Desarrollo con PL/SQL

Prof. Héctor Gómez Gauchía

NOTA: los ejemplos de las transparencias tienen su enunciado en hoja de ejercicios (se indica num.)

Introducción

- Evita necesidad de usar lenguaje externo, c++, java...
- Evita generar muchas peticiones de consultas que llegan al servidor
- Es de Oracle. Muy parecido al lenguaje
  - SQL/PSM (Persistent, Stored Modules) : Estándar ISO
- Incluye las sentencias procedimentales:
  - control (if, case), iteración (loop), excepciones, y las de SQL
- Un programa es una serie de *bloques*. Un bloque contiene:
  - Declaraciones de variables, cursores,... (DECLARE)
  - Cuerpo (BEGIN): son las sentencias a ejecutar
  - Excepciones: para tratar situaciones previsibles excepcionales.
- Los bloques se pueden anidar.
  - Hay que tener en cuenta la visibilidad de las variables.

- Las instrucciones SELECT cambian la sintaxis, veamos. . .
- **NOTA: TODAS LAS COMILLAS SIMPLES DEBEN SER `' verticales`**

Variables Simples en PL/SQL

- Son como declarar un atributo de una tabla: **`NombreCLI CHAR(9);`**
  - Se pueden usar los mismos tipos
- O bien usando un "patrón": es un atributo (**`NombreC`**) que ya existe en una tabla (**`cliente`**):

```
NombreCL    cliente.NombreC%TYPE;
DNICL       cliente.DNI%TYPE;
```

- El contenido inicial es el valor **`NULL`**
- Si puede asignar un valor con **`:= 'un valor'`**,
  - bien en la declaración o en cualquier lugar en el cuerpo del bloque
- Para declarar una constante:
  - **`DNICL CONSTANT cliente.DNI%TYPE;`**

Variables Compuestas en PL/SQL

- **Variables Compuestas** de varios atributos: (un "registro")
  - Contienen una FILA entera de una tabla con sus atributos:

```
MiCliente  cliente%ROWTYPE;
```

  - Consulto el dni y el nombre de cliente, luego inserto en Moroso:

```
SELECT * INTO MiCliente
FROM Cliente  WHERE DNI =  '123456789X';
insert into Moroso values
                (MiCliente.dni, MiCliente.Nombre);
```

  - Puedo usar la fila entera:

```
insert into Moroso values MiCliente;
```

Consultas en PL/SQL

- NO se puede usar : **`Select atributoXX  from TablaYY;`**
  - Salvo que esté en una subconsulta de otra consulta.
- Necesitas almacenar el resultado de la consulta en una variable:

  `Select atributoXX INTO VariableZZ from TablaYY;`

- El resultado solo puede ser una fila o un valor
  - según el tipo de variable: simple o compuesta

- En otro caso da una excepción (vemos como se captura más adelante)

  NO_DATA_FOUND : no encuentra ninguna fila

  TOO_MANY_ROWS : ha devuelto varias filas

Comentarios en PL/SQL

- **El código PL/SQL se deben documentar con comentarios**
  - Como todo lenguaje de programación

- Dos tipos de comentarios:

```
-- comentario de línea:se indica con dos guiones delante

/* comentario de varias líneas
     entre slash y asterisco para empezar
    y orden inverso para terminar ->  */
```

Instrucciones de Control – IF

- **IF clásico : La condición es la misma que en SQL**

```
IF condición THEN
   ...instrucciones ejecutadas si condición TRUE...
END IF;
```

```
IF condición THEN
   ... instrucciones ejecutadas si condición TRUE...

ELSE
   ... instrucciones ejecutadas si condición FALSE...

END IF;
```

Instrucciones de Control – IF

- IF en cascada

```
IF condición1 THEN
   ...instrucciones ejecutadas si condición1 es TRUE...

ELSIF condición2 THEN
   ...instrucciones ejecutadas si condición2 es TRUE...

ELSE
   ... instrucciones ejecutadas si condición1 y condición2
       son FALSE...

END IF;
```

Instrucciones de Control – CASE

- El selector: o expresión es opcional: (ej: un atributo)
  - Si está: se evalúa la 1ª vez y se compara con el valor de cada condición
  - Si no está: la **`condición_i`** se evalúa a TRUE o FALSE
- Se puede usar en una instrucción SQL
  - select atrib1, atrib2, case . . . from TablaX; (a partir de Oracle 9i)

```
CASE [ expresión que se compara con las condiciones ]
   WHEN condición_1 THEN result_1 y sale del CASE
   WHEN condición_2 THEN result_2 y sale del CASE
   . . . .
   WHEN condición_n THEN result_n y sale del CASE
   ELSE result
END
```

- Si no se pone la cláusula ELSE y no ha salido por ninguna opción
  - Oracle activa la excepción CASE_NOT_FOUND
  - Que se puede tratar en la sección de manejo de excepciones

```
EXCEPTION                    (las vemos más adelante)
   WHEN CASE_NOT_FOUND THEN  acciones a realizar
```

Instrucciones de Control – CASE Ejemplo

```
---- ESTE EJEMPLO es SQL.     Para usar en PLSQL: añade INTO
SELECT nombre, CASE dia_semana -- atrib.de Horarios  EJ-PLSQL9
                    WHEN 'L' THEN 'LUNES'
                    WHEN 'M' THEN 'MARTES
                    WHEN 'X' THEN 'MIERCOLES'
                    WHEN 'J' THEN 'JUEVES'
                    WHEN 'V' THEN 'VIERNES'
                    WHEN 'S' THEN 'SABADO'
                    ELSE 'DOMINGO'
               END AS DIA
 FROM Restaurantes, Horarios  WHERE codigo = codigores;
 ------------------------------------ Otro uso
   var_diaTexto:= CASE Var_dia_semana -- la letra del día
                    WHEN 'L' THEN 'LUNES'
                    WHEN 'M' THEN 'MARTES'
                    WHEN 'X' THEN 'MIERCOLES'
                    WHEN 'J' THEN 'JUEVES'
                    WHEN 'V' THEN 'VIERNES'
                    WHEN 'S' THEN 'SABADO'
                    ELSE 'DOMINGO'
               END
```

Instrucciones de Control – CASE Ejemplo

```
------- Otro uso es el de otros lenguajes: WHEN acción

CASE      - sin expression o selector
  WHEN Var_dia = 'L' THEN dbms_output.put_line('LUNES');
  WHEN Var_dia = 'M' THEN dbms_output.put_line('MARTES');
  WHEN Var_dia = 'X' THEN dbms_output.put_line('MIERCOLES');
  WHEN Var_dia = 'J' THEN dbms_output.put_line('JUEVES');
  WHEN Var_dia = 'V' THEN dbms_output.put_line('VIERNES');
  WHEN Var_dia = 'S' THEN dbms_output.put_line('SABADO');
  ELSE dbms_output.put_line('DOMINGO');
END CASE;
```

Instrucciones de Iteración

- Bucles de lenguajes procedimentales: Loop y For con contador

```
Loop
    . . . .
        if  CONDICION-de-fin-de-bucle  then exit;
    . . . .
end loop;
```

```
for  Contador in [reverse]  ValorInicio [.. ValorFinal]
Loop
   . . . .
   . . . .
end loop;
```

Instrucciones de Iteración

- Bucles de lenguajes procedimentales: For y While

```
VariableDeFila cursorX%rowtype; (veremos después con cursores)
For VariableDeFila in CursorX
loop
  . . . .
  . . . .
end loop;
```

```
while condición....
  . . . .
  . . . .
end loop;
```

Cursores

- Actúan como una vista o view
  - sin necesidad de crearla y borrarla manualmente.
  - Solo existe mientras existe el bloque de PL/SQL
  - Contiene filas del resultado de ejecutar una consulta

- Se recorren secuencialmente desde el principio ,fila a fila
  - Hay tres formatos: Loop, cursor fuera del For, cursor dentro del For
  - Las variables sobre las que se almacena el fetch
    - pueden ser de fila (row) o de atributo.
- En el Tema *Transacciones* veremos cómo actualizar la tabla de la consulta

- Son útiles cuando
  - las operaciones a realizar para cada tupla afectada
    - sean diferentes de acuerdo a algunas condiciones.

Cursor: Formato 1

- Funcionamiento:
  1) Declaración: usando instrucción **`CURSOR`**
  2) Abrir cursor antes de usarlo con **`OPEN`**: se coloca en la 1ª fila del cursor
  3) Tratar una a una las filas que contiene con **`LOOP`**:
     - Cada llamada **`fetch`** opera con una tupla: la que está apuntando
     - Sigue hasta que se acaban la filas: instrucción **`exit`**
  4) Cerrar el cursor con **`CLOSE`**
  5) Se ponen **`EXCEPTION`** para tratar situaciones especiales

- Atributos de cursores: NombreCur%Atrib (se activan automáticamente)
  - Después de Fetch, toman un valor: (antes de leer la 1ª vez = NULL)
    - NombreCur%FOUND: "TRUE" si ha leído fila, "FALSO" si no hay fila
    - NombreCur%NOTFOUND: "TRUE" si NO hay fila, "FALSE" si ha leído fila
  - NombreCur%ROWCOUNT: núm. Filas leídas hasta ese momento
    - Después de OPEN es = 0
  - NombreCur%ISOPEN: "TRUE" si abierto, "FALSE" si cerrado

Cursor: Formato 1

```
declare
Cursor NombreCU is          --almacena resultado consulta
       Select .....;
Sal NombreCU%rowtype;     -- variable de fila de cursor
Begin
 open NombreCU;
 loop
   fetch NombreCU into Sal;
        (tratamiento de la fila que está en Sal)
   exit when NombreCU%notfound;
         -- alternativa : if NombreCU%notfound exit;
   ----
   ----    (tratamiento de esa tupla)
    Sal.NombreAtributo    -- uso de v. fila y atributo
 end loop;
 Close NombreCU;
End;
```

Cursor: Formato 1, Ejemplo

```
. . . (resto del bloque,procedimiento o trigger)    EJ-PLSQL1
CURSOR cursor_ricos IS
       select dni, nombreC
       from cliente
       where dni in ( select dni from invierte
                      group by dni
                      having sum(cantidad) > 650000);
BEGIN
 OPEN cursor_ricos;
 LOOP
  FETCH  cursor_ricos INTO TDNICL, TNombreCL;
  EXIT WHEN cursor_ricos%NOTFOUND or
            cursor_ricos is NULL; -- si no hay ninguna fila
  IF TDNICL LIKE '%2' THEN
     DBMS_output.put_line('--- sumando , DNI: '|| TDNICL);
     update invierte
          set cantidad = cantidad + 1
     where dni = TDNICL;
  END IF;
 END LOOP;
IF cursor_ricos%ISOPEN  THEN  CLOSE cursor_ricos;
END;
```

Cursor: Formato 2, implícito

- Hace todo implícitamente.
- La var. de fila **empleado_rec** solo existe durante la ejecución del FOR
- Ej: Listar apellido de empleados del departamento 10
  - El resultado va a un buffer que solo lo muestra al TERMINAR la ejecución

EJ-PLSQL2

```
DECLARE
   empleado_rec empleados%rowtype
   CURSOR empleados_in_10_cur   IS
      SELECT *
        FROM empleados
       WHERE department_id = 10;
BEGIN
   FOR empleado_rec IN empleados_in_10_cur
   LOOP
      DBMS_OUTPUT.put_line (empleado_rec.apellido);
   END LOOP;
END;
```

Cursor: Formato 3

- Este FOR usa internamente un cursor
- Existe la var. de fila **`empleado_rec`** solo durante la ejecución

```
BEGIN
   FOR empleado_rec IN (
        SELECT *
          FROM empleados
         WHERE department_id = 10)
   LOOP
      DBMS_OUTPUT.put_line (empleado_rec.apellido);
   END LOOP;
END;
```

Tipos de Bloques: Formas de uso

- **Bloques sin nombre:**
  - Se escriben en el editor del SQLDeveloper
  - se ejecutan sin almacenarse
- **Procedimientos con nombre:**
  - Son bloques almacenados en oracle
  - que se pueden ejecutar llamandolos desde otro bloque
- **Funciones con nombre:**
  - Como procedimentos, que devuelven un valor
- **Disparadores (triggers):**
  - Como procedimientos, que se activan . . .
  - al detectar el SGBD un evento concreto
    - p.e.: se inserta una fila en una tabla concreta

Bloques sin Nombre

- **Estructura general**

```
DECLARE        --- declarar variables
...
BEGIN          -- cuerpo del bloque
. . .
EXCEPTION      -- captura excepciones
WHEN....
WHEN...
END;
```

- Se ejecuta tal cual en el editor. No se almacena en Oracle. Útiles para pruebas
- Ejemplo: leo un cliente y lo creo en la tabla Moroso

EJ-PLSQL3

```
DECLARE
     -- variables solo visibles dentro del bloque
   MiCliente  cliente%ROWTYPE;
BEGIN
   SELECT * INTO MiCliente
    FROM Cliente  WHERE DNI =  '00000001';
   insert into Moroso values MiCliente;
end;
```

Procedimientos con nombre

- Son bloques compilados y almacenados en Oracle
- Usando CREATE, se almacena en Oracle. Con DROP se elimina
- Parámetros:
  - De entrada, IN (por defecto): valor que se le pasa al proc. para usarlo dentro, no se puede modificar.
  - De salida,OUT: variable para sacar un valor que genera el proc.
  - Ambos, INOUT: introducir y sacar datos del proc.
- Tipos de datos de Parámetros: los mismos que los atributos sin tamaño
- Se puede asignar un valor inicial con "**:=**"

```
Create [or replace] procedure NombreProced (
   nombreAtrib1 [in| out| inout] TIPO1 [:= valor],
   -- para cada parámetro …) as
  NombreVar   TipoVar; -- declara variables
begin
       ----> aquí va el cuerpo del procedimiento
[exception
     when excep1 then …
]
end [NombreProced];
```

Procedimientos con nombre: Ejemplo

- Si existe el cliente, lo actualiza. Si no existe lo crea EJ-PLSQL-4

```
create or replace PROCEDURE crearActualizarCliente (
       dnibusca cliente.DNI%TYPE,     (ver proc3plus2.sql)
       NombreCL   cliente.NombreC%TYPE,
       TelCL  cliente.Telefono%TYPE ,
       DirCL      cliente.Direccion%TYPE
      ) as
BEGIN
  update cliente
       set NombreC = NombreCL
  WHERE DNI = dnibusca;

 IF SQL%NOTFOUND THEN
  DBMS_output.put_line('--- crea cliente:no existía '||
                                                 dnibusca);
  insert into cliente
              values (dnibusca,NombreCL,DirCL,TelCL);
  END IF;
END crearActualizarCliente;
```

Funciones

- Igual que procedimientos, pero devuelven un valor
  - del tipo definido en la cabecera con ***return untipo*** sin tamaño
- Se termina la ejecución con ***return NombreVar***

  que puede ser una expresión, variable (de ***untipo***)
- Se puede usar la recursividad llamándose una función a si misma

```
Create [or replace] function NombreFuncion (
       nombreAtrib1 [in| out| inout] TIPO1 [:= valor],
       -- para cada parámetro …)
    return untipo as
       ---- declaraciones de variables
 NombreVar   untipo;
begin
-----
----
return unaExpresión; -- o bien: return NombreVar

[exception
     when excep1 then …]
end [NombreFuncion];
```

Funciones: Ejemplo

- Ejemplo simplista de Función usando cursor
  - Que tome como parámetro un código de asignatura y devuelva su descripción
  - Tablas:

    Asignatura(Titulacion,codigoAsignatura)

    DescripcionesAsign(codigoAsignatura, descAsignatura)

- USO: Obtener código de asignaturas y descripciones del grado de software

```
SELECT codigoAsignatura, FindDesc(codigoAsignatura) AS
       descripcionAsignatura
FROM Asignaturas
WHERE Titulacion = 'Grado de Software';
```

Funciones: Ejemplo

```
--------- Encuentra la Descripción de la asignatura
CREATE OR REPLACE Function FindDesc(codigo_in IN number )
           RETURN varchar   IS   -- sin tamaño
 var_descAsig varchar(50);
 cursor c1 is
   SELECT descAsignatura
     FROM DescripcionesAsign
     WHERE DescripcionesAsign.codigo = codigo_in;
BEGIN
   open c1;
   fetch c1   into var_descAsig;
   if c1%notfound then
       var_descAsig := 'Asignatura sin Descripción';
   end if;
   close c1;
   RETURN var_descAsig;
EXCEPTION
WHEN OTHERS THEN
   raise_application_error(-20001,'Error inesperado -
     '||SQLCODE ||' -Texto- '||SQLERRM);
END;
```

Ejecución de Bloques, Procedimientos y Funciones

- **Ejecución de Un Bloque sin Nombre en el editor**: SqlDeveloper
  - Icono “Ejecuta Script” o F5
- **Ejecución de Procedimientos**:

a) En el editor

```
begin
  proc3('00000003','nombre 3 nuevo','33300333','dir3');
end;
```

→ **también con execute proc3(…)**

b) Desde otro procedimiento

```
proc3('00000003','nombre 3 nuevo', '33300333','dir3');
```

- **Ejecución de Funciones en el editor** (ver ejemplo en la descripción de funciones)

```
begin  -- en el editor
  varResultado := funcionX('para1');
end;
-- Desde otro procedimiento
   varResultado := funcionX('para1');
```

a) Excepciones de Usuario

- Tratar situaciones problemáticas sin salida lógica: parar el programa
- Se declaran así : **cliente_listillo EXCEPTION;**
- Se activan con **RAISE cliente_listillo;**
- Se capturan con **WHEN cliente_listillo;**
  - se ejecuta lo que hay a continuación del **WHEN**
  - Después, termina el bloque

```
Create function NombreFuncion
 . . . resto de código
cliente_listillo EXCEPTION;

Begin
   If . . . .THEN RAISE cliente_listillo; ENDIF;
. . . resto de código
EXCEPTION
  WHEN cliente_listillo        →   hacer lo que se necesite
     DBMS_output.put_line('se activó el listillo');

End NombreFuncion;
```

b) Excepciones Estándar de Oracle

- Son de Oracle y no se declaran
- Las activa Oracle al detectar un problema
- Si no la capturas con **WHEN**, Oracle da un mensaje de error y termina
- En la variable **SQLCODE** está el código de error
- En la variable **SQLERRM** está el texto del mensaje
  - Lo cortamos a 100 caracteres con **SUBSTR**

```
Create function NombreFuncion . . . resto de código
  nombreCliente cliente.NombreC%TYPE;          EJ-PLSQL-6
Begin
  Select nombreC into nombreCliente from cliente
    where dni = '888888888';
 . . . resto de código
EXCEPTION
WHEN NO_DATA_FOUND THEN
  DBMS_output.put_line('No existe ese DNI,código Oracle: '
     || SQLCODE  || 'texto Oracle: '
     || SUBSTR(SQLERRM,1, 100));
WHEN OTHERS THEN
  DBMS_output.put_line('cualquier otro error');
end [NombreFuncion];
```

Lista Excepciones Estandar Frecuentes

PROGRAM_ERROR	ORA-6501	PL/SQL tiene un problema interno
LOGIN_DENIED	ORA-1017	Error al conectarse con Oracle
CASE_NOT_FOUND	ORA-06592	Ninguna opción WHEN dentro de la instrucción CASE captura el valor, y no hay instrucción ELSE
DUP_VAL_ON_INDEX	ORA-00001	valor duplicados en índice unique
CURSOR_ALREADY_OPEN	ORA-06511	Se intenta abrir un cursor que ya se había abierto
INVALID_CURSOR	ORA-01001	Se realizó una operación ilegal sobre un cursor
INVALID_NUMBER	ORA-01722	conversión errónea de carácter a número
NO_DATA_FOUND	ORA-01403	El SELECT no devolvió nada (PL/SQL)
TOO_MANY_ROWS	ORA-01422	El SELECT devolvió varias filas (PL/SQL)
ROWTYPE_MISMATCH	ORA-06504	Hay incompatibilidad de tipos entre valores de fila actual y las variables a las que se asignan (PL/SQL, cursor)
VALUE_ERROR	ORA-06502	Hay un error aritmético, de conversión, de redondeo o de tamaño
ZERO_DIVIDE	ORA-01476	Se intenta dividir entre el número cero.

c) Excepciones de Aplicación

- Solo da un mensaje y parar la ejecución
- Las activa el programa con **RAISE_APPLICATION_ERROR**
- En la variable **:v_mensa** ponemos el texto para el mensaje
- Damos un código de error: valor permitido entre -20000 y -20999
- Ejemplo: Si el tipo de una tarjeta es xxx es un error y para el programa

```
Create function NombreFuncion
----- resto de código
VAR CHAR(8) := 'xxx';
Begin
----------- resto de código
  If TipoT ='xxx' THEN
   RAISE_APPLICATION_ERROR(-20999, 'Tipo Tarjeta Erroneo:'
     || SQLCODE || 'texto: '|| SQLERMM)
END IF;
End NombreFuncion;
```

c) Excepciones de Aplicación: otro uso

- Otro uso típico: Capturar un error inesperado
  - da ese mensaje y el código con el texto de oracle

```
--- resto del código
EXCEPTION
WHEN OTHERS THEN
  RAISE_APPLICATION_ERROR(-20999, 'Algo raro ha pasado:'
     || SQLCODE || 'texto: '|| SQLERMM)

End NombreFuncion;
```


Pensando en PLSQL: Método para resolver un problema

1. Con qué datos me piden que trabaje, tablas, atributos?
   - Quitar todos los datos que sobren, filtrando con un cursor

2. Qué bucles necesito sobre el cursor?
   - Loop . . . Qué condición de salida
   - For haciendo cursor implícito

3. Qué proceso quiero hacer con cada fila del cursor?

4. Tratar posibles excepciones
   - Siempre, al menos poner la general `WHEN OTHERS`

Disparadores: triggers

- *Como hacer esto en Oracle?:* **EJEMPLO A:**
  - Cuando el saldo de una tarjeta en BDejemplo
  - sea menor que –1000 € , entonces: pasar el cliente a moroso
- Bases de datos Activas:
  - reaccionan a eventos y ejecutan automáticamente instrucciones.
- Paradigma *Evento-Condición-Acción* (ECA)
  - evento: actualiza datos con insert, delete, update
  - condición (opcional): predicado SQL
  - acción: secuencia de instrucciones SQL o un procedimiento
- Mecanismo:
  - cuando un evento ocurre (triggering)
  - si la condición se satisface
  - entonces se ejecuta la acción (se activa o “dispara” el trigger)
- Cada trigger se asocia a eventos de UNA SOLA TABLA (“on”)

Pensando en Disparadores: triggers

- **Sintaxis**

```
create trigger NombreTrigger
<Modo> <Evento>  {, <Evento>}   of   NombreAtributo
      on TablaObjetivo ¿qué tabla y atributo lo activan?
[ [referencing Referencia]  NO USAR
[ for each row ]
   [ when <PredicadoSQL>]]
<Bloque entero de PL/SQL>
```

- **Donde**

<Modo>: **before, after** ¿cuándo quiero que se active?

se ejecuta antes (before) o después(after) de la operación que lo activa

<Evento>: **insert, update, delete** (uno o varios)

- **Granularidad:** ¿cuántas veces quiero que se active? dos niveles
  - Trigger de Fila, **for each row**: se activa una vez para
    - cada fila donde ocurre el evento y se cumple el **when**
  - Trigger de Instrucción (sin el **for each**): se activa una sola vez para
    - todas las filas donde ocurre el evento y se cumple el **when**
- ¿Qué necesito hacer cuando se active?(Cuerpo trigger)

Disparadores: triggers

- **Ejemplo A:** si su saldo es <-1000 inserto ese dni en tabla Moroso

```
create trigger TR1
       After update OF saldo ON tarjeta
       for each row
       when new.saldo < -1000
declare        (si hay variables)
begin
Insert into moroso values (:new.dni, 'emp1')
end TR1;
```

- (solo si **for each row**) valor en la fila afectada que ha disparado el Trigger
  - "old": antes de la operación "new": valor nuevo de la operación
- (solo si **for each row**) Existen esos valores cuando la operación lo permite:
  - Insert: solo new. Delete: solo old. Update: ambos

- **IMPORTANTE: Error de "Tabla Mutante"**
  - No se puede modificar dentro del cuerpo del trigger
    - La tabla asociada al trigger (la que tiene en "on")
      - o cualquiera relacionada con ella por "foreign key"

Para qué usar los Triggers

Triggers de Tabla:

- Gestión de restricciones sobre tablas:
  - controlar duplicidades, DMs, DFs, generar atributos derivados
  - Forzar Reglas de Negocio, FKs de tablas en distintos nodos
- Hacer una tabla de LOGs o Auditoría con los sucesos o eventos de otra tabla
- Detección de situaciones especiales, de alarma, inesperadas, prohibidas,etc

Triggers de sistema:

- Dar avisos generales del sistema: shutdown, startup, del servidor, etc.

Triggers de usuario:

- Actuar en eventos de usuario: log on, log off

Triggers con eventos combinados: multitriggers

- Se tratan en el mismo Trigger la tres posibles operaciones por separado

```
create trigger TR1
before     -- o after
INSERT or DELETE or UPDATE OF campo1 ON tablaX
   FOR EACH ROW
BEGIN
IF DELETING THEN
   Instrucciones ejecutadas si se activó por borrar fila

ELSIF INSERTING THEN
 Instrucciones ejecutadas si se activó por insertar fila

ELSE
 Instrucciones ejecutadas si se activó por actualizar fila
END IF
END;
```

Triggers en la misma tabla: Orden de ejecución

- Orden de ejecución Si una tabla tiene alguno de estos triggers
  1. Ejecuta triggers con BEFORE y de tipo instrucción
  2. Ejecuta triggers con BEFORE y de tipo Fila (una vez por cada fila)
  3. Se ejecuta la instrucción que activó el trigger.
  4. Ejecuta triggers con AFTER y de tipo Fila (una vez por cada fila)
  5. Ejecuta triggers con AFTER y de tipo instrucción (una vez)

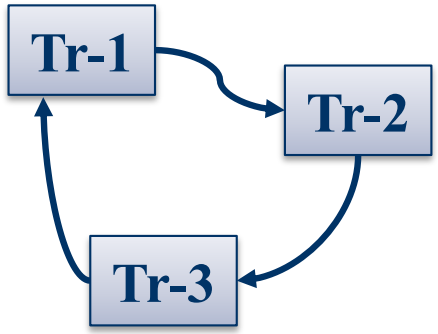
- Forzar el orden de ejecución dentro del mismo tipo 1,2,3,4,5 y tabla
  - Poniendo en TriggerSegundo: FOLLOWS TriggerPrimero

- La activación automática se triggers
  - Puede ser difícil de seguir: cuando se activa cada uno
  - Y causar problemas
    - si hay varios triggers sobre la misma tabla
    - Si afectan a un mismo proceso

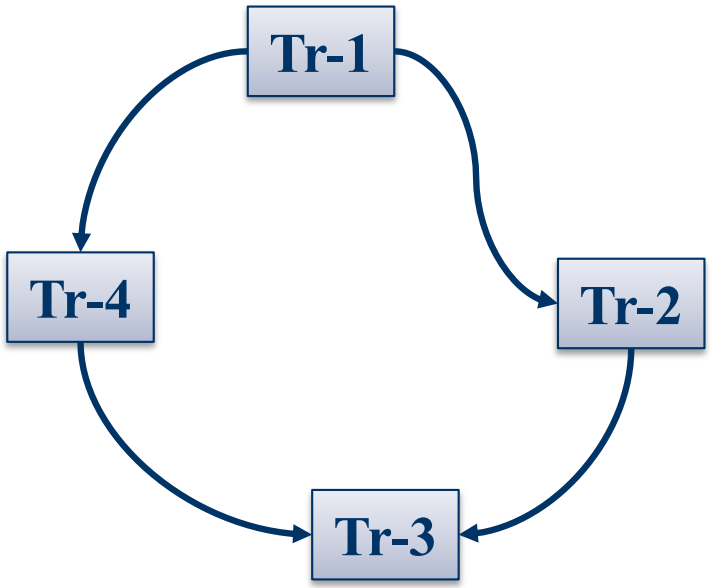
Triggers activados en cascada: Problemas

Tres tipos de Problemas

- Problemas de bucles infinitos:
  - Triggers que activan en cadena otros
  - Pueden activarse en bucle



- Problemas de prioridad
  - Cuando se pide activar el mismo TR-3 desde otros dos TR-4 y TR-2 que tienen el mismo origen TR1



- Problemas de bloqueos de filas y tablas
  - Se verá en tema de Transacciones

Operaciones sobre triggers

- Ver qué triggers tengo creados:

```
select trigger_name from user_triggers;
```

- Más detalles de un solo trigger:

```
select trigger_type, triggering_event,
       table_name, trigger_body
from user_triggers
where trigger_name = 'nombreDeTuTrigger';
```

(poner comillas verticales)

- Borrar trigger: `drop trigger nombreDeTuTrigger;`
- Deshabilitar/Habilitar un trigger:

```
alter trigger nombreDeTuTrigger {disable | enable};
```

- Deshabilitar/Habilitar todos los triggers de una tabla:

```
alter table nombreTabla {disable|enable} all triggers;
```

- Estas operaciones se pueden hacer on-line en el sqlDeveloper
  - Cuando las vas hacer periódicamente las pones en un “procedure”

Ejemplos sobre triggers

- **EJEMPLO B**: En la BDejemplo, no permitir que compre nadie con saldo de tarjeta menor de –1000 €.

  Cuando alguien lo intente se activará una excepción de error y TERMINA.

```
create or replace trigger ErrorSaldo
    before insert or update OF saldo ON tarjeta
    for each row
    when (new.saldo < -1000)
begin
    RAISE_APPLICATION_ERROR(-20999,
                            'saldo no permitido');
end ErrorSaldo;
```

- → Dará un mensaje de error : "ORA-20999 saldo no permitido"
- El número lo das tu, entre -20000 y -20999
- Activación :

```
update tarjeta set saldo = -1001 where NumT = '10000001'
```

Ejemplos sobre triggers

- **EJEMPLO B**: Otra versión de la solución

```
create or replace trigger ErrorSaldo
    before insert or update OF saldo ON tarjeta
    for each row
begin
    if (:new.saldo < -1000)
    then RAISE_APPLICATION_ERROR(-20999,
                                 'saldo no permitido');
   end if;
end ErrorSaldo;
```

- → Dará un mensaje de error : ORA-20999 saldo no permitido
- Activación :

```
update tarjeta set saldo = -1001 where NumT = '10000001'
```

Ejemplos sobre triggers

- **EJEMPLO C**: En la BD de universidad,

```
Asignaturas(codasig, creditos)
Profesores (DNI, Nombre, ToTcred)
Imparte(DNI, codasig)
```

SE PIDE:

Almacenar en Profesores el total de créditos que imparte cada uno y actualizarlos automáticamente cada vez que se modifiquen los créditos de una asignatura.

Se asume que varios profesores imparten la misma asignatura.

Es la típica redundancia cuando decidimos desnormalizar:

El total de créditos los almacenamos por eficiencia aunque son redundantes porque puedo obtenerlos con **`Imparte`**

Ejemplos sobre triggers

- **EJEMPLO C** solución:

```
create or replace trigger TotCreditos
       After insert or update OF creditos ON asignaturas
       for each row
declare
  TDNI CHAR(8);
  CURSOR profesAfectados IS select DNI from imparte
                            where codasig = :new.codasig;
begin
   OPEN profesAfectados;
   loop
     FETCH profesAfectados into TDNI;
     EXIT WHEN profesAfectados%NOTFOUND;
     IF UPDATING THEN
      update profesores
      set totcred = totcred - :old.creditos + :new.creditos
     where profesores.DNI = TDNI;
     ELSE  update profesores
             set totcred = totcred + :new.creditos
            where profesores.DNI = TDNI;
   end loop;
   CLOSE profesAfectados;
```

Ejemplos sobre triggers

- **EJEMPLO C** – Continúa:

```
EXCEPTION
        WHEN OTHERS THEN
                DBMS_OUTPUT.put_line('error imprevisto');
END TotCreditos;
```

- **Activacion:**

```
update asignatura set creditos = 5 where codasig like '6%';
```

- **Se puede hacer sin cursor, p.e. para UPDATING**

```
update profesores
   set totcred = totcred - :old.creditos + :new.creditos
where profesores.DNI in
                select DNI
                from imparte
                where codasig =   :new.codasig;
   end loop;
```

PL/SQL Dinámico

- Genera Código en tiempo de ejecución
- Muy útil cuando no se sabe qué ejecutar al escribir el programa:
  - Ejecutar inserciones sin saber en que tabla hasta la ejecución
  - Crear tablas, si solo se sabe cual en tiempo de ejecución
  - Repetir varias veces la ejecución de un bloque /proc complejo
    - para diferentes argumentos dependiendo de varias condiciones
- Dos partes: declarar una variable con el texto y variables a sustituir
  - Ejecución con **EXECUTE IMMEDIATE** con variables IN, OUT, IN OUT
  - IN: los valores dados a la instrucción
  - OUT: valores que devuelve la instrucción (ej.: si es una Función)

```
BEGIN
plsql_block := 'trozos texto de SQL con :variables a sustituir'

EXECUTE IMMEDIATE plsql_block USING IN OUT valor1,…, valorN;
```

PL/SQL Dinámico: Ejemplo

- Escenario:
  - Crear clientes de un banco distribuidos en una tabla para cada sucursal
  - Hay 1000 sucursales
- Generamos el insert en la tabla Cliente de la sucursal correspondiente
- Los nombres de tablas y columnas hay que concatenarlos con | |
- Los parámetros :a, :b, etc… corresponden a **dnibusca, NombreCL, …**

```
create or replace PROCEDURE insertaDinam (
        NombreSucursal  VARCHAR2,
        dnibusca    cliente.DNI%TYPE,
        NombreCL    cliente.NombreC%TYPE,
        DirCL       cliente.Direccion%TYPE,
        TelCL  cliente.Telefono%TYPE)
as
    plsql_block VARCHAR (2000);
BEGIN                → quitar "begin" y "end ;" si es ddl : CREATE,…
uso las VARS :a, :b si ejecuto con diferentes valores varias veces
  plsql_block := 'BEGIN insert into ` || NombreSucursal ||
                                ` values (:a, :b, :c, :d); END;';
EXECUTE IMMEDIATE plsql_block USING IN
                dnibusca, NombreCL, DirCL, TelCL;
END;
```

Variables sustitución: Datos de Pruebas

- Ejecución en el editor: pide cada dato con el nombre después del '&
  - Solo al empezar el proc.
- Util para: probar un proc. Aislado, simulando datos que viene de otros procs todavía no creados.

```
create or replace PROCEDURE procBind as
       dnibusca CHAR(8) := '&dniCliente';
       NombreCL   cliente.NombreC%TYPE:= '&nombreCliente';
       TelCL  cliente.Telefono%TYPE := '&TelCliente';
       DirCL      cliente.Direccion%TYPE := '&DirCliente';

BEGIN
 insert into cliente values
            (dnibusca, NombreCL, TelCL, DirCL);

END procBind;
```

Imprimir por salida standard del SQLDeveloper

- Para pruebas es muy útil dar mensajes con información sobre la ejecución
- Para ver lo impreso por la salida se usa
  - paquete **DBMS_OUTPUT**
  - Y dos métodos **PUT_LINE y NEW_LINE**
- **Ejemplo**

```
DBMS_output.put_line(' sumando , DNI: '|| TDNICL);
```

- Para que salga el texto al final de la ejecución completa:
  - en el editor ejecutar antes

```
SET SERVEROUTPUT ON SIZE 1000000
```