecta al bloque exterior.

Los bloques anidados no pueden estar dentro de la cláusula **DECLARE** del bloque envolvente Y solo puede comenzar dentro de un **BEGIN** o **EXCEPTION**.

Ejemplo 1. En este script se muestra un bloque externo y un bloque interno.

`v_externo_variable` es la variable respecto a bloque externo y `v_interno_variable` es la variable respecto al bloque interno.

Cuando se accede a la variable en el bloque interno PL/SQL busca primero la variable en el bloque interno y en caso de no existir entonces la busca en el bloque externo.

```
DECLARE
  v_externo_variable VARCHAR(20) := 'GLOBAL VARIABLE';
BEGIN
  DECLARE
    v_interno_variable VARCHAR2(20) := 'LOCAL VARIABLE';
  BEGIN
    DBMS_OUTPUT.PUT_LINE(v_interno_variable);
    DBMS_OUTPUT.PUT_LINE(v_externo_variable);
  END;
  DBMS_OUTPUT.PUT_LINE(v_externo_variable);
END;
```

Salida de Script x Resultado de la Consulta x

Tarea terminada en 0.062 segundos

bloque anónimo terminado
LOCAL VARIABLE
GLOBAL VARIABLE
GLOBAL VARIABLE

Ejemplo 2. En este código se tiene un bloque externo para el padre y uno interno para el hijo.

```
DECLARE
  v_nombre_padre VARCHAR2(20) := 'Patrick';
  v_fecha_de_naci DATE := '20-ABR-1972';
BEGIN
  DECLARE
    v_nombre_hijo VARCHAR2(20) := 'Mike';
    v_fecha_de_nac DATE := '12-DIC-2002';
  BEGIN
    DBMS_OUTPUT.PUT_LINE('Nombre del padre: ' || v_nombre_padre);
    DBMS_OUTPUT.PUT_LINE('Fecha de nacimiento: ' || v_fecha_de_naci);
    DBMS_OUTPUT.PUT_LINE('Nombre del hijo: ' || v_nombre_hijo);
    DBMS_OUTPUT.PUT_LINE('Fecha de nacimiento: ' || v_fecha_de_nac);
  END;
END;
```

```
bloque anónimo terminado
Nombre del padre: Patrick
Fecha de nacimiento: 20/04/1972
Nombre del hijo: Mike
Fecha de nacimiento: 12/12/2002
```

Operadores de PL/SQL

Operador	Operación
**	Exponencial
+, -	Identidad, negación
*, /	Multiplicación, división
+, -,	Adición, resta, concatenación
=, <, >, <=, >=, <>, !=, ~=, ^=, IS NULL, LIKE, BETWEEN, IN	Comparación
NOT	Negación lógica
AND	Conjunción
OR	Inclusión

Cuando se trata de operadores nulos:

- Las comparaciones con valores nulos siempre generan un NULL
- Si se usa el operador NOT a un valor nulo se genera un NULL
- En sentencias de control condicional, si la condición genera un NULL entonces la secuencia de sentencias no se ejecuta.

Recuperación de datos con **SELECT**

Ejemplo 1. Se declaran las variables v_emp_hire_date, v_emp_salary y con %TYPE se especifica que tenga el mismo tipo de dato que las columnas hire_date y salary de la tabla employees. Después dentro del BEGIN se declara SELECT donde las columnas hire_date y salary se

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	HIRE_DATE
200	Jennifer	Whalen	17/09/2011

```

DECLARE
v_emp_hire_date employees.hire_date%TYPE;
v_emp_salary employees.salary%TYPE;
BEGIN
SELECT hire_date, salary
INTO v_emp_hire_date, v_emp_salary
FROM employees
WHERE employee_id = 200;
DBMS_OUTPUT.PUT_LINE ('La fecha de ingreso es: '||v_emp_hire_date);
DBMS_OUTPUT.PUT_LINE ('El salario es de: '||v_emp_salary);
END;
```

Ejemplo 2. Muestra la suma de los salarios del departamento 60

```

DECLARE
v_sum_sal NUMBER(10,2);
v_deptno NUMBER NOT NULL := 60;
BEGIN
SELECT SUM(salary)
INTO v_sum_sal FROM employees
WHERE department_id = v_deptno;
DBMS_OUTPUT.PUT_LINE ('La suma del salario es '||v_sum_sal);
END;
```

Manipulación de datos en PL/SQL: Los datos en una base de datos se manipulan con comandos DML. En PL/SQL se pueden emitir los comandos INSERT,

UPDATE, DELETE y MERGE sin ninguna restricción. Los bloqueos de fila y bloqueos de tabla se crean utilizando las sentencias COMMIT o ROLLBACK dentro del script.

- **INSERT:** Agrega nuevas filas a la tabla.

En este ejemplo se agregan filas a la tabla employees, la tabla employees permanece sin cambios .

```
BEGIN
INSERT INTO employees(employee_id, first_name, last_name, email, hire_date, job_id, salary)
VALUES (employees_seq.NEXTVAL, 'Ruth', 'Cores', 'RCORES', CURRENT_DATE, 'AD_ASST',4000);
END;
```

- **UPDATE:** Modifica las filas de la tabla.

Se incrementa 800 al salario de los empleados que tienen como job_id = 'ST_CLERK'. Se declara una variable `incremento_sal` la cual establece el aumento y después de declara centro del UPDATE para especificar el aumento

```
UPDATE employees
```

```
DECLARE incremento_sal employees.salary%TYPE := 800;
BEGIN
UPDATE employees
SET salary = salary + incremento_sal
WHERE job_id = 'ST_CLERK';
DBMS_OUTPUT.PUT_LINE ('El aumento del salario es de: ' || incremento_sal);
END;
```

- **DELETE:** Elimina las filas e la tabla

A continuación se eliminan las filas que contengan como departamento_id =80, en este caso se declara `deptno`

```
DELETE FROM employees
```

```
DECLARE
deptno employees.department_id%TYPE := 10;
BEGIN
DELETE FROM employees
WHERE department_id =deptno;
END;
```

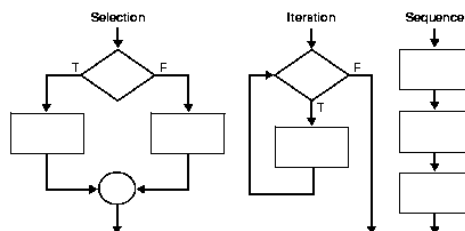
- **MERGE:** Inserta o actualiza filas en una tabla con datos otra tabla

```
MERGE INTO copy_emp c USING
EMPLOYEES e
```

```
BEGIN
MERGE INTO copy_emp c USING
EMPLOYEES e
ON (e.employee_id = c.empno) WHEN
MATCHED THEN
UPDATE SET
c.first_name = e.first_name,
c.last_name = e.last_name,
c.email = e.email,
c.phone_number = e.phone_number,
c.hire_date = e.hire_date,
c.job_id = e.job_id,
c.salary = e.salary,
c.commission_pct = e.commission_pct,
c.manager_id = e.manager_id,
c.department_id = e.department_id
WHEN NOT MATCHED THEN
INSERT VALUES(e.employee_id, e.first_name, e.last_name,
e.email, e.phone_number, e.hire_date, e.job_id,
e.salary, e.commission_pct, e.manager_id,
e.department_id);
END;
```

Estructuras de control: Cualquier lenguaje de programación puede escribirse utilizando estructuras de control básicas. Se pueden combinar de la forma que e¿sea necesaria para resolver un problema dado.

La estructura **Selection** prueba una condición para después ejecutar una secuencia de sentencias, esto dependiendo de si es verdadera o falsa. Donde una *condición* es cualquier variable o expresión que devuelve un valor booleano (TRUE o FALSE). La estructura de **iteration** ejecuta una secuencia de declaraciones repetidamente siempre que una condición se cumpla. La estructura de **Sequence** ejecuta una secuencia de declaraciones en el orden en el que ocurren.



- **IF THEN:** Se evalúa la condición y si es verdadera entonces se ejecutan una o más líneas de código. En el caso de que la condición sea falsa entonces no se realiza ninguna acción. La manera mas simple de IF asocia una condición con una secuencia de declaraciones entre las palabras clave THEN y END IF, como se muestra:

```
IF condicion THEN
secuencia_de_declaraciones
END IF;
```

- **IF-THEN-ELSE:** En este caso se agrega la palabra clave ELSE seguida de una secuencia de declaraciones, de la siguiente manera:

```
IF condicion THEN
secuencia_de_declaraciones1
ELSE
secuencia_de_declaraciones2
END IF;
```

Se evalúa la condición y si resulta verdadera entonces se ejecutan una o más líneas de código del programa. En caso de que la condición sea falsa entonces se ejecuta la instrucción que sigue en la instrucción ELSE. Solo se permite una instrucción else por cada IF.

- **IF-THEN-ELSIF:** En este caso se incorpora la palabra clave ELSIF para introducir condiciones adicionales.

```
IF condicion1 THEN secuencia_de_declaraciones1
ELSIF condicion2 THEN secuencia_de_declaraciones2
ELSE secuencia_de_declaraciones
END IF;
```

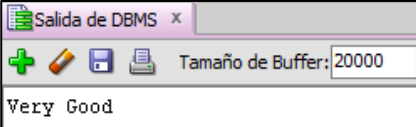
Control de flujo de ejecución. Se puede cambiar el flujo lógico de nuestro código PL/SQL con varias estructuras de control. Existen cuatro tipos de estructuras de control:

- **Sentencia IF:** Permite realizar acciones de manera selectiva según la condición. En este caso pueden existir muchas sentencias de comparación como ELSE, ELSIF, WHERE y otros parámetro de condición.
- **CASE:** Compara los valores de una expresión. Funciona de la misma manera que un ELSIF-THEN-ELSE. Se permite usar los operadores de relación THEN, BETWEEN, AND, OR. Modelo relacional, relaciona las columnas de una tabla. Existen dos tipos de CASE
- **CASE Simple:** Realiza una verificación de igualdad de "n" contra cada una de las opciones "when". Evalúa una sola expresión y compara el resultado con algunos valores. 'accion1', 'accion2'... se evalúan secuencialmente. Si el resultado de una acción es igual al resultado de n la secuencia se ejecuta y el CASE finaliza.

```
case n
when 1 then accion1
when 2 then accion2
else otra_accion
end case;
```

Ejemplo de CASE Simple: Se crean las variables c_grade y c_rank y después se le asigna el valor de 'B' a c_grade. Donde CASE es c_grade, verifica c_grade con cada instrucción WHEN.

```
DECLARE
  c_grade CHAR( 1 );
  c_rank  VARCHAR2( 20 );
BEGIN
  c_grade := 'B';
  CASE c_grade
    WHEN 'A' THEN
      c_rank := 'Excellent' ;
    WHEN 'B' THEN
      c_rank := 'Very Good' ;
    WHEN 'C' THEN
      c_rank := 'Good' ;
    WHEN 'D' THEN
      c_rank := 'Fair' ;
    WHEN 'F' THEN
      c_rank := 'Poor' ;
    ELSE
      c_rank := 'No such grade' ;
    END CASE;
  DBMS_OUTPUT.PUT_LINE( c_rank );
END;
```



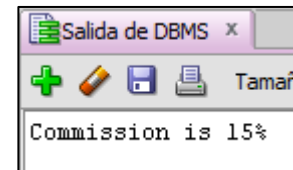
The screenshot shows a window titled 'Salida de DBMS' with a toolbar containing icons for adding, deleting, saving, and printing. Below the toolbar, it says 'Tamaño de Buffer: 20000' and displays the output 'Very Good'.

- **CASE Search:** Evalua múltiples expresiones booleanas y ejecuta la secuencia de sentencias asociadas con la primera condición que da como resultado TRUE. Las condiciones WHEN se evalúan en orden y se ejecuta la secuencia de declaraciones asociadas con WHEN cuya condición se evalúa como TRUE. Si mas de una es TRUE entonces solo ejecuta a primera y si ninguna es TRUE entonces se ejecuta el ELSE. En caso se que se omita ELSE y ninguna expresión sea verdadera entonces se genera CASE_NOT_FOUND.

```
case
when condicion1 then declaracion1
when condicion2 then declaracion2
...
when condicionn then declaracionn
[else
otras_declaraciones]
end case;
```

*Ejemplo de **CASE Search**: Se solicita mostrar que dependiendo el número de ventas de una tienda se gane un porcentaje de comisión. Se declaran las variables *n_ventas* (numero de ventas) y *n_comision* (numero de comisión). Se asigna que el *n_ventas* es 150000. Y dentro de *case* se generan las condiciones de cada WHEN con los operadores <, >, >= y AND.*

```
DECLARE
  n_VENTAS    NUMBER;
  n_comision  NUMBER;
BEGIN
  n_ventas := 150000;
  CASE
    WHEN n_ventas > 200000 THEN
      n_comision := 0.2;
    WHEN n_ventas >= 100000 AND n_ventas < 200000 THEN
      n_comision := 0.15;
    WHEN n_ventas >= 50000 AND n_ventas < 100000 THEN
      n_comision := 0.1;
    WHEN n_ventas > 30000 THEN
      n_comision := 0.05;
    ELSE
      n_comision := 0;
    END CASE;
  DBMS_OUTPUT.PUT_LINE('Commission is ' || n_comision * 100 || '%');
END;
```



Bucle Loop y EXIT: Estas sentencias permiten ejecutar una secuencia de sentencias varias veces. Existen tres tipos

- **LOOP:** Es un bucle básico (o infinito) que encierra una secuencia de declaraciones entre las palabras clave LOOP y END LOOP:

```
LOOP
  secuencia_de_sentencias
END LOOP;
```

Con cada iteración del LOOP se ejecuta una secuencia de instrucciones y luego se reanuda el control en la parte superior del ciclo. Si el proceso no es deseable o imposible entonces se puede usar EXIT para completar el LOOP.

- **EXIT:** Obliga a un LOOP a completarse incondicionalmente. Cuando EXIT se encuentra entonces el LOOP se completa inmediatamente y el control pasa a la siguiente declaración. Debe de estar dentro de un bucle.
- **EXIT WHEN:** Permite que el LOOP se complete condicionalmente. Cuando EXIT se encuentra en la declaración, se evalúa la condición en la cláusula WHEN. Si la condición es verdadera, el ciclo se completa y el control pasa a la siguiente declaración después del ciclo. Hasta que la condición sea verdadera, el ciclo no puede completarse.

Procedimientos y funciones : Son subprogramas compuestos por un conjunto de sentencias SQL. Los procedimientos o funciones PL/SQL pueden realizar distintas tareas dependiendo los parámetros que se les hayan pasado. Están compuestos por una parte que se define de variables y cursores y otra parte ejecutable compuesta por sentencias SQL y PL/SQL y otra parte enfocada a el manejo de excepciones y errores ocurridos durante la ejecución.

- Procedimiento: Para los procedimientos, se pueden crear parámetros,
- Para la excusión del procedimiento se utiliza la función EXECUTE

```
execute nombre_procedimiento();
```

dentro del paréntesis se agregan los parámetros y en caso de que no se quiera agregar se pone la palabra 'NULL'.

■

La diferencia entre los procedimientos funciones es que las funciones devuelven un valor al bloque PL/SQL que la llamo (un único valor), sin embargo en los procedimientos se pueden definir múltiples parámetros de salida.

Sintaxis de procedimiento:

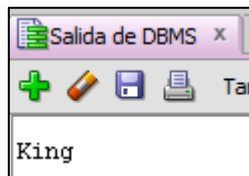
```
CREATE OR REPLACE PROCEDURE [esquema].nombre-procedim
(nombre-parámetro {IN, OUT, IN OUT} tipo de dato, ..) {IS, AS}
  Declaración de variables;
  Declaración de constantes;
  Declaración de cursores;
  BEGIN
    Cuerpo del subprograma PL/SQL;
  EXCEPTION
    Bloque de excepciones PL/SQL;
  END;
```

Sintaxis de función:

```
CREATE OR REPLACE FUNCTION [esquema].nombre-funcion
(nombre-parámetro IN tipo-de-dato, ..)
RETURN tipo-de-dato {IS, AS}
  Declaración de variables;
  Declaración de constantes;
  Declaración de cursores;
  BEGIN
    Cuerpo del subprograma PL/SQL;
  EXCEPTION
    Bloque de excepciones PL/SQL;
  END;
```

- Ejemplo practico de un procedimiento: Dela base de datos HR se pide mostrar el primer apellido del empleado de la tabla EMPLOYEES.

```
CREATE OR REPLACE
PROCEDURE primer_apellido
IS
  v_apellido employees.last_name%type;
BEGIN
  SELECT last_name
  INTO v_apellido
  FROM employees
  WHERE employee_id = 100;
  DBMS_OUTPUT.PUT_LINE ('Algo salio mal');
END;
/
--ejecucion
BEGIN
  primer_apellido();
  --rollback;
END;
```



Funciones de tabla: Una función de tabla es una función que se puede invocar dentro de la cláusula FROM de una instrucción SELECT. Es una función PL/SQL definida por el usuario que devuelve una colección de filas.

```
SELECT * FROM TABLE(table_function_name(parameter_list))
```

Consideraciones para llamar una función desde la cláusula FROM de una consulta se debe de:

Definir el tipo de datos RETURN, 'para que sea un tipo de colección. En este caso cuando se incorpora return es porque se desea ingresar un parámetro.

Para llamar una función necesario usar DESCRIBE

Todos los parámetros deben de ser del modo IN y deben tener tipos compatibles con SQL. (Por ejemplo no se puede llamar a una función con argumento booleano).

Crear una función de tabla:

```
CREATE OR REPLACE TYPE strings_t IS TABLE OF VARCHAR2 (100);  
/
```

```
CREATE OR REPLACE FUNCTION random_strings (count_in IN INTEGER)  
RETURN strings_t  
AUTHID DEFINER  
IS  
  l_strings strings_t := strings_t ();  
BEGIN  
  l_strings.EXTEND (count_in);  
  
  FOR indx IN 1 .. count_in  
  LOOP  
    l_strings (indx) := DBMS_RANDOM.string ('u', 10);  
  END LOOP;  
  
  RETURN l_strings;  
END;  
/
```

```
DECLARE  
  l_strings strings_t := random_strings (5);  
BEGIN  
  FOR indx IN 1 .. l_strings.COUNT  
  LOOP  
    DBMS_OUTPUT.put_line (l_strings (indx));  
  END LOOP;  
END;  
/
```

1.Creamos el tipo de tabla anidada que devolverá la función.

2.Creamos la función que devuelve una matriz de cadenas aleatorias.

3.Se demuestra que la función funciona.

Cursores: Un cursor es un puntero al área de memoria privada asignada por Oracle Server.

Existen dos tipos de cursores:

- **Implícitos:** Como su nombre lo dice son cursores que están creados de manera implícita lo que quiere decir que no se declaran y Oracle Server crea y gestiona este tipo de cursores. Se usan cuando la consulta devuelve un único registro. En caso de que devuelva mas de una fila entonces generara una excepción.

Ejemplo de un cursor implícito:

Se declara una variable que almacenara lo que se solicita en nuestra instrucción **SELECT**.

Después a través de un **DBMS** se mostrará lo que se almaceno. En este caso tal cual no se esta declarando el cursor como tal, sin embargo si nos ponemos analíticos se logra percibir que la variable es la que esta funcionando como cursor y que devuelve únicamente una fila que como indicamos tiene que cumplir con la condición indicada en el **WHERE**.

```
SET SERVEROUTPUT ON;
declare
  vdescripcion VARCHAR2(50);
begin
  SELECT DESCRIPCION INTO vdescripcion from PAISES WHERE CO_PAIS = 'ESP';
  dbms_output.put_line('La lectura del cursor es: ' || vdescripcion);
end;
```

- **Explícitos:** Son cursores declarados y controlados por el programador. Se utiliza cuando la consulta devuelve un conjunto de registros. Para trabajar con este tipo de cursores es necesario seguir los siguientes pasos:

1. Declarar el cursor.

Al igual que cualquier otra variable el cursor se declara en la sección **DECLARE**. Se define nuestro cursor y se indica que consulta **SELECT** ejecutará

```
CURSOR nombre_cursor IS instrucción_SELECT
```

Una vez que el cursor fue declarado entonces puede usarse dentro de un bloque de código. Ahora bien, antes de utilizar un cursor es necesario abrirlo, al momento de abrirlo entonces se ejecutara la sentencia **SELECT** asociada y se almacena en el servidor en un área de memoria interna.

2. Abrir el cursor en el servidor

```
OPEN nombre_cursor;
```

Al abrir un cursor el puntero señalara al primer registro y una vez que el cursor fue abierto entonces se podrán pedir resultados al servidor.

3. Recuperar cada una de sus filas (bucle)

Cuando se abre el cursor entonces se puede solicitar al servidor tener acceso a la recuperación de filas. Como en cada recuperación solo se accede a una fila a la vez. Par recuperar una fila se utiliza la instrucción **FETCH**:

```
FETCH nombre_cursor INTO variables;
```

Se podrán recuperar filas siempre y cuando la instrucción SELECT tenga filas pendientes de recuperar. Para saber si no hay mas filas entonces se pueden consultar los atributos del cursor

4. Cerrar el cursor

Una vez que se han recuperado todas las filas del cursor, hay que cerrarlo para que se liberen de la memoria del servidor los objetos temporales creados

```
CLOSE nombre_cursor;
```

Se debe de considerar que:

- Cuando un cursor está cerrado no se puede leer.
- Cuando se lee un cursor se debe de corroborar el resultado
- El nombre del cursor se utiliza como identificador por lo tanto no se puede utilizar en os parámetros.

Atributos de los cursores:

Los cursores implícitos no se pueden manipular por el usuario, pero Oracle permite utilizar sus atributos. En este caso m se debe de anteponer al nombre de atributo prefijo SQL en lugar del nombre del cursor.

- **SQL%FOUND:** Atributo booleano que se evalúa como TRUE(verdadero) cuando la ultima sentencia SELECT devuelve alguna fila o cuando INSERT, DELETE o UPDATE no afectan ninguna fila
- **SQL%NOTFOUD:** Devuelve TRUE cuando la última sentencia SELECT no recupero ninguna fila, o cuando la ultima sentencia SELECT no recupero ninguna fila o INSERT, DELETE o UPDATE no afectan a ningún fila
- **SQL%ROWCOUNT:** Valor entero que devuelve el número de filas afectadas por un INSERT, DELETE o UPDATE o las filas devueltas por una sentencia SELECT
- **SQL%ISOPEN:** Siempre devuelve un FALSE, por que Oracle cierra de manera automática el cursor implícito cuando termina la ejecución de la sentencia SELECT.

REF CURSOR: Como ya sabemos, un cursor es un puntero en un área privada de SQL que almacena información sobre el procesamiento de una sentencia **SELECT** o lenguaje de manipulación de datos(**DML**)(**INSERT,UPDATE, DELETE o MERGE**).

Por lo tanto un **REF CURSOR** es una variable, definida como un tipo de cursor, que apuntara o hará referencia a un resultado de un cursor.

Entre las ventajas que abarcan a los **REF CURSOR** es que a diferencia de un cursor normal, un **REF CURSOR** puede ser pasado como variable a un procedimiento y/o una función. Otra ventaja es que un REF CURSOR puede ser asignado a otras variables REF CURSOR. También un REF CURSOR puede ser el valor de retorno de una función.

Se debe tener en claro que un REF CURSOR no es un cursor como tal, si no una variable que apunta a un cursor. Antes de asignar una variable, debe definirse un tipo cursor.

Weak REF CURSOR: Se dice que un REF CURSOR es débil cuando no se define el tipo de dato que retorna.

```
TYPE typ_ref_cur IS REF CURSOR;
```

Strong REF CURSOR: En este tipo de cursor se especifica que se va a devolver. Teniendo en cuenta que si se quiere devolver algo distinto a su tipo de retorno, se genera una excepción **"ROWTYPE_MISMATCH"**

```
TYPE typ_ref_cur IS REF CURSOR  
RETURN hr.employees%ROWTYPE;
```

Manejo de excepciones.

El código solo identifica las excepciones que se identifican, existen dos tipos de excepciones:

Explicitas: Son claramente definidas por el usuario.

implícitas: Pueden ser predefinidas de Oracle Server o no PREDEFINIDAS POR ORACLE SERVER.

WHEN OTHERS: Cuando una excepción no se identifica entonces, al final del código se grega la excepción WHEN OTHERS seguido de este ejecutamos un COMMIT que indica que no habrá un ROLLBACK en caso de que se desee.

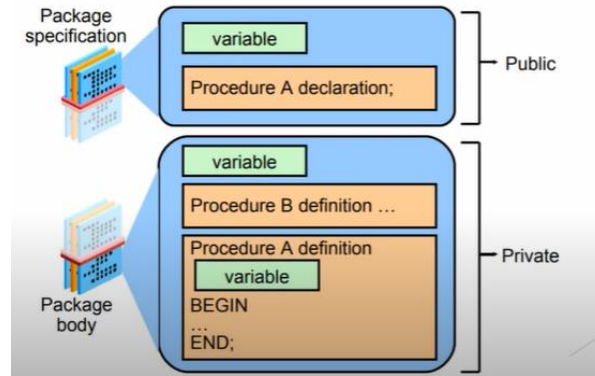
También las excepciones se pueden incluir los errores que arroja Oracle por defecto y evitar que marque error, declarando excepciones predefinidas.

PACKAGES

Un paquete(package) es un objeto de esquema que agrupa varios tipos de variables, constantes subprogramas, cursores y excepciones de PL/SQL relacionados lógicamente. Un paquete siempre tendrá una especificación, que declara los elementos públicos a los que se pueden hacer referencia fuera del paquete.

Un paquete se divide en dos partes:

1. Especificación: Describe el contenido del paquete, este puede incluir distintos objetos que se encuentren dentro de un esquema en una base de datos. Estos objetos pueden incluir Procedimientos, Funciones, Cursores y variables que compongan nuestro paquete.
2. Cuerpo: Incluye el código de la especificación, donde se agregan los requerimientos que queremos que cumpla nuestro paquete. Cada elemento que se agrega al paquete funciona de manera independiente y al mismo tiempo están almacenados dentro de un mismo paquete.



Ejemplo de estructura del paquete

```
CREATE PACKAGE citi AS
    FUNCTION p_strng RETURN VARCHAR2;
END citi;
/
```

Sintaxis del cuerpo del paquete

```
CREATE [OR REPLACE] PACKAGE BODY <package_n>
IS
    < private element definition >
    < function, procedure and public element definition >
BEGIN
    < package initialization code >
END < package_n >
```

Los elementos públicos de los paquetes incluyen cursores o subprogramas, estos deben de contener un cuerpo para que nuestro paquete funcione. El cuerpo puede declarar o definir elementos privados a los que no se puede hacer referencia fuera del paquete, esto se puede hacer con una serie de cláusulas especificada por el usuario titular del paquete.

A los paquetes se le pueden agregar ciertos permisos para que determinados usuarios de la base de datos y que tengan acceso al mismo esquema tengan los mismos permisos.

Ejemplo: Se requiere un paquete que agrupe una serie de funciones y procedimientos.

- Una función que muestre al décimo empleado con el salario más alto
- Un procedimiento que muestre el número de empleados que son ingresados
- Función que muestre el id del empleado (employee_id) que se mostrara en la primera función .

Solución: Para este ejemplo requerimos crear la tabla LOG

```
CREATE TABLE log
(
    date_of_action DATE,
    user_id        VARCHAR2(20),
    package_name   VARCHAR2(30)
);
```

TRIGGERS

Son bloques de código PL/SQL que se ejecutan automáticamente cuando se requieren hacer modificaciones a cierta tabla

Que contiene tres variables, en este caso son variable que no son vistas por el usuario pero que permanecen de manera privada dentro del paquete como reacción a una operación DML (INSERT, DELETE, UPDATE) específica sobre dicha tabla.

Cuando se utiliza la sentencia ON REPLACE dentro de un trigger se puede sobrescribir otro trigger. Así mismo dentro de un trigger también se pueden llamar procedimientos y funciones siempre y cuando:

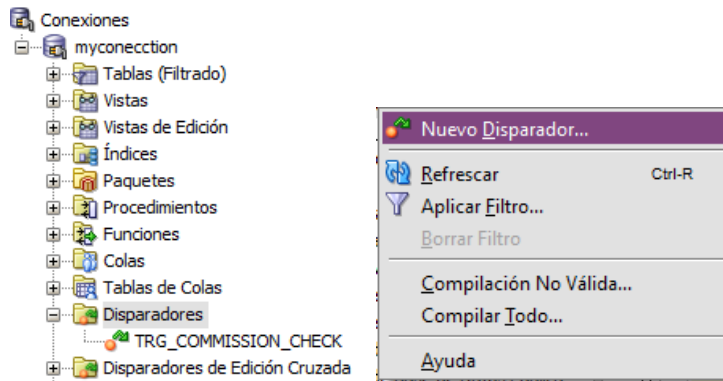
- No utilicen declaraciones DML.
- No utilicen sentencias de control
- No modifiquen o alteren la secuencia del programa, cuidar el manejo de excepciones dentro de los posibles procedimientos y funciones.
- No alteren las reglas de integridad, como tratar de modificar primary keys o foreign keys.

Ahora bien, cuando el triggers ha sido disparado por un INSERT, DELETE y UPDATE entonces se devolverá **true**.

Ahora para el manejo de las palabras clave **BEFORE** y **AFTER**. Se deben considerar que antes de que se altere una columna o una fila se utiliza BEFORE y despues AFTER.

Crear un trigger

1. Abrimos nuestro esquema HR dentro de nuestra conexión.



Crear Disparador

Esquema: ORA21

Nombre: TRIGGER1

☐ Agregar Nuevo Origen en Minúsculas

Disparador: DDL

Tipo de Disparador: TABLE ☒ Activado

Propietario de Tabla: ORA21 Nombre de la Tabla:

☐ Insertar ☐ Suprimir ☐ Actualizar

Referencia

Antiguo: Nuevo:

Antes de ☒ Nivel de Sentencia
Después de ☐ Nivel de Fila

Cuando:

Columnas Disponibles

Columnas Seleccionadas

Ayuda Aceptar Cancelar

Una vez que se nos abrió esta ventana entonces procedemos a llenar nuestros campos. Para este ejemplo utilizaremos los datos siguientes:

Tipo de disparador > Table > Actualizar

Nombre de la tabla > LIBROS> Antes de de > Nivel Sentencia

Columnas disponibles> Precio

Crear Disparador

Esquema: ORA21

Nombre: TR_EJEMPLO

☐ Agregar Nuevo Origen en Minúsculas

Disparador: DDL

Tipo de Disparador: TABLE ☒ Activado

Propietario de Tabla: ORA21 Nombre de la Tabla: LIBROS

☐ Insertar ☐ Suprimir ☒ Actualizar

Referencia

Antiguo: Nuevo:

Antes de ☒ Nivel de Sentencia
Después de ☐ Nivel de Fila

Cuando:

Columnas Disponibles

PRECIO

Columnas Seleccionadas

CODIGO
TITULO
AUTOR
EDITORIAL

Ayuda Aceptar Cancelar

Una vez capturado nuestro trigger entonces se nos arrojará el siguiente código:

```
1 create or replace trigger tr_ejemplo
2 before update of CODIGO,TITULO,AUTOR,EDITORIAL on LIBROS
3 begin
4   null;
5 end;
```

En NULL podremos agregar las especificaciones de lo que queremos que ejecute nuestro programa. En este caso por ser un ejemplo no se agregarán especificaciones.

Errores definidos por el usuario en Trigger: Para definir errores en Oracle existe la sentencia `RAISE_APPLICATION_ERROR` que Emite directamente un mensaje de error en caso de que este previsto.

Como consideraciones de `RAISE_APPLICATION_ERROR` se debe tener que:

```
RAISE_APPLICATION_ERROR(numero, texto);
```

- El numero de mensaje debe de ser un numero negativo entre -2000 y -20999
- El texto del mensaje debe de ser una cadena de caracteres de hasta 2048 bytes.

Ejemplo: Crear un trigger de actualización a nivel fila sobre la tabla LIBROS. Ante cualquier modificación de los registros de LIBROS, se debe ingresar en la tabla CONTROL, el nombre del usuario que realizo la actualización y la fecha ;pero, controlar que NO se permita modificar el campo “código” en caso de suceder, en caso contrario mostrar un mensaje de error.

```
create or replace trigger tr_actualizar_libros
before update
on libros
for each row
begin
if updating('codigo') then
raise_application_error(-20001, 'No se puede modificar el código de los Libros');
else
insert into control values (user, sysdate);
end if;
end;
```

A través de un IF en el bloque anónimo se indica que se arroje el error.

```
update libros
set codigo = 100
where autor = 'Borges';
```

Si se intentara modificar cualquier dato de la columna de código entonces automáticamente aparece el siguiente error:

```
Informe de error:
Error SQL: ORA-20001: No se puede modificar el código de los Libros
ORA-06512: en "ORA21.TR_ACTUALIZAR_LIBROS", línea 3
ORA-04088: error durante la ejecución del disparador 'ORA21.TR_ACTUALIZAR_LIBROS'
```

Trigger uso de OLD y NEW: Cuando se trabaja con un trigger a nivel fila, Oracle provee dos tablas temporales a las cuales se puede acceder. Estas tablas contienen los valores antiguos y nuevos de los campos del registro afectado (esto con la instrucción UPDATE)

Ejemplo: Crear un trigger que a nivel fila se dispare “antes ” de que se ejecute un “update ” sobre el campo “precio” de la tabla “libros”. En el cuerpo del disparador se debe de ingresar en la tabla “control” el nombre de usuario que realizó la actualización, la fecha, el código del libro que ha sido modificado el precio anterior y el nuevo.

- A continuación, se creo el trigger que actualiza el precio de los libros y almacena los cambios en la tabla CONTROL donde además se muestra el valor anterior y el nuevo de la tabla libros, junto con la fecha y el usuario que realizo los cambios.

```
create or replace trigger tr_actualizar_precio_libros
before update of precio
on libros
for each row
begin
if (:new.precio > 50 )then
:new.precio:= floor(:new.precio);
end if;
insert into control values(user, sysdate, :new.codigo, :old.precio, :new.precio);
end tr_actualizar_precio_libros;
```

- Se obtiene la siguiente salida, donde se muestran 3 actualizaciones con sus respectivos

precios anteriores y nuevos:

AZ	USUARIO	AZ	FECHA	AZ	CODIGO	AZ	PRECIOANTERIOR	AZ	PRECIONUEVO
	ORA21		13/01/2023		120		55		60
	ORA21		13/01/2023		100		25		300
	ORA21		13/01/2023		100		300		54

- Cuando se actualizan varias filas, debido a que se uso FOR EACH ROW entonces se actualiza la tabla y las actualizaciones quedan registradas en la tabla CONTROL

```
update libros
set precio = precio + precio * 0.1
where editorial = 'Planeta';
```

2 filas actualizadas.

ORA21	13/01/2023	100	54	59
ORA21	13/01/2023	145	35	38.5

Deshabilitar o habilitar un trigger. Una de las funciones dentro de los trigger es ue se pueden habilitar o deshabilitar. Esto funciona para que en caso de que la tabla reciba otra trigger que afecte a los mismos datos de una previamente creado, entonces se sepa identificar de manera correcta los cambios. Esta acción es muy sencilla:

```
alter trigger nombre_del_trigger disable;
alter trigger nombre_del_trigger enable;
```

Funciona de la misma manera que cuando se quiere “alterar” algún objeto existente dentro de un mismo schema.

Si se deseara habilitar o deahabilitar TODOS los triggers que existen dentro de una misma tabla.

```
alter table nombre_tabla disable all triggers;  
alter table nombre_tabla enable all triggers;
```

Ejercicios de práctica

1. Crear un bloque anónimo que muestre los primeros 10 múltiplos de un numero dado e indique si es par o impar.

Por ejemplo, para el múltiplo 2 de 5 es 10 y es par

El múltiplo 3 de 5 es 15 y es impar

- Se declaran dos variables

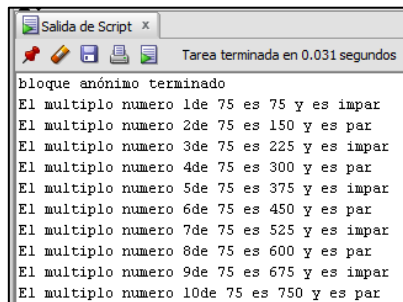
```
CONTADOR NUMBER(4) := 1;  
NUMERO NUMBER(2) :=75;
```

En este caso 1 para que el contador inicie en 1 y el número se asigna a 75 por que será el número del que se calcula el múltiplo.

- Después se declara un **WHILE**
- Se establece que el contador continúe en un **LOOP** donde se detenga cuando sea < 11 (menor a 11)
- Se inicia el IF donde se incluye la función **MOD()** que devuelve el módulo entre el CONTADOR y 2. Por ejemplo **MOD(m,n)** donde m= contador que en este caso inicia en 1 y n = 2

```
DECLARE  
    CONTADOR NUMBER(4) := 1;  
    NUMERO NUMBER(2) :=75;  
BEGIN  
    WHILE CONTADOR < 11 LOOP  
        IF MOD(CONTADOR, 2)=0 THEN  
            DBMS_OUTPUT.PUT_LINE('El multiplo numero ' || TO_CHAR (CONTADOR) || 'de ' || NUMERO || ' es ' || TO_CHAR (NUMERO * CONTADOR) || ' y es par');  
        ELSE  
            DBMS_OUTPUT.PUT_LINE('El multiplo numero ' || TO_CHAR(CONTADOR) || 'de ' || NUMERO || ' es ' || TO_CHAR (NUMERO * CONTADOR) || ' y es impar');  
        END IF;  
        CONTADOR := CONTADOR + 1;  
    END LOOP;  
END;  
/
```

Y como salida obtenemos:



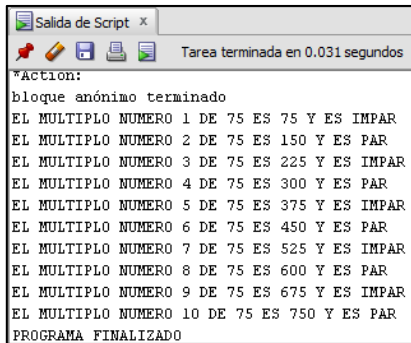
Salida de Script x
Tarea terminada en 0.031 segundos

```
bloque anónimo terminado  
El multiplo numero 1de 75 es 75 y es impar  
El multiplo numero 2de 75 es 150 y es par  
El multiplo numero 3de 75 es 225 y es impar  
El multiplo numero 4de 75 es 300 y es par  
El multiplo numero 5de 75 es 375 y es impar  
El multiplo numero 6de 75 es 450 y es par  
El multiplo numero 7de 75 es 525 y es impar  
El multiplo numero 8de 75 es 600 y es par  
El multiplo numero 9de 75 es 675 y es impar  
El multiplo numero 10de 75 es 750 y es par
```

- Una alternativa para nuestro programa es
- No se usa **WHEN**
- Directamente al final del código se establece que el CONTADOR termine en 10

```
EXIT WHEN CONTADOR > 10;
```

```
DECLARE
    CONTADOR NUMBER(4) := 1;
    NUMERO NUMBER(2) := 75;
BEGIN
    LOOP
        IF MOD(CONTADOR, 2)=0 THEN
            DBMS_OUTPUT.PUT_LINE('EL MULTIPLO NUMERO '||TO_CHAR(CONTADOR)||' DE '||NUMERO||' ES '||TO_CHAR(NUMERO * CONTADOR)||' Y ES PAR');
        ELSE
            DBMS_OUTPUT.PUT_LINE('EL MULTIPLO NUMERO '||TO_CHAR(CONTADOR)||' DE '||NUMERO||' ES '||TO_CHAR(NUMERO * CONTADOR)||' Y ES IMPAR');
        END IF;
        CONTADOR := CONTADOR + 1;
        EXIT WHEN CONTADOR > 10;
        /* DBMS_OUTPUT.PUT_LINE('PROGRAMA FINALIZADO'); -- preguntar porque esto se ejecuta */
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('PROGRAMA FINALIZADO');
END;
```



```
*Action:
bloque anónimo terminado
EL MULTIPLO NUMERO 1 DE 75 ES 75 Y ES IMPAR
EL MULTIPLO NUMERO 2 DE 75 ES 150 Y ES PAR
EL MULTIPLO NUMERO 3 DE 75 ES 225 Y ES IMPAR
EL MULTIPLO NUMERO 4 DE 75 ES 300 Y ES PAR
EL MULTIPLO NUMERO 5 DE 75 ES 375 Y ES IMPAR
EL MULTIPLO NUMERO 6 DE 75 ES 450 Y ES PAR
EL MULTIPLO NUMERO 7 DE 75 ES 525 Y ES IMPAR
EL MULTIPLO NUMERO 8 DE 75 ES 600 Y ES PAR
EL MULTIPLO NUMERO 9 DE 75 ES 675 Y ES IMPAR
EL MULTIPLO NUMERO 10 DE 75 ES 750 Y ES PAR
PROGRAMA FINALIZADO
```

2. Construir un bloque PL/SQL que escriba en la pantalla la cadena 'ORACLE' al revés (ELCARO).

- Se declaran 3 variables

```
palabra VARCHAR2(10) := 'ORACLE';
invertida VARCHAR2(10);
contador NUMBER(2) := LENGTH('ORACLE');
```

Palabra que contiene la palabra a invertir

Invertida que invertirá nuestra cadena con ayuda de **SUBSTR**

Contador que con **LENGHT** devuelve la longitud de nuestra cadena 'ORACLE'

```
DECLARE
    palabra VARCHAR2(10) := 'ORACLE';
    invertida VARCHAR2(10);
    contador NUMBER(2) := LENGTH('ORACLE');
BEGIN
    WHILE contador > 0 LOOP
        invertida := invertida || SUBSTR(palabra, contador, 1);
        contador := contador - 1;
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('ORACLE escrito al revés es: ' || invertida);
END;
```

Y como resultado devuelve:

```
bloque anónimo terminado
ORACLE escrito al revés es: ELCARO
```

- Otra alternativa, usando la clausula **FOR** y con la declaración **IN REVERSE**

```
DECLARE
    palabra VARCHAR2(10) := 'ORACLE';
    invertida VARCHAR2(10);
BEGIN
    FOR i IN REVERSE 1..LENGTH(palabra) LOOP
        invertida := invertida || SUBSTR(palabra, i, 1);
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('ORACLE escrito al revés es: '||INVERTIDA);
END;
/
```

Y devuelve el mismo resultado:

```
bloque anónimo terminado
ORACLE escrito al revés es: ELCARO
```

3. Crear un bloque anónimo que a partir de un parámetro de entrada muestre la tabla de multiplicar de un numero dado. Únicamente permitirá ingresar valores numéricos de lo contrario que muestre el mensaje de error usando manejo de excepciones.

```
SET SERVEROUTPUT ON
SET VERIFY OFF
DECLARE
    e_numero_invalido EXCEPTION;
    v_entrada VARCHAR2(4);
    v_numero NUMBER;
    PRAGMA EXCEPTION_INIT(e_numero_invalido, -06502);
BEGIN
    v_entrada := '&numero';
    v_numero := v_entrada;
    --
    DBMS_OUTPUT.PUT_LINE('-----');
    DBMS_OUTPUT.PUT_LINE('Tabla del '||v_numero);
    DBMS_OUTPUT.PUT_LINE('-----');
    FOR i IN 1..12 LOOP
        DBMS_OUTPUT.PUT_LINE(v_numero||' X '||i||' = '||i*v_numero);
    END LOOP;

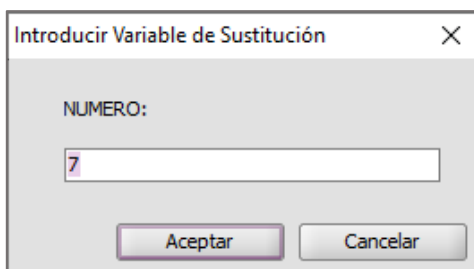
    EXCEPTION
    WHEN e_numero_invalido THEN
        DBMS_OUTPUT.PUT_LINE('Entrado invalida: '||v_entrada);
        DBMS_OUTPUT.PUT_LINE('Debe introducir un valor numérico.');
```

1. Para ejecutar nuestra excepción se declara como cualquier otra variable. En este caso se especifica que se ingreso un valor que no es del tipo numérico.

2. Para nuestro parámetro de entrada se utiliza el símbolo & y se le asigna a una variable para que incursione como el valor a usar.

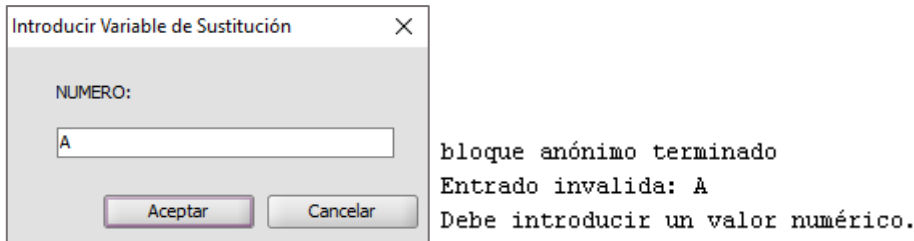
3. Finalmente se especifica la excepción utilizando **WHEN** y **THEN** respectivamente

Al ejecutar nuestro código nos aparecerá el recuadro que nos pide ingresar un parámetro, en este caso se ingresará el valor 7 y se ejecuta de manera exitosa



```
-----
Tabla del 7
-----
7 X 1 = 7
7 X 2 = 14
7 X 3 = 21
7 X 4 = 28
7 X 5 = 35
7 X 6 = 42
7 X 7 = 49
7 X 8 = 56
7 X 9 = 63
7 X 10 = 70
7 X 11 = 77
7 X 12 = 84
```

Sin embargo, si se ingresara otro tipo de dato como parametro de enetrada entonces nos marcara lo establecido en la excepción

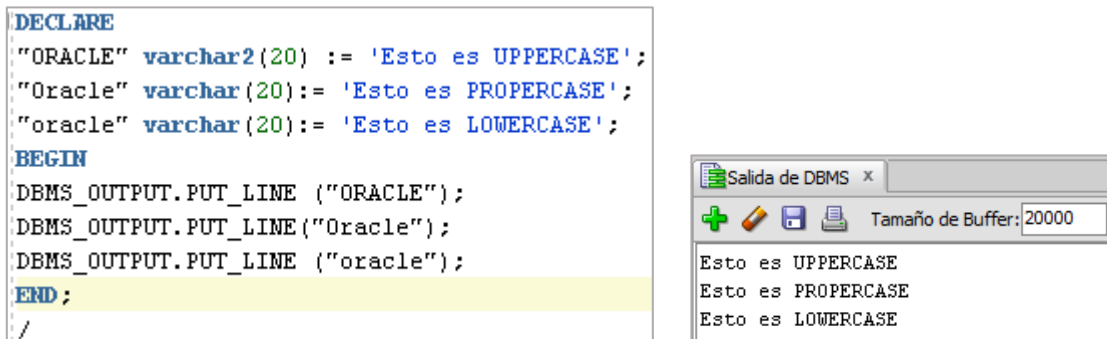


4. Escriba un bloque PL/SQL que muestre una palabra reservada que se puede usar como un identificador definido por el usuario.

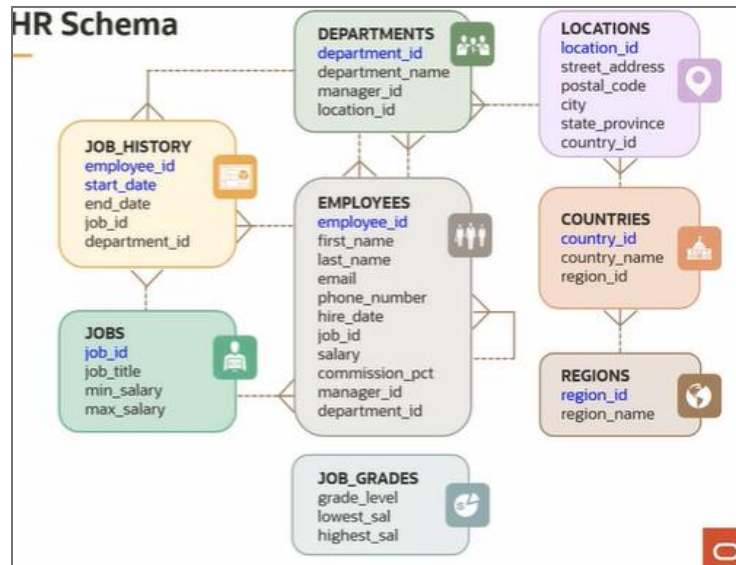
- Se usara como palabra reservada “Oracle”, es este caso para identificar las funciones **UPPERCASE**, **PROPERCASE** y **LOWERCASE**. En este caso cada variable muestra lo que cada función hace. Pero simplemente se usara como ejemplo de indicador definido.

- Una vez declaradas las variables en donde se asignan sus respectivos indicadores ('Esto es...')

- Se inicia el bloque donde a partir de un DBMS y las variables mostrara las asignaciones correspondientes.



Para estos ejercicios se usará el esquema HR.



5. **Crear un bloque anónimo que muestre los meses del año del 1-12 y seguido la cantidad de empleados que ingresaron en ese mes según su fecha de ingreso (hire_date)**

- Para este programa se declaran dos variables **V_MONTH** se encarga de los meses y **V_COUNT** la cuenta de los empleados por mes
- A la variable **V_MONTH** se le asigna el valor de 1 para que de cierta manera agrupe los meses uno a la vez
- Después mediante un **LOOP** se establece que la cuenta inicie en 1.. 12 tomando en cuenta la consulta mediante un **SELECT**, que lleva la cuenta por mes.
- Usando la condición WHERE dentro de la instrucción **SELECT** se especifica con **TO_CHAR** que solo se utilizara el mes(month).
- Finalmente con un DBMS declaramos las especificaciones de nuestra salida.

```

set serveroutput on
declare
    v_month number(2) := 1;
    v_count number(3);
begin
    for month in 1 .. 12
    loop
        select count(*) into v_count
        from employees
        where to_char(hire_date,'mm') = month;

        dbms_output.put_line( to_char(month,'00') || to_char(v_count,'999') );
    end loop;
end;
```

bloque anónimo terminado

01	4
02	2
03	2
04	0
05	3
06	3
07	1
08	1
09	2
10	1
11	2
12	0

Donde la primer columna indica el mes en el formato numérico y la segunda columna la cantidad de empleados que ingresaron en ese mes

6. Crear un bloque anónimo que muestre una lista de los employee_id que no están en uso, tomando como limite el employee_id con el humero mas alto.

```

DECLARE
    V_MIN_EMPID EMPLOYEES.EMPLOYEE_ID%TYPE;
    V_MAX_EMPID EMPLOYEES.EMPLOYEE_ID%TYPE;
    V_EMPID EMPLOYEES.EMPLOYEE_ID%TYPE;
BEGIN
    SELECT MIN(EMPLOYEE_ID),
           MAX(EMPLOYEE_ID)
    INTO V_MIN_EMPID,
         V_MAX_EMPID
    FROM EMPLOYEES;
    FOR EMPID IN V_MIN_EMPID + 1.. V_MAX_EMPID-1
    LOOP
        -- Bloques anidados
        BEGIN
            SELECT EMPLOYEE_ID INTO V_EMPID FROM EMPLOYEES WHERE EMPLOYEE_ID = EMPID;
        EXCEPTION
            WHEN NO_DATA_FOUND THEN
                DBMS_OUTPUT.PUT_LINE(EMPID);
        END;
    END LOOP;
END;
    
```

Mediante este bloque de código se especifica mediante que limites se tomaran los valores a mostrar

Y como salida del script obtenemos que:

```

bloque anónimo terminado
105
106
108
109
110
111
112
... 204
    
```

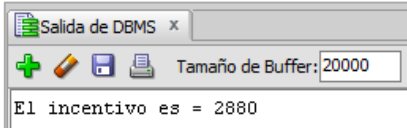
7. Escriba un bloque PL/SQL para calcular el incentivo (12%) de un empleado cuyo ID es 100.

Se tiene que

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY
100	Steven	King	SKING	515.123.4567	17/06/2011	AD_PRES	24000

- Se declara la variable *incentivo* del tipo *NUMBER (8,2)*, donde 8 es el número de dígitos y 2 el número de decimales. En este caso se usara 0.12
- Se inicia nuestro bloque donde se establece el incentivo de *salary * 012* en la variable *incentivo*
- Con la condicional *WHERE* indicamos que el ID debe ser igual a 100.
- Finalmente, con *DBMS* mostramos nuestro resultado. Que en este caso es 2800.


```
SET SERVEROUTPUT ON
DECLARE
incentivo NUMBER (8,2);
BEGIN
SELECT salary * 0.12 INTO incentivo
FROM employees
WHERE employee_id = 100;
DBMS_OUTPUT.PUT_LINE ('El incentivo es = ' || TO_CHAR(incentivo));
END;
/
```



8. Crear un bloque anónimo que muestre un listado del número de empleados que su salario es <, > e = que el salario promedio.

```
set serveroutput on
declare
v_avg_salary employees.salary%type;
v_mayor number(2);
v_igual number(2);
v_menor number(2);
begin
select avg(salary) into v_avg_salary
from employees;

select count(*) into v_mayor
from employees
where salary > v_avg_salary;

select count(*) into v_igual
from employees
where salary = v_avg_salary;

select count(*) into v_menor
from employees
where salary < v_avg_salary;

dbms_output.put_line('Mayor que el promedio: ' || v_mayor);
dbms_output.put_line('Igual al promedio: ' || v_igual);
dbms_output.put_line('Menor que el promedio: ' || v_menor);
end;
```

Consulta que almacena el promedio en la variable **v_avg_salary**

Serie de consultas que realizan la comparación de la variable **v_avg_salary** con el salario de los empleados

Y al ejecutar obtenemos:

```
bloque anónimo terminado
Mayor que el promedio: 7
Igual al promedio: 0
Menor que el promedio: 14
```

9. Crear un bloque anónimo que muestre la fecha en la que los empleados empezaran su nuevo trabajo, esto utilizando la tabla **JOB_HISTORY** donde se muestran las fechas de **END_DATE**, por lo tanto, la fecha que se mostrara será un día después de su **END_DATE**

- Para mostrar la ejecución de este ejercicio se utilizará al empleado 101

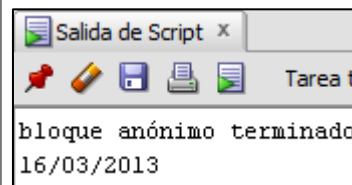
EMPLOYEE_ID	START_DATE	END_DATE	JOB_ID	DEPARTMENT_ID
102	13/01/2009	24/07/2014	IT_PROG	60
101	28/10/2009	15/03/2013	AC_MGR	10
201	17/02/2012	19/12/2015	MK_REP	20
114	24/03/2014	31/12/2015	ST_CLERK	50
122	01/01/2015	31/12/2015	ST_CLERK	50
176	01/01/2015	31/12/2015	SA_MAN	80
200	01/07/2010	31/12/2014	AC_ACCOUNT	90

- Se declaran dos variables **V_DATE** para la fecha y **V_EMPID** a la cual se le asignara un id de empleado (en este caso el 101)
- Después mediante una consulta con un **SELECT** se especifica que se toma el **MAX** de la **END_DATE**(fecha de finalización) de el empleado. El incorporar un **+1** significa que se agregara un valor a la fecha de end_date
- Con un IF se especifica que si no existe un valor de END_DATE para un empleado dado entonces tomara el valor de HIRE_DATE como resultado.

```
set serveroutput on
declare
  v_date date;
  v_empid employees.employee_id%type := 101;
begin
  select max(end_date) + 1 into v_date
  from job_history
  where employee_id = v_empid;

  if v_date is null then
    select hire_date into v_date
    from employees
    where employee_id = v_empid;
  end if;

  dbms_output.put_line(v_date);
end;
```



10. Crear un bloque anónimo que le aumente el sueldo al empleado 150 basándose en su JOB_HISTORY y en su comisión

1	FIRST_NAME	2	LAST_NAME	2	SALARY	2	COMMISSION_PCT
	Juan		Guarnizo		2500		0.2

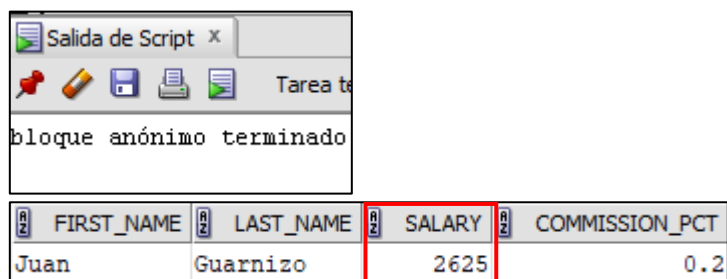
- Primero se declaran dos variables del tipo **NUMBER**, **V_COUNT** para la cuenta de **JOB_HISTORY** y **V_PER** que es la que almacenara el aumento
- Si el empleado no se encuentra en el historial de **JOB_HISTORY** entonces aumentara un 15% a su salario actual. Esto se realizará con la ayuda de un IF
- Después con un **UPDATE** aumentara el salario.
- Cada vez que se llama una consulta **SELECT** se debe condicionar con **WHERE** que el ,employee_id sea 150
-

```
DECLARE
  V_COUNT NUMBER(2);
  V_PER NUMBER(2);
BEGIN
  SELECT COUNT(*) INTO V_COUNT-- Realiza una cuenta
  FROM JOB_HISTORY
  WHERE EMPLOYEE_ID = 150;

  IF V_COUNT > 0 THEN
    V_PER := 15;
  ELSE
    SELECT NVL2(COMMISSION_PCT,5,10) INTO V_PER
    FROM EMPLOYEES_1
    WHERE EMPLOYEE_ID = 150;
  END IF;

  UPDATE EMPLOYEES_1
  SET SALARY = SALARY + SALARY * V_PER / 100
  WHERE EMPLOYEE ID = 150;
END;
```

Al ejecutar nuestro bloque anónimo, nos muestra la leyenda de que fue ejecutado y después al consultar los datos del empleado 150 podemos notar el aumento en el salario



Salida de Script x

Tarea te

bloque anónimo terminado

FIRST_NAME	LAST_NAME	SALARY	COMMISSION_PCT
Juan	Guarnizo	2625	0.2

11. Crear un bloque anónimo que muestre una lista con el nombre, nombre de su puesto y fecha en la que me empezaron su actual trabajo de todos los empleados, considerando que su actual trabajo sea un día después de su **END_DATE** , de lo contrario que muestre su fecha de ingreso. Utilizar un cursor para la obtención de datos y que una las tablas **EMPLOYEES** y **DEPARTMENTS**.

```
set serveroutput on
declare
  cursor empcur is
    select employee_id, first_name, job_title , hire_date
    from employees natural join jobs;

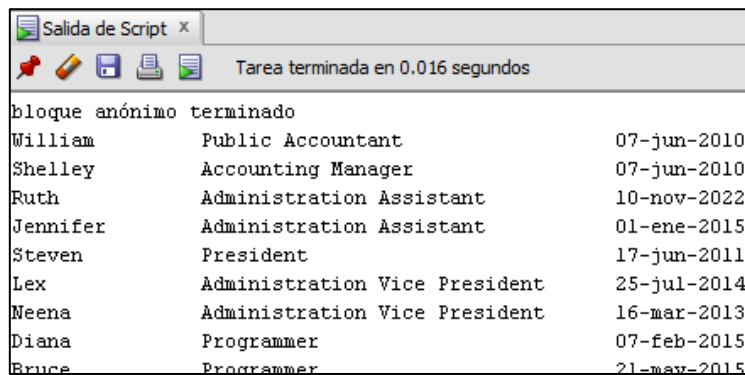
  v_date date;
begin
  for emprec in empcur
  loop
    -- find out most recent end_date in job_history
    select max(end_date) + 1 into v_date
    from job_history
    where employee_id = emprec.employee_id;

    if v_date is null then
      v_date := emprec.hire_date;
    end if;
    dbms_output.put_line( rpad(emprec.first_name,15) || ' ' || rpad(emprec.job_title,35)
    || to_char(v_date,'dd-mon-yyyy'));
  end loop;
end;
```

Cursor que contiene todos los datos que se solicitan de ambas tablas unidas mediante un JOIN

Utilizando el código del ejercicio anterior donde se agrega un día a end_date (en caso de existir)

Y como salida obtenemos una lista de los empleados con el respectivo nombre de su puesto y la fecha cuando empezaron su actual trabajo.



Bloque anónimo terminado		
William	Public Accountant	07-jun-2010
Shelley	Accounting Manager	07-jun-2010
Ruth	Administration Assistant	10-nov-2022
Jennifer	Administration Assistant	01-ene-2015
Steven	President	17-jun-2011
Lex	Administration Vice President	25-jul-2014
Neena	Administration Vice President	16-mar-2013
Diana	Programmer	07-feb-2015
Bruce	Programmer	21-may-2015

12. Escribir un bloque anónimo que muestre una lista de todos los departamentos seguidos de el empleado que gana mas de ese departamento. Utilizar un cursores

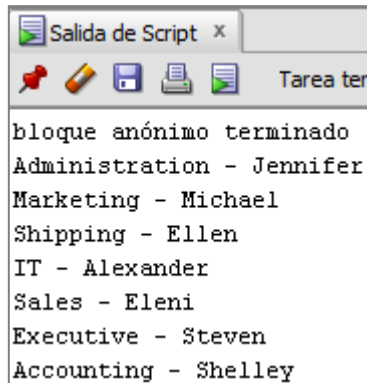
```
set serveroutput on
declare
  cursor deptcur is-- cursor que almacena los datos
    select department_id, department_name, max(salary) maxsalary
    from employees join departments using (department_id)
    group by department_id, department_name;
  v_name employees.first_name%type;
begin
  for deptrec in deptcur
  loop
    begin
      select first_name into v_name -- seleccciona solo el nombre del empleado
      from employees
      where department_id = deptrec.department_id and
            salary = deptrec.maxsalary;-- que tiene el salario mas alto

      dbms_output.put_line( deptrec.department_name || ' - ' || v_name);
    exception
      when too_many_rows then -- si dos empleos tienen el mismo salario y es el mas
        dbms_output.put_line( deptrec.department_name || ' - Mas que un empleado');
    end;
  end loop;
end;
```

Cursor que almacena los datos solicitados de la tabla **employees** y **departments** usando el **department_id** y agrupándolos (por uso de grupo de funciones

Se agrega una **excepción** en caso de que las filas devueltas sean mas de un empleado por departamento.

Como resultado de nuestro bloque anónimo obtenemos que:



```
Salida de Script x
Tarea ter
bloque anónimo terminado
Administration - Jennifer
Marketing - Michael
Shipping - Ellen
IT - Alexander
Sales - Eleni
Executive - Steven
Accounting - Shelley
```

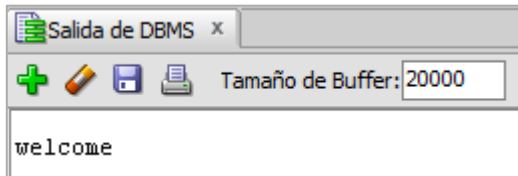
Existen otras maneras de realizar este código. Sin embargo como en este ejercicio se especifica que se unen cursores y excepciones, entonces por eso se usan de esa

13. Escriba un bloque PL/SQL para mostrar una referencia no sensible a mayúsculas y minúsculas a un identificador definido por el usuario entre comillas y sin comillas.

- Se declara la variable “HOLA” a la cual se le asigna el identificador ‘hola’, a pesar de contener cadena de caracteres similares lo que las distingue son las comillas simples, dobles y las mayúsculas y minúsculas.

- Finalmente se muestra el resultado con DBMS

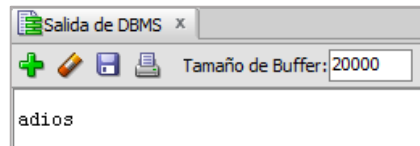
```
DECLARE
  "HOLA" varchar2(10) := 'hola'; -- identificador con comillas
BEGIN
  DBMS_Output.Put_Line("HOLA"); -- referencia con comillas dobles
END;
/
```



14. Escriba un bloque que haga referencia a un identificador sin uso de las comillas y la sensibilidad de mayúsculas o minúsculas

- De la misma manera que en el ejercicio 2 y 3 se declara nuestra variable entre comillas, esta vez en mayúsculas, sin embargo cuando se usa para mostrar la salida de DBMS no presenta ningún problema aun que se llamo en minúsculas. Esto sucede por el uso de comillas dobles.

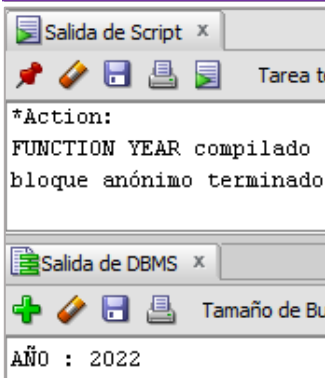
```
DECLARE
  "ADIOS" varchar2(10) := 'adios';
BEGIN
  DBMS_OUTPUT.PUT_LINE(adios);
END;
/
```



15. Escribir una función que reciba una fecha y devuelva el año correspondiente a esa fecha Se crea la función year(año) donde se declara que fecha es del tipo DATE y que el valor devuelto debe de ser del tipo NUMBER así como la variable v_year se declara del tipo NUMBER. Dentro de la instrucción BEGIN se asigna a v_year que el dato que devolverá será el año en el formato 'YYYY'. Finalmente se declara a n que será la variable que mostrara el año de SYSDATE.

```
CREATE OR REPLACE FUNCTION year(fecha DATE)
RETURN NUMBER
AS v_year NUMBER(4);
BEGIN
v_year := TO_NUMBER(TO_CHAR(fecha,'YYYY'));
RETURN v_year;
END year;

DECLARE
n NUMBER(4);
BEGIN
n := year(SYSDATE);--Se puede agregar en el formato 'DD/MM/YYYY'
DBMS_OUTPUT.PUT_LINE('AÑO : '|| n);
END;
```



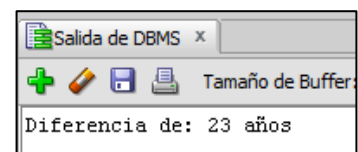
16. Desarrolla una función que devuelva la diferencia de años que hay entre dos fechas

En este caso se crea la función como *dif_fechas* donde se pedirán dos fechas *fecha1* y *fecha2*. Dentro de BEGIN se desarrolla la función con la función MONTHS_BETWEEN que obtiene la diferencia de meses entre dos fechas.

```
CREATE OR REPLACE FUNCTION dif_fechas(
fechal DATE, fecha2 DATE)
RETURN NUMBER AS v_dif_fechas NUMBER(10);
BEGIN
v_dif_fechas := ABS(TRUNC(MONTHS_BETWEEN(fecha2,fechal)/12));
RETURN v_dif_fechas;
END dif_fechas;
```

Se prueba la función donde se tomaran como parámetros sysdate y una fecha con YYYY en 1998, por lo tanto mostrara que la diferencia es de 23, esto tomando como parámetro que MONTHS_BETWEEN obtiene la diferencia de meses entre una fecha y otra, se divide entre 12 por cada año y se TRUNC cuando la diferencia de meses excede el valor.

```
DECLARE
dif NUMBER(10);
BEGIN
dif := dif_fechas(SYSDATE,'21/12/1998');
DBMS_OUTPUT.PUT_LINE('Diferencia de: '||dif||' años');
END;
```



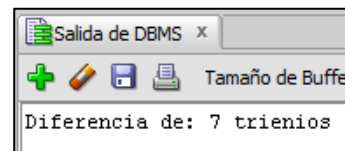
17. Escribe una función que muestre los trienios (bloques de tres años) que hay entre dos fechas. Usa de apoyo la función anterior

Se crea la función *trienios* que pide 2 datos *fecha1* y *fecha2*. Después declara que la variable *v_trienios* sea del tipo NUMBER. Posteriormente dentro de BEGIN se desarrolla la función donde se llama a la función creada anteriormente *dif_fechas*

```
CREATE OR REPLACE FUNCTION trienios(  
  fechal DATE, fecha2 DATE)  
RETURN NUMBER AS  
  v_trienios NUMBER(10);  
BEGIN  
  v_trienios := TRUNC(dif_fechas(fechal, fecha2)/3);  
RETURN v_trienios;  
END;
```

Se prueba la función. En este caso la diferencia de años entre las fechas es de 21, por lo tanto al ejecutarse la función realiza $21/3$ y por eso devuelve 7.

```
DECLARE  
b NUMBER(10);  
BEGIN  
b:=trienios ('01/01/2001','01/01/2022');  
DBMS_OUTPUT.PUT_LINE('Diferencia de: '||b||' trienios');  
END;
```



Procedimientos y funciones

18. Escribir una función que devuelva caracteres alfabéticos sustituyendo cualquier otro carácter por espacios en blanco.

En este caso se ayuda de ASCII

```
CREATE OR REPLACE FUNCTION sustitucion(cadena VARCHAR2)  
  RETURN VARCHAR2  
AS  
  nueva_cadena VARCHAR2(50);  
  caracter      CHAR;  
BEGIN  
  FOR i IN 1..length(cadena)  
  LOOP  
    caracter := SUBSTR(cadena,i,1);  
    IF (ascii(caracter) NOT BETWEEN 65 AND 90) AND (ascii(caracter) NOT BETWEEN 97 AND 122) THEN  
      caracter := ' ';  
    END IF;  
    nueva_cadena := nueva_cadena || caracter;  
  END LOOP;  
  RETURN nueva_cadena;  
END sustitucion;
```

Prueba de la función donde la cadena esta con distintos caracteres y solo muestra los alfabéticos


```
DECLARE
    x VARCHAR2(50);
BEGIN
    x:= sustitucion('Esta4es6una%cadena2#');
    dbms_output.put_line('Resultado: '||x);
END;
```

Salida de Script x

Tarea terminada

Resultado: Esta es una cadena

19. Mostrar todos los PROCEDURES y FUNCTIONS que existen en la DB

```
select object_name, object_type, status
from user_objects
where object_type in ('PROCEDURE','FUNCTION');
```

OBJECT_NAME	OBJECT_TYPE	STATUS
SUSTITUCION	FUNCTION	VALID
TODAY_IS	PROCEDURE	VALID
DIF_FECHAS	FUNCTION	INVALID
RANDOM_STRINGS	FUNCTION	VALID
MORE_NUMBERS	FUNCTION	VALID
MOSTRAR_REGION	PROCEDURE	VALID
TRIENIOS	FUNCTION	INVALID
YEAR	FUNCTION	VALID

Cursores

20. Crear un cursor que muestre el nombre de los empleados

```
DECLARE
CURSOR CONSULTA IS
    SELECT first_name, last_name
    FROM employees
    ORDER BY first_name;
BEGIN
    FOR FILA IN CONSULTA LOOP
        DBMS_OUTPUT.PUT_LINE(FILA.first_name||' '||FILA.last_name);
    END LOOP;
END;
```

Salida de DBMS x

Tarea terminada

Alexander Hunold
 Bruce Ernst
 Curtis Davies
 Diana Lorentz
 Eleni Zlotkey
 Ellen Abel

En este caso se crea el cursor 'CONSULTA' que se llama después en un LOOP que muestra una lista de los empleados según las condiciones del cursor.

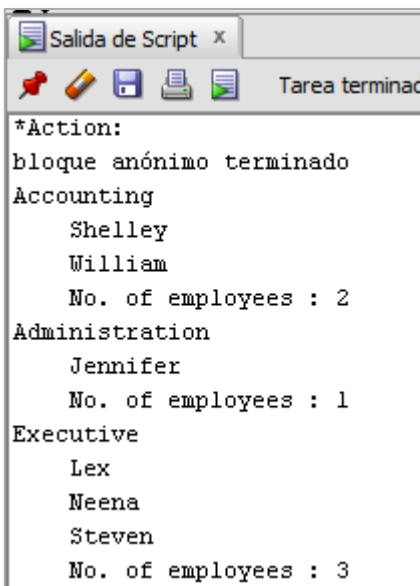
21. Crear un cursor explicito que muestre una lista de los departamentos con todos sus empleados y la cantidad de empleados que hay en el

```
DECLARE
  CURSOR depto
  IS
    SELECT department_id,
           department_name
    FROM departments
   WHERE department_id IN
      ( SELECT department_id FROM employees
      )
   ORDER BY 2;
  CURSOR empcur (p_deptid NUMBER)
  IS
    SELECT first_name FROM employees WHERE department_id = p_deptid ORDER BY 1;
  v_count NUMBER(3);
BEGIN
  FOR deptrec IN depto
  LOOP
    dbms_output.put_line(deptrec.department_name);
    v_count :=0;
    FOR emprec IN empcur(deptrec.department_id)
    LOOP
      dbms_output.put_line('      ' || emprec.first_name);
      v_count := v_count + 1;
    END LOOP;
    dbms_output.put_line('      ' || 'No. de empleados: ' || v_count);
  END LOOP;
END;
```

Se crea el **cursor depto** que después dentro de un **select** llama a la tabla **departments**.

Con el cursor **empcur** se establece a los empleados

Se crea un loop para el conteo de empleados por departamento. Y también un loop para que arroje la lista de empleados por departamento



```
*Action:
bloque anónimo terminado
Accounting
  Shelley
  William
  No. of employees : 2
Administration
  Jennifer
  No. of employees : 1
Executive
  Lex
  Neena
  Steven
  No. of employees : 3
```

En la salida se muestra una lista de cada nombre de departamento con sus respectivos empleados y número de empleados. El ORDER BY 2 se declaró para que existiera un orden de columnas.

22. Crear un cursor explícito que muestre una lista de los empleados según su año de ingreso

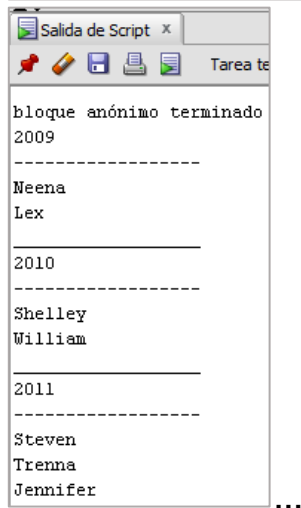
En este caso nuestro cursor es C_EMP que a partir del parámetro ANIO del tipo NUMBER obtendrá los datos especificados en el SELECT donde además incluye la condicional con WHERE. Con el WHERE lo que se pide es que se recopile el año de la fecha de ingreso de los empleados

```
declare
cursor c_emp (anio number) is
select first_name
from employees
where to_char(hire_date,'YYYY')= anio;
begin
for year in 2009..2022
loop
  dbms_output.put_line(year);
  dbms_output.put_line('-----');
  for emp in c_emp(year)
  loop
    dbms_output.put_line( emp.first_name);
  end loop;

  dbms_output.put_line(' ');
end loop;
end;
```

Declaración del cursor, donde se obtendrá el año de a fecha de ingreso

LOOP que recorrerá año por año de la fecha de ingreso e ira mostrando subsecuentemente



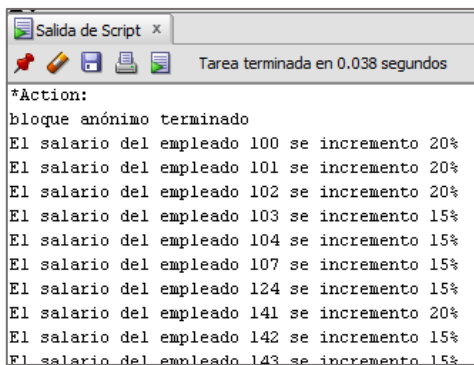
```
Salida de Script x
Tarea te
bloque anónimo terminado
2009
-----
Neena
Lex
-----
2010
-----
Shelley
William
-----
2011
-----
Steven
Tenna
Jennifer
...
```

23. Crear un cursor explícito que aumente un porcentaje del salario según los años laborados.

Si el empleado lleva más de 10 años laborando aumenta el 20%, si lleva más de 5 aumenta 15% y si tiene menos entonces aumenta el 10%

```
DECLARE
CURSOR empcur
IS
    SELECT employee_id,
           TRUNC(months_between(sysdate,hire_date) / 12) exp --años que hay entre la fecha de ingreso a la actual
    FROM employees;
v_increment NUMBER(2);
BEGIN
FOR emprec IN empcur
LOOP
    -- incremento basado en años laborados(exp)
    v_increment :=
    CASE
    WHEN emprec.exp > 10 THEN
        20
    WHEN emprec.exp > 5 THEN
        15
    ELSE
        10
    END ;
    update employees
    SET salary = salary + (salary * v
    WHERE employee_id = emprec.employee_id,
    dbms_output.put_line('El salario del empleado ' ||emprec.employee_id||' se incremento '||v_increment || '%');
END LOOP;
END;
```

CASE para especificar cada caso, si no ejecuta uno entonces ejecutara el siguiente.



Salida de Script x

Tarea terminada en 0.038 segundos

*Action:

bloque anónimo terminado

El salario del empleado 100 se incremento 20%

El salario del empleado 101 se incremento 20%

El salario del empleado 102 se incremento 20%

El salario del empleado 103 se incremento 15%

El salario del empleado 104 se incremento 15%

El salario del empleado 107 se incremento 15%

El salario del empleado 124 se incremento 15%

El salario del empleado 141 se incremento 20%

El salario del empleado 142 se incremento 15%

El salario del empleado 143 se incremento 15%

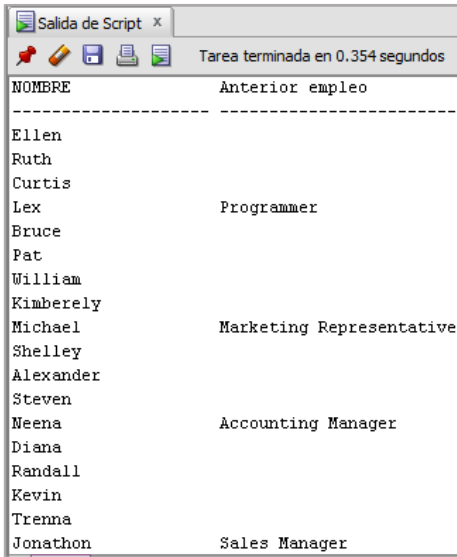
24. Crea una función que contenga un cursor explícito que muestre una lista de todos los empleados y su empleo anterior

```
create or replace function get_job_history(empid number)
return varchar2
is
    cursor jobscur is
        select job_title from jobs
        where job_id in ( select job_id from job_history where employee_id = empid);
    v_jobs varchar2(200) := '';
begin
    for jobrec in jobscur
    loop
        v_jobs := v_jobs || jobrec.job_title || ',';
    end loop;

    return rtrim(v_jobs,',');
end;
```

Se crea el cursor *jobscur* donde se realiza una subconsulta con select para tomar los datos de la tabla *Jobs* y *job_history*.

Se inicia un loop donde a la variable *v_jobs* se le asigna la concatenación. Con *return* se devuelve la variable anterior sin espacios(*rtrim*)



NOMBRE	Anterior empleo
Ellen	
Ruth	
Curtis	
Lex	Programmer
Bruce	
Pat	
William	
Kimberely	
Michael	Marketing Representative
Shelley	
Alexander	
Steven	
Neena	Accounting Manager
Diana	
Randall	
Kevin	
Trena	
Jonathon	Sales Manager

25. Se requiere ver el salario más bajo y el más alto de los departamentos. Sin embargo, algunos departamentos solo quieren ver el más bajo, otros el más alto y algunos más quieren ver ambos.

Crear una función que pida 3 parámetros de entrada, numero de departamento, nombre de departamento y una opción de entre 3 letras (igual como parámetro)

B: Que muestre salario más bajo y salario más alto

H: Que muestre el salario más alto

L: Que muestre el salario más bajo

- Primero creamos nuestra función que pide 3 parámetros, en el caso del parámetro de la letra entonces marca por **DEFAULT** la letra B que muestra el mas bajo y el más alto.
- Después que devuelva el salario bajo y el alto.

```
CREATE OR REPLACE
FUNCTION low_high_salary(
    p_dept IN departments.department_id%TYPE,
    p_job_id IN jobs.job_id%TYPE,
    p_sal IN CHAR DEFAULT 'B' -- L: Mas Bajo; H: Mas Alto, B: Ambos
)
RETURN VARCHAR2
IS
    CURSOR c_high_low-- cursor que muestra ambos salarios
    IS
        SELECT MIN(salary) AS bajo,--salario bajo
               MAX(salary) AS alto -- salario alto
        FROM employees
        WHERE department_id = p_dept
        AND job_id = p_job_id;

    v_high_low_rec c_high_low%ROWTYPE;

    e_wrong_entry EXCEPTION;-- declaracion de excepcion
```

- Con **FETCH** se asocia el cursor con la variable
- Y con **IF** llama a la excepción
- Y con **CASE** establecemos que es lo que se mostrara según la letra que se ingrese como parámetro de entrada.
-

```


BEGIN
    OPEN c_high_low;
    FETCH c_high_low INTO v_high_low_rec;
    CLOSE c_high_low;
    --
    IF v_high_low_rec.bajo IS NULL THEN
        RAISE e_wrong_entry;
    END IF;
    --
    CASE
    WHEN UPPER(p_sal) = 'B' THEN-- se garga UPPER por si la opcion se inresa en minuscula
        RETURN 'Salario Mas Alto: '||v_high_low_rec.alto|| ', Salario Mas Bajo: '||v_high_low_rec.bajo;
    WHEN UPPER(p_sal) = 'L' THEN
        RETURN 'Salario Mas Bajo: '||v_high_low_rec.bajo;
    WHEN UPPER(p_sal) = 'H' THEN
        RETURN 'Salario Mas Alto: '||v_high_low_rec.alto;
    ELSE
        RETURN 'Valores Validos para p_sal: B,L,H.';-- por si la letra ingresada es incorrecta
    END CASE;
    --
EXCEPTION
    WHEN e_wrong_entry THEN
        RETURN 'Debe introducir un departamento/empleo valido.';-- por si el departamento ingresaod es in
    END;
/
    
```

- Finalmente para visualizar la ejecución de nuestra función entonces creamos una consulta **SELECT** donde ingresamos los 3 parámetros

```

SELECT low_high_salary(50, 'ST_CLERK', 'b')
FROM dual;
    
```

- Y como salida obtenemos que

Resultado de la Consulta x	
 Todas las Filas Recuperadas: 1 en 0.016 segundos	
LOW_HIGH_SALARY(50,'ST_CLERK','B')	
1	Salario Mas Alto: 3500, Salario Mas Bajo: 2500

REF CURSOR

26. Crear un procedimiento que muestre una lista de todos los empleados seguido del nombre de su puesto. Utilizar un REFCURSOR para los empleados y recuerda que los datos están almacenados en 2 tablas distintas: **EMPLOYEES** y **DEPARTMENTS**.

Para este caso:

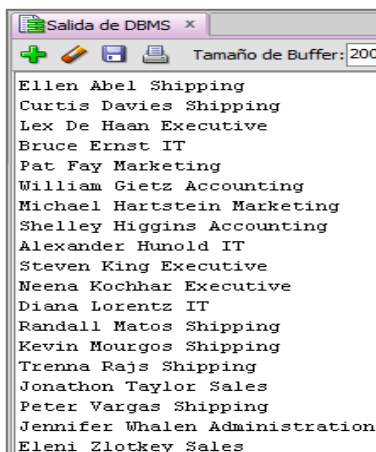
1. Se crea el procedimiento VER_EMPLEADOS

2. Como parámetro se asigna un REFCURSOR
3. Nuestro REFCURSOR a través de una consulta SELECT obtiene los datos que requerimos. Une la tabla EMPLOYEES y DEPARTMENTS con un JOIN y usando el department_id como elemento de unión.

```
create or replace procedure ver_empleados (emp out sys_refcursor)
as
begin
open emp for
select last_name, first_name, department_name
from employees
join departments using (department_id)
order by last name;
end ver_empleados;
```

4. Ahora para ver el resultado de nuestro procedimiento, lo visualizamos mediante un bloque anónimo:
Dentro de nuestras variables se incorpora el **REFCURSOR** ya que mediante un **FETCH** se especificará que filas se quieren obtener.

```
declare
emp sys_refcursor;
nombre varchar2(50);
apellido varchar2(50);
departamento varchar2(50);
begin
ver_empleados(emp);
loop
fetch emp into apellido, nombre, departamento;
exit when emp%notfound;
dbms_output.put_line (nombre || ' ' || apellido || ' ' || departamento);
end loop;
close emp;
end;
```



```
Salida de DBMS
+-----+
 Ellen Abel Shipping
Curtis Davies Shipping
Lex De Haan Executive
Bruce Ernst IT
Pat Fay Marketing
William Gietz Accounting
Michael Hartstein Marketing
Shelley Higgins Accounting
Alexander Humold IT
Steven King Executive
Neena Kochhar Executive
Diana Lorentz IT
Randall Matos Shipping
Kevin Mourgos Shipping
Trenna Rajs Shipping
Jonathon Taylor Sales
Peter Vargas Shipping
Jennifer Whalen Administration
Eleni Zlotkey Sales
```

Como resultado obtenemos la lista de los empleados con su puesto.

27. Crear un bloque anónimo que a través de un REF CURSOR muestre la lista de todos los empleados y que además les asigne un numero aleatorio del 20 al 50.

1. Declaración de variables, types y de los respectivos REF CURSOR. En este caso TYP_REF_CUR es nuestro REF CURSOR. También se crean variables del tipo TYPE.

```
SET SERVEROUTPUT ON
DECLARE
--TYPE
TYPE typ_ref_cur
IS
REF
CURSOR; --TIPO REF CURSOR
TYPE typ_rec
IS
RECORD
(
id VARCHAR2(10),
name VARCHAR2(50),
col_3 VARCHAR2(20),
col_4 VARCHAR2(10) );
v_rec typ_rec;--VARIABLES Y TYPE RECORDS
v_ref_cur typ_ref_cur; ---VARIABLE REF CURSOR
v_input NUMBER(1)v_query VARCHAR2(1000);
```

Se especifica que nuestro TYPE TYP_REF_CUR sea el REF CURSOR

Ahora dentro de nuestro TYPE se agregan las variables que se utilizaran para asociar a los empleados con los requisitos de la consulta.

```
BEGIN
v_input := 3;
CASE
WHEN v_input > 2 THEN
v_query := q'[SELECT
employee_id,
first_name||' '||last_name,
TRUNC(DBMS_RANDOM.value(20,50)),
'M'
FROM employees]';
WHEN v_input < 2 THEN
v_query := q'[SELECT
job_id,
job_title,
min_salary,
max_salary
FROM jobs]';
ELSE
v_query := q'[SELECT
d.department_id,
d.department_name,
l.city,
(
SELECT COUNT(*)
FROM employees
WHERE department_id = d.department_id
)
FROM departments d, locations l
WHERE d.location_id = l.location_id]';
END CASE;
```

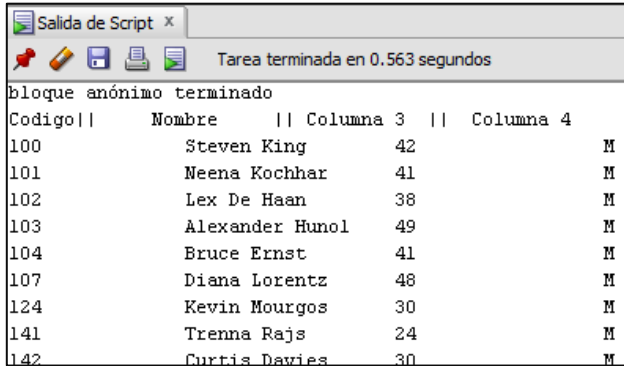
Ahora solo se cree un CASE para que cada vez que se desee consultar o ver el listado de los empleados entonces el CASE ejecute la instrucción.

Finalmente abrimos nuestro **REF CURSOR** para que tenga salida en nuestro DBMS y luego como cualquier otro cursor utilizamos un **FETCH** para especificar que filas queremos que regrese


```

OPEN v_ref_cur FOR v_query;
DEMS_OUTPUT.PUT_LINE ( 'Codigo||      Nombre      || Columna 3 || Columna 4 ' );
LOOP
    FETCH v_ref_cur INTO v_rec;
    EXIT
WHEN v_ref_cur%NOTFOUND;
DEMS_OUTPUT.PUT_LINE(RPAD(v_rec.id,10,' ')||'      '||RPAD(v_rec.name,15,' ')||'      '||RPAD(v_rec.col_3,15,' ')||'      '||v_rec.col_4);
END LOOP;
CLOSE v_ref_cur;
END;
    
```

Y como resultado de la consulta obtenemos lo siguiente



Codigo	Nombre	Columna 3	Columna 4
100	Steven King	42	M
101	Neena Kochhar	41	M
102	Lex De Haan	38	M
103	Alexander Hunol	49	M
104	Bruce Ernst	41	M
107	Diana Lorentz	48	M
124	Kevin Mourgos	30	M
141	Trenna Rajs	24	M
142	Curtis Davies	30	M

Packages

28. Crear un paquete que de de alta, baja y modifique a la tabla EMPLOYEES_1.

Especificaciones de la creación del paquete:

- Se deben controlar el manejo de excepciones
- El número de empleado será el siguiente al último.
- El departamento deberá de ser el mismo que el de su jefe
- El salario será el salario medio de su departamento
- La fecha de alta será la de el sistema (SYSDATE)

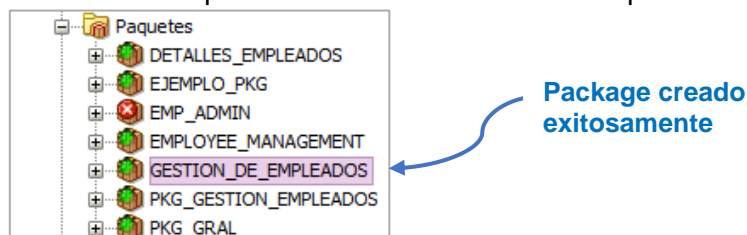
Función **BAJA_EMP** deberá recibir un parámetro(employee_id) y debe de devolver una variable boolean si el empleado fue borrado o no

Procedure **MOD_EMP** deberá recibir dos parámetros: employee_id, nuevo_depto. Mostrar si se modifico al empleado y además verificar si el empleado y el departamento existe.

1. Para la creación de nuestro *package* primero es importante crear la estructura. Entonces declaramos las funciones y procedimientos que vamos a crear dentro de el mismo.

Se tiene que verificar que la estructura se haya compilado correctamente.

Para eso en la parte de conexiones validamos que este creada



Como parte de la estructura tenemos el siguiente código:

```
CREATE OR REPLACE
PACKAGE GESTION_DE_EMPLEADOS
AS
    PROCEDURE alta_emp(
        nombre employees_1.first_name%TYPE,
        apellido employees_1.last_name%TYPE,
        correo employees_1.email%TYPE,
        telefono employees_1.phone_number%TYPE,
        t_id employees_1.job_id%TYPE,
        jefe employees_1.manager_id%TYPE );
    FUNCTION BAJA_EMP(
        EMP_NUM NUMBER)
        RETURN BOOLEAN;
    PROCEDURE mod_emp(
        emp_num NUMBER,
        dept_num NUMBER);
END GESTION_DE_EMPLEADOS;
```

Procedimiento que da de alta a los empleados y sus respectivos parámetros.

Función que da de baja a los empleados

Procedimiento que modifica el departamento de un empleado dado.

2. Después creamos el cuerpo del paquete de la siguiente manera

```
CREATE OR REPLACE
PACKAGE BODY GESTION_DE_EMPLEADOS
AS
```

Seguido incorporamos todo el código de los paquetes y funciones previamente indicados

3. Primero el paquete que agrega empleados, lo llamamos **ALTA_EMP**
Este es el código mas largo de este body package, debido a la cantidad de parámetros.

```
PROCEDURE alta_emp(
    nombre employees_1.first_name%TYPE,
    apellido employees_1.last_name%TYPE,
    correo employees_1.email%TYPE,
    telefono employees_1.phone_number%TYPE,
    t_id employees_1.job_id%TYPE,
    jefe employees_1.manager_id%TYPE )
IS
    emp_id employees_1.employee_id%TYPE;
    avg_sal employees_1.salary%TYPE;
    dept_jefe employees_1.department_id%TYPE;
    comi employees_1.commission_pct%TYPE;
BEGIN
    SELECT DISTINCT department_id
    INTO dept_jefe --Departamento del emp
    FROM employees_1
    WHERE employee_id = jefe;
    SELECT MAX(employee_id) INTO emp_id FROM employees_1;
    --
    IF lower(t_id) = lower('SA_MAN')-- Comision solo si es Salesman
    THEN
        comi:= 0;
    END IF;
    --
    SELECT AVG(NVL(salary,0))
    INTO avg_sal -- Salario proporcional al promedio del departamento
    FROM employees_1
    WHERE department_id = dept_jefe;
```

```

INSERT
INTO employees_1
    (employee_id,first_name,last_name,email,phone_number,hire_date,job_id,salary,commission_pct,manager_id,department_id )
VALUES
    (emp_id + 1, nombre,apellido,correo,telefono,sysdate,t_id,avg_sal,comi,jefe,dept_jefe);
DBMS_OUTPUT.PUT_LINE('Empleado insertado con éxito!');
EXCEPTION
WHEN NO_DATA_FOUND THEN -- Manejo de excepciones
    DBMS_OUTPUT.PUT_LINE('El manager_id no existe');
END alta_emp;
    
```

4. Con el insert indicamos que datos queremos use se agreguen a nuestra tabla según las variables que declaramos y los requerimientos. En este caso se solicitaba que se agregara el id de manera consecutiva y que el id del departamento fuera el mismo que el de el **manager_id**
5. Creamos la función que da de baja a los empleados según su ID

```

FUNCTION BAJA_EMP
    (emp_num NUMBER)
    RETURN BOOLEAN
IS
BEGIN
    DELETE FROM employees_1 WHERE employee_id = emp_num;
    IF sql%rowcount < 1 THEN --cuenta de las filas añadidas
        RETURN false;
    ELSE
        RETURN true;
    END IF;
END BAJA_EMP;
    
```

6. Procedimiento que modifica el departamento de los empleados según su ID y un nuevo número de departamento dado

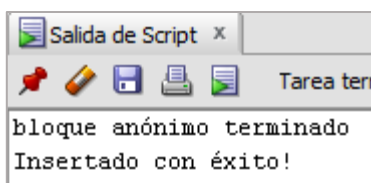
```

PROCEDURE mod_emp(
    emp_num NUMBER,
    dept_num NUMBER)
IS
    modi NUMBER(5);
    noemp EXCEPTION;
BEGIN
    SELECT department_id
    INTO modi
    FROM departments
    WHERE department_id = dept_num;
    UPDATE employees_1 SET department_id = modi WHERE employee_id = emp_num;
    IF sql%rowcount < 1 THEN
        raise noemp;
    END IF;
    dbms_output.put_line('Empleado actualizado con éxito!');
EXCEPTION --Manejo de excepciones
WHEN NO_DATA_FOUND THEN
    dbms_output.put_line('El departamento ingresado no existe');
WHEN NOEMP THEN
    dbms_output.put_line('El empleado ingresado no existe');
END mod_emp;
END GESTION_DE_EMPLEADOS;
    
```

7. Finalmente creamos los bloques anónimos para mandar a llamar a nuestros procedimientos y funciones previamente creados.

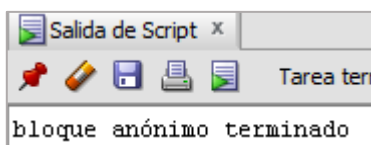
Bloque anónimo para llamar a nuestro procedimiento de inserción de filas. Solo se ingresan los datos que se indican como parámetros y se asignan directamente a variables creadas.

```
DECLARE
  NOMBRE   VARCHAR2(20);
  APELLIDO VARCHAR2(25);
  CORREO   VARCHAR2(25);
  TELEFONO VARCHAR2(20);
  T_ID     VARCHAR2(10);
  JEFE     NUMBER;
BEGIN
  NOMBRE   := 'EMPLEADO';
  APELLIDO := 'NUEVO';
  CORREO   := 'ENUEVO';
  TELEFONO := '12345678910';
  T_ID     := 'SA_REP';
  Jefe     := 100;
  GESTION_DE_EMPLEADOS.ALTA_EMP( NOMBRE,APELLIDO,CORREO, TELEFONO, T_ID, JEFE);
END;
```



Bloque anónimo para la función de baja de empleados.

```
DECLARE
  EMP_NUM  NUMBER;
  v_Return BOOLEAN;
BEGIN
  EMP_NUM  := 150;
  v_Return := GESTION_DE_EMPLEADOS.BAJA_EMP( EMP_NUM);
END;
```



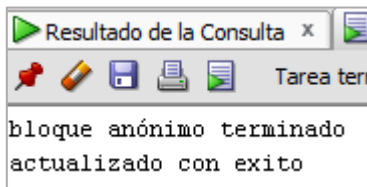
Bloque anónimo para el procedimiento de modificar empleados

```
DECLARE
    EMP_NUM  NUMBER;
    DEPT_NUM  NUMBER;
BEGIN
    EMP_NUM  := 208;
    DEPT_NUM := 50;
    GESTION_DE_EMPLEADOS.MOD_EMP( EMP_NUM, DEPT_NUM );
END;
```

Para corroborar que se modificó, tomaremos a el empleado recién creado, del cual su departamento es el 90:

EMPLOYEE_ID	FIRST_NAME	DEPARTMENT_ID
208	EMPLEADO	90

Ejecutado, obtenemos que:



Y consultamos de nuevo la tabla y notaremos que el valor de department_id fue cambiado.

EMPLOYEE_ID	FIRST_NAME	DEPARTMENT_ID
208	EMPLEADO	50

Ejemplo Package 2.

29. Especificaciones del paquete:

- Crear un procedimiento que muestre en una fila los datos de un empleado. El id de empleado debe de ser un parámetro de entrada.
 - Los datos mostrados en la fila de deben ser en el siguiente orden: employee_id, last_name, job_id, salary
 - Crear otro procedimiento que muestre el salario de un empleado. El id de empleado debe de ser un parámetro
1. Primero creamos la estructura de nuestro paquete, donde únicamente se contiene en parámetro de entrada. El parámetro tiene el mismo tipo de dato que employee_id gracias a %TYPE.

```
create or replace
package package_adv
is
procedure e_salary (e_id employees_1.employee_id%type);
procedure e_all (e_eid employees_1.employee_id%type);
end package adv;
```

- Después creamos el cuerpo de nuestro paquete, donde agregamos ambos procedimientos.
Primero agregamos el procedimiento que muestra el salario de los empleados

```
create or replace
PACKAGE BODY package_adv
is
    e_sal    employees_1.salary%type;
    e_eid    employees_1.employee_id%type;
    e_ln     employees_1.last_name%type;
    e_jid    employees_1.job_id%type;
    --
    PROCEDURE e_salary (e_id EMPLOYEES_1.EMPLOYEE_ID%TYPE)
    IS
    BEGIN
        SELECT salary
        into e_sal
        FROM employees_1
        WHERE employee_id = e_id;

        DBMS_OUTPUT.put_line (e_sal);
    END e_salary;
```

- Y después agregamos el procedimiento que muestra en una fila los datos del empleado.
En este caso se deben tomar en cuenta los datos que queremos que se muestren de nuestros empleados, así como la manera en que queremos como se muestren

```
PROCEDURE e_all (e_eeid EMPLOYEES_1.EMPLOYEE_ID%TYPE)
IS
begin
    SELECT employee_id, last_name, job_id, salary
    INTO e_eid,
        e_ln,
        e_jid,
        e_sal
    FROM employees_1
    WHERE employee_id = e_eeid;

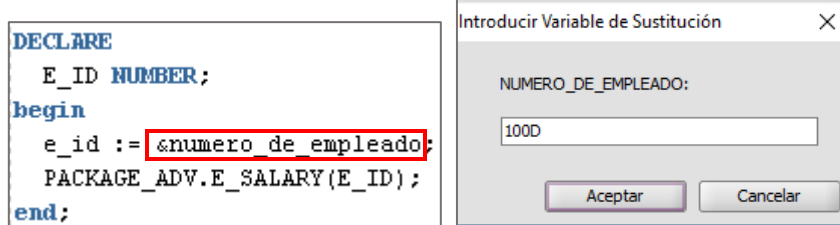
    dbms_output.put_line (e_eid || ' '
                          || e_ln || ' '
                          || e_jid || ' '
                          || e_sal);
end e_all;
end package_adv;
```

- Después mediante dos bloques anónimos mostramos qué es lo que realizan nuestros procedimientos.

```
DECLARE
    E_EEID NUMBER;
begin
    e_eeid := 100;
    PACKAGE_ADV.E_ALL(E_EEID);
end;
```

```
100 King AD_PRES 26460
```

Para el segundo bloque anónimo que muestra el procedure **e_salary** se agrego que la variable previamente creada se le asignara un parámetro de entrada ingresado por el usuario.



Y como salida, simplemente se muestra el salario del empleado 100

26460

30. Crear un trigger que se dispare cada vez que se ingrese un registro en la tabla "LIBROS".

- El trigger debe de ingresar en la tabla CONTROL el nombre del usuario, la fecha y la hora en la cual se realizó la instrucción INSERT en la tabla LIBROS.
- Primero se crean las dos tablas a partir de las siguientes descripciones:

Nombre	Nulo	Tipo
-----	----	-----
CODIGO		NUMBER(6)
TITULO		VARCHAR2(40)
AUTOR		VARCHAR2(30)
PRECIO		NUMBER(6,2)

Nombre	Nulo	Tipo
-----	----	-----
USUARIO		VARCHAR2(30)
FECHA		DATE

- Después continuamos con la creación del TRIGGER el cual se encargara de almacenar todas las veces que la tabla LIBROS sufre algún cambio:

```
create or replace trigger tr_ingresar_libros
before insert
on libros
begin
insert into control
values (user, sysdate);
end tr_ingresar_libros;
```

- Utilizamos un ALTER SESSION para modificar la manera en la que se almacena la hora en la tabla CONTROL.

```
alter session set nls_date_format = 'DD/MM/YYYY HH24:MI';
```

- Para comprobar que nuestro TRIGGER se creo de manera satisfactoria entonces agregamos datos a la tabla LIBROS

```
insert into libros
values (100, 'Uno', 'Richard Bach', 25);
insert into libros
values (100, 'Matematica estas ahi', 'Paenza', 12);
insert into libros
values (100, 'El aleph', 'Borges', 42);
```

- Entonces al momento de consultar la tabla CONTROL obtenemos que:

R	USUARIO	R	FECHA
ORA21		12/01/2023	12:37
ORA21		12/01/2023	12:39
ORA21		12/01/2023	12:39

Se guarda el registro de todas las veces que se insertaron (INSERT) valores en la tabla CONTROL, que en esta ocasión fueron 3 veces. Y con eso se concluye que nuestro TRIGGER fue ejecutado con éxito.

31. Crear un TRIGGER que se dispare una vez por cada registro que se ingrese en la tabla OFERTAS. El trigger debe de ingresar en la tabla CONTROL, el nombre de usuario, fecha y hora en que se realizó un INSERT en la tabla OFERTAS.

- Se utilizan las siguientes tablas:

describe LIBROS	
Nombre	Nulo Tipo

CODIGO	NUMBER(6)
TITULO	VARCHAR2(40)
AUTOR	VARCHAR2(30)
PRECIO	NUMBER(6,2)

describe OFERTAS	
Nombre	Nulo Tipo

TITULO	VARCHAR2(40)
AUTOR	VARCHAR2(30)
PRECIO	NUMBER(6,2)

describe CONTROL	
Nombre	Nulo Tipo

USUARIO	VARCHAR2(30)
FECHA	DATE

- De la misma manera que el ejercicio anterior, se utiliza un ALTER SESSION para mostrar la fecha en un formato de 24hrs.

```
alter session set nls_date_format = 'DD/MM/YYYY HH24:MI';
```

Se crea nuestro TRIGGER según las características

```
create or replace trigger tr_ingresar_ofertas
before insert
on ofertas
for each row
begin
insert into control
values(user, sysdate);
end tr_ingresar_ofertas;
```

- La instrucción FOR EACH ROW indica que el trigger disparara a nivel fila.
- Para corroborar que el TRIGGER se ejecuto correctamente, entonces se insertan filas en la tabla libros

R	CODIGO	R	TITULO	R	AUTOR	R	PRECIO
	100		Uno		Ricahrd Bach		25
	102		Matematica estas ahi		Paenza		12
	105		El aleph		Borges		32
	120		Aprenda PHP		Molina Mario		55

- Después para ingresar valores a la tabla ofertas se especifica que solo se agreguen los libros que tienen un valor menor a 30.

```
insert into ofertas
select titulo, autor, precio from libros
where precio <30;
```

Donde se agregan 2 filas

R	TITULO	R	AUTOR	R	PRECIO
1	Uno	1	Ricahrd Bach	1	25
2	Matematica estas ahi	2	Paenza	2	12

Por lo tanto al consultar la tabla de CONTROL se muestran 2 eventos

R	USUARIO	R	FECHA
1	ORA21	1	12/01/2023 13:31
2	ORA21	2	12/01/2023 13:31

32. Crear un TRIGGER que inserte, elimine y actualice los datos de la tabla LIBROS previamente usada en los dos ejercicios anteriores de triggers.

- Utilizando las mismas tablas creadas que en los ejercicios anteriores y teniéndolos en cuenta entonces.
- Ahora directamente vamos a crear nuestro TRIGGER

```
create or replace trigger t_cambios_libros
before insert or update or delete
on libros
for each row
begin
if inserting then
insert into control values (user, sysdate, 'Insertado');
end if;
if updating then
insert into control values (user, sysdate, 'Actualizado');
end if;
if deleting then
insert into control values (user, sysdate, 'Borrado');
end if;
end tr_cambios_libros;
```

Para este trigger se anidaron a partir de la instrucción IF una serie de 3 instrucciones DML:INSERT, DELETE y UPDATE.

- A su vez las ejecuciones de cada una se almacenaron en la tabla CONTROL que muestra qué usuario efectuó cambios en tabla.

R	USUARIO	R	FECHA	R	OPERACION
1	ORA21	1	12/01/2023	1	Insertado
2	ORA21	2	12/01/2023	2	Borrado
3	ORA21	3	12/01/2023	3	Borrado
4	ORA21	4	12/01/2023	4	Borrado
5	ORA21	5	12/01/2023	5	Borrado
6	ORA21	6	12/01/2023	6	Actualizado

Sobre el ejercicio anterior, existe otra manera (así como otras múltiples) de realizar el trigger.

- A comparación del ejercicio anterior. En este se crea una variable a la que se le asigna cada acción realizada en la tabla LOG.
- Se debe de crear la tabla LOG2 que se encargara de guardar las transacciones que realicemos dentro de la primer tabla LOG.

```
CREATE TABLE log2 ( log_date DATE  
                    , action VARCHAR2(50));
```

- Y después mediante la instrucción INSERT se inserta cada “acción” dentro de la tabla LOG2.

```
create or replace trigger trg_audita  
after insert or update or delete  
on log  
declare  
log2_accion varchar2(15);  
begin  
    if inserting then  
        log2_accion := 'insert';  
    elsif updating then  
        log2_accion := 'update';  
    elsif deleting then  
        log2_accion := 'delete';  
    else  
        log2_accion := null;  
    end if;  
    insert into log2(log_date, action)  
    values(sysdate, log2_accion);  
end;
```

- Y a continuación un trigger que muestra los eventos que realizo algún usuario en específico dentro de un mismo schema.

```
create or replace trigger tgr_test_logon  
after logon  
on schema  
begin  
    insert into user_audit (usuario, actividad, fecha_evento)  
    values(user, 'LOGON', sysdate);  
end;
```

33. Crear un trigger que cuando se modifique cualquier campo de tabla EMPLEADO almacene en otra tabla CONTROL_CAMBIOS el nombre de usuario, fecha, dato y el nuevo valor que se le proporcione.

- En este ejemplo se utilizaran 2 tablas. La tabla *empleados* que almacenara todos los datos de los empleados y la tabla *control_cambios* que almacenara las veces que se modificar datos en la tabla empleados.

describe empleados			
Nombre	Nulo	Tipo	
DOCUMENTO	NOT NULL	CHAR	(10)
NOMBRE	NOT NULL	VARCHAR2	(30)
DOMICILIO		VARCHAR2	(30)
SECCION		VARCHAR2	(20)

describe control_cambios			
Nombre	Nulo	Tipo	
USUARIO		VARCHAR2	(30)
FECHA		DATE	
DATO_ANTERIOR		VARCHAR2	(30)
DATO_NUEVO		VARCHAR2	(30)

- Se van a crear 3 distintos triggers. Cada uno se encargará de manera independiente de insertar, borrar y actualizar datos de la tabla empleados. En este ejemplo se vera la funcionalidad de
- Primero el trigger que se encarga de controlar cuando se actualizan datos de la tabla empleados. En este caso se considera cada una de las columnas.

```
create or replace trigger tr_actualizar_empleados
before update
on empleados
for each row
begin
    if updating ('documento') then
        insert into control_cambios
        values( user, sysdate, :old.documento, :new.documento);
    end if;
    if updating ('nombre') then
        insert into control_cambios
        values (user, sysdate, :old.nombre, :new.nombre);
    end if;
    if updating ('domicilio') then
        insert into control_cambios
        values (user, sysdate, :old.domicilio, :new.domicilio);
    end if;
    if updating ('seccion') then
        insert into control_cambios
        values (user, sysdate, :old.seccion, :new.seccion);
    end if;
end tr_actualizar_empleados;
```

- Se crea un trigger que se encarga de controlar cuando se insertan empleados a la tabla

```
create or replace trigger tr_ingresar_empleados
before insert
on empleados
for each row
begin
    insert into control_cambios values (user, sysdate, null, :new.documento);
end tr_ingresar_empleados;
```

- Se crea un trigger que se encarga de controlar cuando se eliminan empleados.

```
create or replace trigger tr_eliminar_empleados
before delete
on empleados
for each row
begin
    insert into control_cambios values(user, sysdate, :old.documento, null);
end tr_eliminar_empleados;
```

- Se consulta el diccionario de triggers para validar que se deshabiliten o en su defecto los triggers se encuentren habilitados.

TRIGGER_NAME	TRIGGERING_EVENT	STATUS
TR_ACTUALIZAR_EMPLEADOS	UPDATE	DISABLED
TR_ELIMINAR_EMPLEADOS	DELETE	DISABLED
TR_INGRESAR_EMPLEADOS	INSERT	DISABLED

<https://www.tutorialesprogramacionya.com/oracleya>