# Programación de Bases de Datos (UD 6)

**Adaptado para Oracle Express Edition**

## Contenido

Programación de Bases de Datos (UD 6) .......... 1
Introducción al lenguaje de programación .......... 1
Variables .......... 1
Estructuras de control de flujo .......... 2
Procedimientos y funciones almacenadas .......... 3
Etiquetas en procedimientos .......... **¡Error! Marcador no definido.**
Función LAST_INSERT_ID .......... **¡Error! Marcador no definido.**
Ejecutar procedimientos y funciones .......... 6
Ventajas y Desventajas .......... 6
Creación de scripts .......... **¡Error! Marcador no definido.**
Triggers (Disparadores) .......... 7
Palabras clave NEW y OLD .......... 8
Codificación .......... 8
Control de errores con Triggers .......... 9
Usos y limitaciones de los disparadores .......... 10
Ejercicios .......... 11

## Introducción al lenguaje de programación

### Variables

Declaración de variables:

```
DECLARE nombre_variable tipo_variable
[DEFAULT valor];
```

Asignación de valor a una variable directamente:

```
nombre_variable := valor_variable;
```

Asignación de valor a una o más variables como resultado de una consulta. (La consulta debe devolver una sola fila)

```
SELECT campo1, campo2, . . . INTO variable1, variable2, . . .
FROM nombre_tabla WHERE . . .
```

## Estructuras de control de flujo

Hay que destacar que MySQL, actualmente, no soporta bucles \emph{FOR}

- Sentencia IF

```
IF condicion THEN
  sentencias;
[ELSEIF condicion2 THEN
  sentencias;] . . .
[ELSE
  sentencias;]
END IF
```

- Sentencia CASE

```
CASE variable
  WHEN valor1 THEN
    sentencias;
  [WHEN valor2 THEN
    sentencias;] . . .
  [ELSE
    sentencias;]
END CASE

CASE
  WHEN condicion THEN
    sentencias;
  [WHEN condicion2 THEN
    sentencias;] . . .
  [ELSE
    sentencias;]
END CASE
```

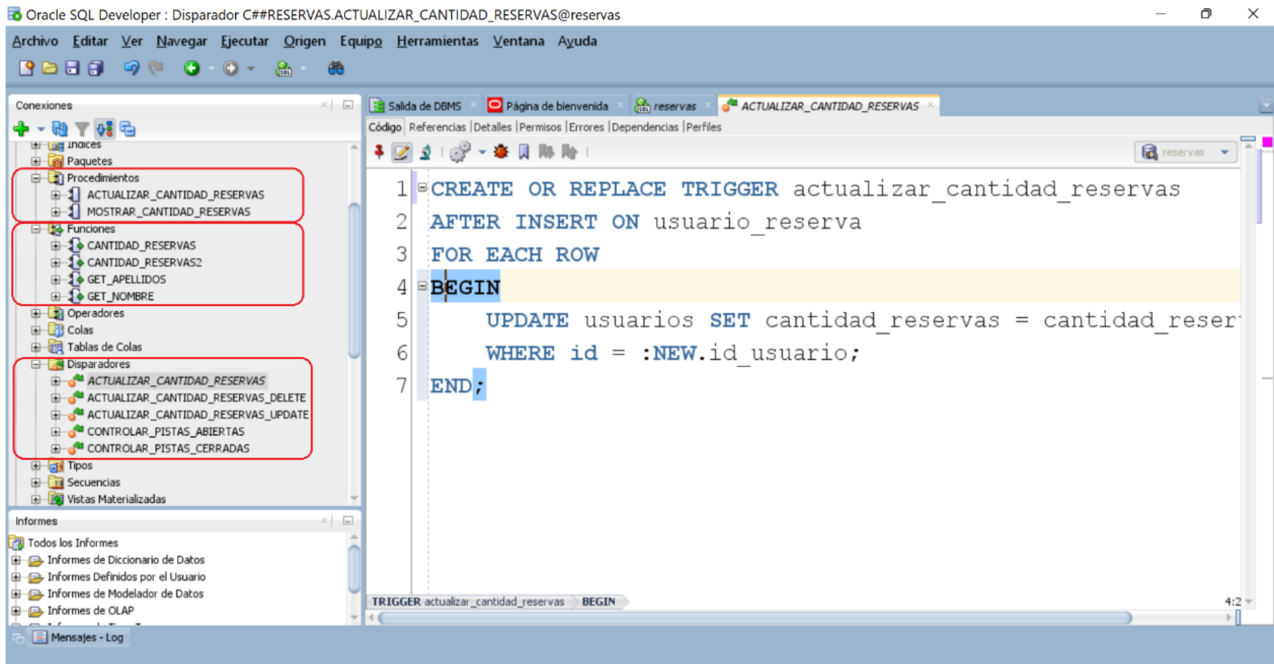
- Sentencia RETURN

Se utiliza para romper la ejecución de una función o de un procedimiento almacenado.

```
RETURN;
```

- Además, temenos otras sentencias como los bucles (ITERATE, REPEAT, LOOP, etc), aunque no profundizamos en ellos ya que no son tan intuitivos para programar rutinas.

Los elementos que vamos a crear en esta unidad se almacenan en el servidor del mismo modo que las tablas, los usuarios o los datos. Para consultar dónde se alojan, podemos utilizar el cliente *SQL Developer* de Oracle. Además, pulsando sobre los elementos creados, podemos ver el código con el que se han generado:




# Procedimientos y funciones almacenadas

Los procedimientos y funciones almacenadas son conjuntos de comandos SQL y PL/SQL que pueden ser almacenados en el servidor. Se asocia un nombre a un conjunto determinado de instrucciones para, posteriormente, ejecutar dicho conjunto tantas veces como se desee sin necesidad de volver a escribirlas.

```
CREATE [OR REPLACE] PROCEDURE nombre_procedimiento([param1 tipo, param2 tipo, . . .])
AS [ó IS]
[declaraciones_locales]
BEGIN
   sentencias;
END
```

```
CREATE [OR REPLACE] FUNCTION nombre_funcion([param1 tipo, param2 tipo, . . .])
RETURN tipo
IS
[declaraciones_locales]
BEGIN
   sentencias;
END
```

Para eliminar estos elementos, uso la sentencia DROP:

```
DROP PROCEDURE nombre_procedimiento;
DROP FUNCTION nombre_funcion;
```

Ejemplos:

```
-- Muestra toda la información sobre los usuarios
CREATE OR REPLACE PROCEDURE inserta_usuarios(p_nombre VARCHAR, p_apellidos VARCHAR)
BEGIN
  INSERT INTO usuarios (nombre, apellidos) VALUES (p_nombre, p_apellidos);
END;
```

```
-- Muestra la información sobre las reservas de una pista determinada
-- (se pasa como parámetro)
CREATE OR REPLACE FUNCTION obtener_pista_reserva(p_id_reserva INT)
RETURN INT
IS
 v_pista INT;
BEGIN
  SELECT id_pista INTO v_pista FROM reservas WHERE id_pista = p_id_pista;
  RETURN
END;
```

```
-- Procedimiento para dar de alta una nueva pista en un polideportivo
-- determinado. Se pasan como parámetros todos los datos necesarios
-- para dar de alta la nueva pista asumiendo que se trata de una
-- pista abierta al público

CREATE OR REPLACE PROCEDURE nueva_pista(p_codigo VARCHAR, p_tipo VARCHAR,
  p_precio FLOAT, p_id_polideportivo INT)
AS
 v_id_generado INT;
BEGIN

  INSERT INTO pistas (codigo, tipo, precio, id_polideportivo)
    VALUES (p_codigo, p_tipo, p_precio, p_id_polideportivo)
  RETURNING id INTO v_id_generado;
  INSERT INTO pistas_abiertas (id_pista, operativa)
    VALUES (v_id_generado, ‘Si’);

END;
```

```
-- Función que devuelva el número de reservas que ha realizado un usuario
CREATE OR REPLACE FUNCTION get_numero_reservas(p_id_usuario INT)
RETURN INT
IS
 v_cantidad INT;
BEGIN
  SELECT COUNT(*) INTO v_cantidad FROM reservas R, usuario_reserva UR
    WHERE R.id = UR.id_reserva AND UR.id_usuario = p_id_usuario;
  RETURN v_cantidad;
END;
```

```
-- Función que devuelva el número de reservas que ha realizado un usuario
-- determinado. Si el usuario no existe, devuelve -1
CREATE OR REPLACE FUNCTION get_numero_reservas(p_id_usuario INT)
RETURN INT
IS
 v_cantidad INT;
 v_existe_usuario INT;
BEGIN

  SELECT COUNT(*) INTO v_existe_usuario FROM usuarios
              WHERE id = p_id_usuario;
  IF existe_usuario = 0 THEN
    -- Si el usuario no existe se devuelve valor de error
    RETURN -1;
  END IF;

  -- Si todo va bien, se calcula la cantidad y se devuelve
  SELECT COUNT(*) INTO  v_cantidad  FROM reservas R, usuario_reserva UR
    WHERE R.id = UR.id_reserva AND UR.id_usuario = p_id_usuario);
  RETURN v_cantidad;
END;
```

## Salir de un procedimiento

Si deseo terminar un procedimiento en un momento dado, puedo usar la sentencia RETURN, del mismo modo que la uso en una función. Sin embargo un procedimiento no devuelve valor:

```
-- Procedimiento para dar de alta una nueva pista en un polideportivo
-- determinado. Se pasan como parámetros todos los datos necesarios
-- para dar de alta la nueva pista asumiendo que se trata de una
-- pista abierta al público

CREATE OR REPLACE PROCEDURE nueva_pista(p_codigo VARCHAR, p_tipo VARCHAR,
  p_precio FLOAT, p_id_polideportivo INT)
AS
  v_existe_polideportivo INT;
BEGIN

  SELECT COUNT(*) INTO v_existe_polideportivo FROM polideportivos
               WHERE id = p_id_polideportivo;
  IF existe_polideportivo = 0 THEN
    RETURN;

  END IF;

  INSERT INTO pistas (codigo, tipo, precio, id_polideportivo)
    VALUES (p_codigo, p_tipo, p_precio, p_id_polideportivo)

  RETURNING  id  INTO  v_id_generado;
  INSERT INTO pistas_abiertas (id_pista, operativa)
```

```
   VALUES (v_id_generado, 'Si');
END;
```

## Clausula RETURNING

Cuando queremos ejecutar varias operaciones que modifican la base de datos es posible que al realizar una inserción en una tabla cuya clave primaria es AUTONUMÉRICO (IDENTITY), necesitemos conocer el valor del *id* generado, para poder utilizarlo en la siguiente inserción en caso de que las tablas estén relacionadas.

La clausula RETURNIN de la sentencia INSERT INTO nos devuelve el valor del los campos que se han insertado, lo que nos permite acceder al valor de un índice auto generado (IDENTITY)

```
-- Se desea insertar una nueva pista abierta. Para ello debo
-- insertar la pista en la tabla pistas, y también en la tabla
-- pistas abiertas:


-- necesito una variable para almacenar el valor:
  v_id_generado INT;


-- Primero hago la inserción en la tabla padre: pistas
-- En la clausula RETURNING indico el valor de la columna que
-- deseo obtener (pistas(id)), y en INTO, la variable donde lo guardo
  INSERT INTO pistas (codigo, tipo, precio, id_polideportivo)
    VALUES (p_codigo, p_tipo, p_precio, p_id_polideportivo)
    RETURNING id INTO v_id_generado;

-- Después hago la inserción en la tabla hija: pistas_abiertas
-- Necesito el valor generado en la columna autonumérica
-- anterior, ya que es clave ajena en esta tabla

  INSERT INTO pistas_abiertas (id_pista, operativa)
    VALUES (v_id_generado, 'Si');
```

## Ejecutar procedimientos y funciones

Usamos EXECUTE para procedimientos y SELECT para funciones. Podemos usar una función en la cláusula SELECT de una consulta usando la tabla auxiliar DUAL.

```
EXECUTE nombre_procedimiento();

SELECT nombre_funcion() FROM DUAL;
```

Sin embargo la potencia de las funciones la obtenemos cuando son utilizadas en consultas en la que queremos realizar cálculos. Del mismo modo que he hecho con funciones de fecha, por ejemplo.

```
CREATE OR REPLACE FUNCTION cantidad_reservas(p_id_usuario INT)
RETURN INT IS
 v_cantidad INT;
BEGIN
```

```
    SELECT COUNT(*) INTO v_cantidad FROM usuario_reserva
    WHERE id_usuario = p_id_usuario;
    RETURN v_cantidad;
END;

-- Usamos la función anterior en otras consultas
UPDATE usuarios
SET cantidad_reservas = cantidad_reservas(id);

SELECT nombre, apellidos, cantidad_reservas(id)
FROM usuarios;
```

### Ventajas y Desventajas

- Resultan útiles, por ejemplo, en casos en los que varias aplicaciones diferentes trabajan sobre la misma Base de Datos y deben ejecutar el mismo código. En vez de escribir ese código para cada una de las diferentes aplicaciones, se puede escribir una sola vez en el servidor y ejecutarse desde las diferentes aplicaciones clientes.
- Además, aportan mayor seguridad puesto que las aplicaciones o usuarios no necesitan acceder directamente a la información de la Base de Datos, sino que solamente acceden a ejecutar determinados procedimientos o funciones.
- Y por último, se reduce el tráfico de red generado entre la aplicación cliente y el servidor al no tener que mandar multiples consultas, sino tan solo el nombre de un procedimiento o función.

- Como desventaja, resultan bastante complicados de escribir y mantener puesto que requieren conocimientos bastante precisos que no todos los programadores suelen tener.

## Triggers (Disparadores)

Los disparadores o triggers son procedimientos de la Base de Datos que se ejecutan o activan cada vez que ocurren un evento determinado sobre una tabla determinada, según se haya indicado en el momento de su implementación. Debemos indicar el momento en que se deben ejecutar (**timing**) y el **evento** que queremos que lo dispare.

#### Eventos

Los eventos que ocurren en una tabla ante los que podemos asociar a la ejecución de un trigger son:

- *INSERT*
- *UPDATE*
- *DELETE*

Esto quiere decir que un *trigger* se puede ejecutar al producirse uno de los eventos anteriores en una tabla concreta.

**Timing**

También podemos decidir que se activen antes o después del evento en cuestión, utilizando las palabras reservadas **BEFORE** y **AFTER**.

Respecto al timing debemos tener algunas consideraciones:

- Un trigger BEFORE se activa ante el simple intento de un INSERT, UPDATE, DELETE, independientemente de si estas sentencias pueden lanzar un error.
- Un trigger AFTER se ejecuta solamente si la operación que lo dispara se realiza correctamente.
- Un error durante la ejecución de cualquier trigger, cancela automáticamente la operación que lo disparó. Es útil para controlar valores no deseados.

## Palabras clave NEW y OLD

Las palabras NEW y OLD se emplean para referirse a las filas afectadas por el disparador, es decir, a las filas de la tabla sobre la que se activa, para referirse al estado de esa fila, antes (OLD) o después (NEW) de haber actuado el disparador.

Las referencias NEW y OLD no están disponibles siempre, ya que si se borran o insertan registros no existe modificación:

| Evento del Trigger | OLD | NEW |
|---|---|---|
| INSERT | NO | SI |
| UPDATE | SI | SI |
| DELETE | SI | NO |

Hay que tener en cuenta que cuando nos referimos a una columna precedida por OLD, el acceso es de sólo lectura, por lo que se podrá hacer referencia a ella sólo para leerla. En el caso de las columnas precedidas por NEW, su valor se podrá leer y también modificar con la instrucción SET, siempre que el trigger se active antes de la operación (BEFORE).

## Codificación

```
-- Crear trigger
CREATE TRIGGER nombre_trigger
{BEFORE | AFTER} {INSERT | UPDATE| DELETE }
ON nombre_tabla [FOR EACH ROW]
[WHEN condicion]
[DECLARE declaraciones locales]
BEGIN
  ...
  ...
END;

-- Borrar trigger
DROP TRIGGER nombre_trigger;
```

La sentencia FOR EACH ROW es opcional. Si se indica, el trigger se ejecuta por cada una de las filas afectadas. Es decir, si se borran 15 filas, se insertan ó se actualizan, el trigger se ejecutará 15 veces. Si no se indica, el trigger se ejecuta una sola vez.

Ejemplos:

```
-- Calcula automáticamente la edad de los usuarios
-- en el mismo momento en el que se dan de alta
-- a partir de la fecha de nacimiento que introduzca
-- el usuario
CREATE TRIGGER nuevo_usuario BEFORE INSERT ON usuarios
FOR EACH ROW
BEGIN
  IF :NEW.fecha_nacimiento IS NOT NULL THEN
    SET :NEW.edad = EXTRACT(YEAR FROM SYS_DATE()) -
                    EXTRACT (YEAR FROM :NEW.fecha_nacimiento);
  END IF;
END;
```

```
-- Actualiza la fecha de última reserva de una pista
-- cada vez que ésta se reserva
CREATE TRIGGER anota_ultima_reserva
AFTER INSERT ON reservas
FOR EACH ROW
BEGIN
  UPDATE pistas_abiertas
    SET fecha_ultima_reserva = CURRENT_TIMESTAMP()
    WHERE id_pista = :NEW.id_pista;
END;
```

```
-- Registra una pista como pista clausurada al público cuando
-- ésta se elimina de la Base de Datos
CREATE TRIGGER retira_pista
AFTER DELETE ON pistas_abiertas
FOR EACH ROW
BEGIN
  INSERT INTO pistas_cerradas (id_pista, fecha_clausura, motivo)
    VALUES (:OLD.id_pista, CURRENT_TIMESTAMP, 'Eliminada');
END;
```

También podemos terminar un trigger mediante la sentencia LEAVE, del mismo modo que en los procedimientos y funciones almacenadas.

## Control de errores con Triggers

Es interesante conocer el *Timing* de actuación de los triggers ya que podemos modificar los valores en los casos en que el trigger actue antes (**BEFORE**) del evento.

En los triggers que se disparan ante un UPDATE o un INSERT en una tabla, podemos acceder a la información del registro NEW y modificar sus valores:

```
-- Modificar valores incorrectos
CREATE TRIGGER comprobar_nota_examen
BEFORE INSERT ON examenes
FOR EACH ROW
BEGIN
```

```
IF :NEW.nota < 0 THEN
  :NEW.edad := 0
ELSEIF :NEW.nota > 10 THEN
  SET :NEW.nota = 10;
END IF;

END;

 -- ó infringir una restricción para impedir la operación
-- En caso de que se intente inserter una pista_cerrada,
-- que ya está en la tabla pistas_abiertas anulo la inserción
CREATE TRIGGER control_pista_cerrada
BEFORE INSERT ON pistas_cerradas
FOR EACH ROW
DECLARE
 v_existe  INT;
BEGIN
 SELECT  COUNT(*)  INTO  v_existe  FROM  pistas_abiertas
     WHERE  id_pista  =  :NEW.id_pista;

 IF v_existe <> 0 THEN
   :NEW.id_pista = NULL;
 END IF;
END;


-- También puedo lanzar una excepción
CREATE TRIGGER control_edad_socio
BEFORE INSERT ON socios
FOR EACH ROW
BEGIN

 IF :NEW.edad < 18 THEN
   RAISE_APPPLICATION_ERROR(-20001, ‘El usuario es menor’)
 END IF;
END;
```

## Usos y limitaciones de los disparadores

Antes de comenzar a utilizar los disparadores, conviene conocer cuándo deben ser utilizados, y cuáles son sus limitaciones.

Uno de los usos más comunes de los disparadores es el utilizarlos para mantener actualizados los campos calculados, de manera que cuando ocurra algún cambio en los datos se pueda actualizar automaticamente dicho campo calculado, si tuviera que verse afectado.

Además, permiten realizar tareas de auditoría, puesto que es posible registrar la actividad que ocurre en una o varias tablas en otra tabla, con el fin de registrar las operaciones que se realizan sobre ella, cuando se hacen, quién las hace, . . .

# Ejercicios

1. Con las siguientes tablas, implementa los procedimientos/funciones que se enumeran a continuación

Empleados (#id, nombre, apellidos, oficio, fecha_alta, salario, comision, -id_departamento)
Departamentos (#id, nombre, ubicacion)

- Muestra todos los empleados de un departamento determinado
- Da de alta un empleado
- Da de baja un empleado
- Sube el salario a todos los empleados de un determinado departamento
- Función que devuelva el salario total de los empleados de un departamento determinado
- Función que devuelva el número de empleados que trabajan en un departamento determinado

2. Sobre el modelo relacional del Ejercicio 1 del tema anterior, realiza los siguientes procedimientos/funciones:
   - Da de alta un jugador
   - Da de alta un equipo
   - Registra el resultado de un partido
   - Elimina un jugador
   - Anota las incidencias de un partido determinado
   - Función que devuelva el número de goles que un equipo ha metido en un partido
   - Función que devuelva el número de goles que un equipo ha metido a otro equipo, comprobando que ambos equipos existen

3. Dadas estas tablas, realiza los siguientes procedimientos/funciones:

Empleados (#id, dni, nombre, salario)
Vendedores (#id, nro_vendedor, zona, -id_empleado)
Polizas (#id, nro_poliza, importe, beneficiario, -id_vendedor, fecha, fecha_vencimiento)
Empleado_Jefe (#(-id_empleado, -id_jefe))

- Da de alta un empleado como vendedor. Comprueba que no existe otro empleado con el mismo DNI
- Asigna un jefe a un empleado determinado. Comprueba que ambos existen
- Registra una nueva póliza, comprobando si existe el vendedor que se le asigna
- Elimina una póliza determinado. Comprueba antes que existe
- Función que devuelva el número de vendedores
- Función que devuelva cuantas pólizas tiene asignadas un vendedor determinado. Comprobar si existe el vendedor
- Función que compruebe si un determinado empleado es jefe de otro empleado
- Función que devuelva cuantos vendedores hay en una zona determinada

4. Realiza los siguientes procedimientos/funciones sobre estas tablas:

Autores (#id, nombre, fecha_nacimiento, fecha_fallecimiento, nacionalidad)
Obras (#id, titulo, fecha, -id_museo)

Museos (#id, nombre, direccion, ciudad, pais)
Obra_Autor (#(-id_obra, -id_autor))

- Da de alta un autor. Comprueba que no existe anteriormente
- Da de alta una obra. Comprueba que no existe anteriormente y que el museo asignado existe
- Asigna una obra determinado a un autor
- Elimina un autor. Si éste tiene obras asignadas no podrá ser eliminado
- Elimina una obra. Si está asignada a algún autor, se desvinculará antes de éste
- Modifica el museo de una obra. Comprueba que el nuevo museo existe
- Elimina un museo. Asigna todas las obras de dicho museo a otro determinado
- Función que devuelva el número de obras de un museo determinado. Comprobar que existe dicho museo
- Función que devuelva el número de obras que ha creado un autor determinado. Comprobar que existe dicho autor
- Función que devuelva el autor de una obra determinada