Introducción a procedimientos almacenados PL/SQL y disparadores

Bases de Datos

Curso 2018-2019

Jesús Correas – jcorreas@ucm.es

Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid

.

Bibliografía

- Bibliografía básica:
  - `http://www.plsqltutorial.com/` (en inglés).
  - **Oracle Database PL/SQL Language Reference 11g Release 2 (11.2).** E25519-13. `https://docs.oracle.com/cd/E11882_01/appdev.112/e25519.pdf`
- Bibliografía complementaria:
  - R. Elmasri, S.B. Navathe. **Fundamentals of Database Systems** (6a Ed). Addison-Wesley, 2010. (en español: **Fundamentos de Sistemas de Bases de Datos** (5a Ed). Addison-Wesley, 2007). Capítulo 13 (6a ed.)

Programación de aplicaciones de BD

- **SQL** es un lenguaje de consulta muy potente, pero **no es un lenguaje de programación.**
- Para realizar operaciones complejas sobre la BD se necesita un **lenguaje de programación procedimental.**
- Normalmente se utilizan las BD desde **aplicaciones** que **se conectan** al gestor de BD y **envían** consultas SQL para su ejecución:
  - Suelen estar programados en lenguajes de propósito general (Java, C++, PHP, Javascript, etc.).
  - Acceden a la BD mediante librerías como JDBC, ODBC, ADO.NET.
  - Acceden a la BD utilizando una **arquitectura cliente/servidor:** Cada instrucción se envía a través de la red para que la ejecute el SGBD.
- Para operaciones complejas sobre la BD, **este enfoque es poco adecuado:**
  - Es **ineficiente.**
  - El programa resultante es complejo, por el **diferente enfoque** de SQL frente a los lenguajes procedimentales.

Ventajas de Procedimientos almacenados

- La mayor parte de los SGBD actuales proporcionan un **lenguaje de programación procedimental en la propia BD.**
- Agrupa en un **bloque** una serie de operaciones que se puede ejecutar en la BD **sin necesidad de comunicar con la aplicación cliente** por cada operación de BD.
- Intercala de forma sencilla instrucciones SQL con operaciones de un lenguaje procedimental.
  - Reduce el **desajuste** (*Impedance Mismatch*) entre el lenguaje SQL y el lenguaje procedimental.
- Se pueden compilar **subprogramas** y almacenarlos en la BD junto con los demás elementos de la BD.
  - Es más eficiente y proporciona un alto nivel de seguridad.
  - Reutilizable desde distintas aplicaciones.
- El código de los subprogramas es **optimizable** para aumentar la eficiencia de las operaciones de BD.
- El código es **portable** a cualquier otra instalación del SGBD.

Ventajas de los procedimientos almacenados

- Casi todos los SGBD disponen de un lenguaje para escribir procedimientos almacenados:
  - Oracle: **PL/SQL.**
  - MySQL: *Stored Procedure Support* (a partir de la versión 5.0.0).
  - Microsoft: **TransactSQL.**
  - PostgreSQL: **PL/pgSQL.**
  - IBM DB2: **SQL Procedural Language.**
- Los estándares **SQL:1999** y **SQL:2003** especifican un lenguaje (SQL/PSM) para crear funciones y procedimientos, pero cada fabricante proporciona variantes no muy compatibles.
- Todos estos lenguajes se caracterizan por tener **dos niveles de representación del procedimiento:**
  - El lenguaje procedimental.
  - El lenguaje de acceso a datos SQL.
- Aunque se facilita la integración de los dos niveles, **siempre hay que tener en cuenta el nivel de cada elemento de un procedimiento.**

PL/SQL: bloques, funciones, procedimientos, disparadores

- En Oracle se utiliza el lenguaje **PL/SQL.**
  - Lenguaje procedimental imperativo con **variables locales, estructuras de control, procedimientos y funciones con parámetros y gestión de excepciones.**
- Hay cuatro tipos fundamentales de bloques de código en PL/SQL:
  - **Bloque anónimo:** Es un fragmento de código sin nombre que se ejecuta una sola vez.
  - **Procedimiento:** Es un fragmento de código con nombre que puede tener **parámetros** de entrada, salida o ambos.
  - **Función:** Es un fragmento de código que puede tener parámetros y devuelve un **valor de retorno.**
  - **Disparador** (*trigger*): Es un fragmento de código que **se ejecuta automáticamente** cuando ocurre un evento, normalmente una modificación de la BD.
- Todos los tipos de bloques excepto los bloques anónimos se **compilan** y **almacenan** como objetos de la base de datos.
- Veremos una **introducción a este lenguaje.**

bloques PL/SQL anónimos

- Un bloque anónimo se declara y **se ejecuta una sola vez**: **como no tiene nombre, no se puede almacenar en la BD ni invocar.**
- **`ejemplo01.sql:`** un ejemplo sencillo de bloque PL/SQL anónimo es:

```
-- ejemplo01.sql
DECLARE -- Sección de declaraciones
  varSaludo VARCHAR2(20);
BEGIN   -- Sección de instrucciones
  varSaludo := 'Hola Mundo';
  DBMS_OUTPUT.PUT_LINE(varSaludo);
END;
/   -- En SQL*Plus y SQLDeveloper se debe finalizar el bloque
    -- con una barra para que lo ejecute.
```

- Los bloques **se pueden anidar** (como en Java o C++).
- Los bloques están estructurados en **secciones**. La única obligatoria es la sección de **instrucciones**, que incluye **`BEGIN`** y **`END`**.
- Un bloque puede tener hasta tres secciones: **declaraciones** (**`DECLARE`**), instrucciones y **excepciones** (**`EXCEPTION`**).

Sección de declaraciones

- Debe comenzar con la palabra **DECLARE**.
- Permite declarar variables locales del bloque con un tipo determinado y asignar un valor inicial.
- Los tipos permitidos son los mismos que en las columnas de las tablas y otros más específicos de PL/SQL. Los más importantes:
  - **VARCHAR2**(n) Texto variable de hasta n caracteres.
  - **NUMBER**(p,s) Variable numérica de p dígitos, de los que s son decimales.
  - **INTEGER** (Enteros de 16 bits), **DATE**, BOOLEAN, etc.
- Veremos que pueden declararse variables de tipo **registro**, *arrays*, **cursores** y excepciones de usuario.
- Pueden tomar un valor inicial en la propia declaración.
- Pueden declararse como constantes con **CONSTANT.**
- Ejemplos:

```
DECLARE
  nombre VARCHAR2(50) := 'Valor inicial.';
  saludo CONSTANT VARCHAR2(11) := 'Hola Mundo!';
  v_importe NUMBER(12,2);
```

Sección de declaraciones: %TYPE

- Se pueden declarar variables (y parámetros) cuyo tipo **está referenciado al tipo de una columna de una tabla** o a otra variable.
- Así no es necesario buscar en la descripción de la tabla para definir la variable del tipo adecuado.
- Además, **Si cambia el tipo de la columna**, al recompilar el programa PL/SQL la variable se crea con el nuevo tipo.
- Por ejemplo, si se dispone de la tabla `Clientes` con la descripción:

```
CREATE TABLE Cliente(
  dni VARCHAR2(9) PRIMARY KEY,
  nombre CHAR(35) NOT NULL,
  importe_maximo NUMBER(12,2)
);
```

- Se pueden declarar variables PL/SQL de la siguiente forma:

```
DECLARE
  v_dni_cliente Cliente.dni%TYPE;
  importe_total Cliente.importe_maximo%TYPE := 0.0;
```

Sección de instrucciones: asignación

- La **sección de instrucciones** debe comenzar con la palabra **BEGIN**. Es la única sección obligatoria en un bloque anónimo.
- En ella se pueden utilizar asignaciones **a variables locales**, instrucciones de control de flujo y llamadas a procedimientos y funciones, combinadas con otras de acceso a datos (SQL y de gestión de cursores).
- La **asignación** tiene la forma:

```
var := expresion; -- var debe ser una variable local!
```

- La expresión puede combinar **variables locales** y literales (numéricos, texto entre comillas simples) mediante operadores:
  - Numéricos: **+, -, *, /, ****
  - Concatenación de texto: **||**

Sección de instrucciones: Instrucciones condicionales

- Sintaxis:

```
IF condicion THEN
  instrucciones
END IF;
```

```
IF condicion1 THEN
  instrucciones1
ELSIF condicion2 THEN
  instrucciones2
...
ELSE
  instrucciones
END IF;
```

- La condición puede ser cualquier expresión lógica que combine **variables locales** y literales (numéricos, texto entre comillas simples) mediante operadores:
  - relacionales: **`<, <=, >, >=, =, !=`**,
  - Booleanos: **`AND, OR, NOT`**
  - de comprobación de NULL: **`expr IS [NOT] NULL`**
- Los nombres de columna **solo están permitidos si se utilizan en cursores o registros.**

Sección de instrucciones: Selección múltiple

- Similar a la sentencia **`switch`** de C++ y Java.
- Pero en PL/SQL no es una instrucción, sino una **expresión que devuelve un valor.**
- Ejemplo:

```
calif_alfa := CASE calif
  WHEN NULL THEN 'Sin calificar'
  WHEN 'SB' THEN 'Sobresaliente'
  WHEN 'NT' THEN 'Notable'
  WHEN 'AP' THEN 'Aprobado'
END;
```

- También se puede utilizar para cualquier expresión, como un **`IF`** compuesto:

```
calif_alfa := CASE
  WHEN calif >= 9 AND ejerc = 'apto' THEN 'Sobresaliente'
  WHEN calif < 9 AND calif >= 7 AND ejerc='apto' THEN 'Notable'
  WHEN calif < 7 AND calif >= 5 THEN 'Aprobado'
END;
```

Sección de instrucciones: Bucles

(los elementos entre paréntesis cuadrados son **opcionales**).

- Bucle **LOOP** general:

```
LOOP
  instrucciones
  EXIT [WHEN condicion] -- en cualquier parte del bucle!
  instrucciones
END LOOP;
```

- Bucle **WHILE:**

```
WHILE condicion LOOP
  instrucciones
END LOOP;
```

- Bucle **FOR** numérico:

```
FOR variable IN [REVERSE] valorInf..valorSup LOOP
  instrucciones
END LOOP;
```

- Más adelante veremos una variante de bucle `FOR` para **iterar sobre los resultados de consultas SQL.**

Uso de funciones y procedimientos externos en paquetes

- Aunque no lo veremos en esta introducción, PL/SQL permite definir paquetes para formar librerías de código.
- Oracle proporciona gran número de paquetes con funcionalidades muy diversas. Se pueden consultar en:

  `https://docs.oracle.com/cd/E11882_01/appdev.112/e40758.pdf`
- Utilizaremos algunos paquetes muy básicos incluidos en el SGBD:
  - **DBMS_OUTPUT** permite escribir texto **para depuración:**

    ```
    DBMS_OUTPUT.PUT_LINE(texto); -- Escribe texto en la consola.
    ```

    - ★ PL/SQL no tiene interfaz de usuario por defecto.
    - ★ En **SQLDeveloper** se debe activar la escritura de texto en la consola con: **SET** SERVEROUTPUT **ON;**
  - **DBMS_RANDOM** para generar valores aleatorios. Procedimientos y funciones más relevantes:

    ```
    DBMS_RANDOM.SEED; -- Inicializa semilla.
    v := DBMS_RANDOM.VALUE; -- Devuelve valor entre 0 y 1.
    v := DBMS_RANDOM.VALUE(min,max); -- Devuelve valor en rango.
    v := DBMS_RANDOM.STRING(opt,len); -- Devuelve texto random.
    ```

SQL en PL/SQL

Acceso a los datos de la BD desde PL/SQL

- En lo que hemos visto hasta ahora, PL/SQL permite crear programas básicos de forma similar a cualquier otro lenguaje.
- Además PL/SQL **integra el lenguaje SQL** para poder acceder a los datos de la BD de forma sencilla.
- La comunicación de información entre PL/SQL y las tablas de BD se realiza mediante sentencias SQL y otras instrucciones específicas.

En PL/SQL se programa en dos niveles: procedural (PL/SQL) y de **acceso a datos** (sentencias SQL).

- Es importante tenerlo en cuenta para comunicar datos entre las variables locales PL/SQL (parte procedural) y las sentencias SQL (acceso a datos) y viceversa.

Comunicación PL $\Longrightarrow$ SQL

- Envío de datos del lenguaje procedural a SQL:

> Como regla general, **dentro de una sentencia SQL se pueden utilizar las variables locales del bloque.**

- **sentencias DML de modificación de datos:** En la sección de instrucciones de un bloque **se pueden intercalar sentencias `INSERT`, `UPDATE` y `DELETE`.**
- **ejemplo01b.sql:** Dada la tabla `piezas(cod NUMBER(5),descr VARCHAR2(30), precio NUMBER(11,2))`, podemos insertar datos en ella de la siguiente forma:

```
BEGIN
  FOR X IN 1..15 LOOP
    INSERT INTO piezas
    VALUES (X, 'pieza num: '||X, X*10);
  END LOOP;
END;
```

Dentro de la sentencia **`INSERT`** se utiliza la variable local **`X` con el valor que tenga en cada momento.**

Comunicación SQL $\Longrightarrow$ PL

- Recepción de datos desde SQL al lenguaje procedural:

  La comunicación desde tablas de la BD y variables PL/SQL **no es automática:** se deben utilizar **mecanismos específicos** basados en la sentencia **`SELECT`**.

- En un programa PL/SQL **no se pueden utilizar columnas de tablas fuera de las sentencias SQL y `%TYPE`.**
- En esta comunicación se produce lo que se denomina *"impedance mismatch"*:
  - **`SELECT`** devuelve un **conjunto de tuplas**.
  - Estos conjuntos pueden ser **demasiado grandes (millones de tuplas)** para mantenerlos en estructuras en memoria (listas, arrays, etc.)
- Hay dos mecanismos en PL/SQL para realizar esta comunicación:
  - Sentencias **`SELECT ... INTO`**.
  - **Cursores.**

Comunicación SQL $\Longrightarrow$ PL – SELECT ... INTO

- La sentencia **SELECT ... INTO** se puede utilizar cuando vamos a realizar **una consulta que devuelve una sola fila.**
- Es una sentencia **SELECT** con una cláusula **INTO** adicional.
- En la cláusula **INTO** se indican **las variables locales PL/SQL que contendrán el resultado** de la ejecución de la consulta SQL.
- El número y tipo de variables **debe coincidir con el resultado.**
- **ejemplo02.sql:** Consulta de una fila de la tabla `piezas`:

```
DECLARE
  v_cod piezas.cod%TYPE := 7;
  v_descr piezas.descr%TYPE;
  v_precio piezas.precio%TYPE;
BEGIN
  SELECT descr, precio INTO v_descr, v_precio
  FROM piezas WHERE cod = v_cod;
  DBMS_OUTPUT.PUT_LINE('Pieza : ' || v_cod || ' - ' || v_descr);
  DBMS_OUTPUT.PUT_LINE('Precio: ' || v_precio);
END;
```

Comunicación SQL $\Longrightarrow$ PL – Cursores

- Si una sentencia **SELECT ... INTO** devuelve más de una fila o no devuelve filas, **se produce una excepción.**
- Para recorrer y tratar una a una las filas resultantes de una sentencia **SELECT que devuelve varias filas** se utiliza un **cursor.**
- Un cursor es un **elemento del lenguaje PL/SQL** de tipo **cursor** que se asocia a una consulta. Se debe declarar en la sección de declaraciones:

```
DECLARE
  CURSOR cr_piezas IS
    SELECT cod, descr, precio FROM piezas WHERE precio > 100;
```

- El procesamiento de un cursor requiere realizar las siguientes operaciones en la sección de instrucciones:
  - Apertura del cursor: **OPEN nombre_cursor;**
  - Lectura de la siguiente fila del cursor: **FETCH nombre_cursor INTO lista_variables;**
  - Cierre del cursor: **CLOSE nombre_cursor;**

Cursores

- La instrucción **FETCH** recupera la siguiente fila de la consulta asociada y asigna los valores a la lista de variables.
- Las variables deben coincidir en número y tipo con el resultado de la consulta.
- Se puede comprobar el resultado de la lectura consultando los **atributos** del cursor:
  - **nombre_cursor %NOTFOUND** devuelve **cierto** si la última operación `FETCH` no devolvió ningún valor.
  - **nombre_cursor %FOUND** devuelve lo opuesto al anterior.
  - **nombre_cursor %ROWCOUNT** devuelve el número de filas que se han leído del cursor hasta el momento.
  - **nombre_cursor %ISOPEN** devuelve **cierto** si el cursor está abierto. Un cursor se puede cerrar y abrir de nuevo.
- Para facilitar el uso de cursores se pueden utilizar **variables de tipo registro.**

Cursores

- **ejemplo03.sql:** Ejemplo completo de recorrido de un cursor: escribe las piezas con precio mayor a 100.

```
DECLARE
  v_cod piezas.cod%TYPE;
  v_descr piezas.descr%TYPE;
  v_precio piezas.precio%TYPE;
  CURSOR cr_piezas IS
    SELECT cod, descr, precio FROM piezas WHERE precio > 100;
BEGIN
  OPEN cr_piezas;
  LOOP
    FETCH cr_piezas INTO v_cod, v_descr, v_precio;
    EXIT WHEN cr_piezas%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE(TO_CHAR(v_cod,'99999') || ' - ' ||
                         RPAD(v_descr,25) || ' ' ||
                         TO_CHAR(v_precio,'99G999D99'));
  END LOOP;
  CLOSE cr_piezas;
END;
```

Cursores y tipo de datos registro

- Es muy habitual seleccionar con **FETCH** o **SELECT ... INTO** una serie de columnas.
- Para evitar tener que declarar una variable para cada columna, se puede utilizar **una variable de tipo RECORD**.
- Se debe declarar asociada a una tabla o cursor con **%ROWTYPE**.
- **ejemplo04.sql:**

```
DECLARE
  CURSOR cr_piezas IS
    SELECT cod, descr, precio FROM piezas WHERE precio > 100;
  r_piezas cr_piezas%ROWTYPE;
BEGIN
  OPEN cr_piezas;
  LOOP
    FETCH cr_piezas INTO r_piezas;
    EXIT WHEN cr_piezas%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE(r_piezas.cod || ' - ' ||
        RPAD(r_piezas.descr,25) || ' ' || r_piezas.precio);
  END LOOP;
  CLOSE cr_piezas;
END;
```

Cursores: bucle FOR para cursores

- El uso de cursores y bucles para recorrerlos es muy habitual en PL/SQL.
- Por ello, **existe un tipo de bucle `FOR` específico para recorrer cursores que facilita su programación.**
- **`ejemplo05.sql`:** El bloque anterior puede ser mucho más corto:

```
DECLARE
  CURSOR cr_piezas IS
    SELECT cod, descr, precio FROM piezas WHERE precio > 100;
BEGIN
  FOR r_piezas IN cr_piezas LOOP
    DBMS_OUTPUT.PUT_LINE(TO_CHAR(r_piezas.cod,'99999') || ' - ' ||
                         RPAD(r_piezas.descr,25) || ' ' ||
                         TO_CHAR(r_piezas.precio,'99G999D99'));
  END LOOP;
END;
```

- No es necesario abrir, leer ni cerrar el cursor, se hace implícitamente en el bucle **`FOR`** de cursor.
- No es necesario declarar la variable de tipo registro.

Cursores para actualización de datos

- La sentencia SQL **UPDATE** se puede utilizar para modificar el contenido de la **fila actual de un cursor.**
- Para ello se debe indicar en el cursor añadiendo la cláusula: **FOR UPDATE OF campo**
- Además la sentencia **UPDATE** es especial para referenciar al cursor.
- **ejemplo06.sql:**

```
DECLARE
  CURSOR cr_piezas IS
    SELECT cod, descr, precio FROM piezas WHERE precio > 100
      FOR UPDATE OF precio;
BEGIN
  FOR r_piezas IN cr_piezas LOOP
    UPDATE piezas SET precio = precio * 0.95
    WHERE CURRENT OF cr_piezas;
  END LOOP;
  COMMIT;
END;
```

Procedimientos y funciones

Procedimientos y funciones

- Los **procedimientos** y **funciones** extienden los bloques anónimos con parámetros de entrada/salida y valor de retorno (funciones).
- Las funciones se pueden invocar en cualquier sitio donde se espera una expresión (p.ej. dentro de una consulta SQL).
- Se compilan para aumentar su eficiencia y se almacenan en la BD como cualquier otro objeto (tablas, índices, etc.): se pueden reutilizar.
- **ejemplo07.sql:**

```
CREATE OR REPLACE PROCEDURE proc1(p_param VARCHAR2) IS
  v_local VARCHAR2(50) := 'Mi primer procedimiento.';
BEGIN
  DBMS_OUTPUT.PUT_LINE(v_local || ' Parametro: ' || p_param);
END;
```

- Para probarlo se debe invocar desde dentro de otro bloque o procedimiento:

```
BEGIN
  proc1('Hola mundo!');
END;
```

Procedimientos y funciones: parámetros y declaraciones

- En los parámetros se deben indicar dos cosas:
  - El **tipo:** **VARCHAR2**, **NUMBER**, **INTEGER**. **No se debe indicar el tamaño.**
  - El **modo:** entrada **IN**, salida **OUT** o entrada-salida**IN OUT**. Por defecto **los parámetros son de entrada.**
- la sección de declaraciones de variables locales no debe contener la palabra clave **DECLARE**.
- **ejemplo08.sql:**

```
CREATE OR REPLACE PROCEDURE proc2(p_in IN VARCHAR2,
                                  p_out OUT VARCHAR2,
                                  p_inout IN OUT VARCHAR2) IS
BEGIN
  DBMS_OUTPUT.PUT_LINE('Entrada: p_in: ' || p_in);
  DBMS_OUTPUT.PUT_LINE('Entrada: p_out: ' || p_out);
  DBMS_OUTPUT.PUT_LINE('Entrada: p_inout: ' || p_inout);
-- p_in no puede modificar su valor porque es de entrada.
  p_out := 'este valor sale del procedimiento.';
  p_inout := 'este tambien.';
END;
```

Funciones: tipo y valor de retorno

- Las funciones devuelven un valor de retorno como resultado de la llamada.
- Se debe declarar **el tipo de retorno de la función** en el encabezamiento.
- Se debe incluir al menos una instrucción **RETURN** para devolver **un valor de retorno.**
- **ejemplo09.sql:**

```
CREATE OR REPLACE FUNCTION fun1(p_param VARCHAR2)
RETURN VARCHAR2 IS
BEGIN
  RETURN '***' || p_param || '***';
END;
```

- Se puede utilizar en cualquier contexto donde se espera una expresión del mismo tipo que el valor de retorno. Por ejemplo, dentro de una consulta SQL:

```
SELECT fun1(DESCR) FROM PIEZAS where precio > 100;
```

Excepciones

Excepciones

- Las **excepciones** son eventos que se producen durante la ejecución de un programa y que impiden su funcionamiento normal.
- Se pueden producir por diversos motivos:
  - Errores detectados por el entorno de ejecución de Oracle (por ejemplo, división por cero).
  - Situaciones anómalas (errores de acceso a disco o acceso a datos de la BD, comunicaciones, etc.)
  - Situaciones provocadas por el desarrollador en el programa.
- Normalmente, una excepción provoca la finalización del programa que la recibe.
- Pero se pueden capturar, utilizando una sección específica **EXCEPTION** al final del bloque, procedimiento o función.
- Esta sección está formada por sentencias

```
WHEN excepcion1 [OR excepcion2...] THEN
  instrucciones
```

Manejo de excepciones

- **ejemplo10.sql:**

```
CREATE OR REPLACE PROCEDURE manejo_excepciones IS
  v_cod piezas.cod%TYPE;
BEGIN
  SELECT cod INTO v_cod FROM piezas WHERE 1=2; --prueba con 1=1/0
  DBMS_OUTPUT.PUT_LINE('Todo bien.');
EXCEPTION
  WHEN TOO_MANY_ROWS THEN
    DBMS_OUTPUT.PUT_LINE('Consulta SELECT INTO devuelve varias
        filas.');
  WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE('Consulta SELECT INTO no devuelve ninguna
         fila.');
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('Otro error : ' || SQLCODE);
    DBMS_OUTPUT.PUT_LINE('con mensaje: ' || SQLERRM);
END;
```

- Para capturar todas las excepciones: **WHEN OTHERS THEN**. Debe ser la última cláusula `WHEN` del bloque.
- Se puede saber el código y texto del mensaje: **SQLCODE**, **SQLERRM**.

Tipos de excepciones

- **Internas:** Tienen un código asociado pero no identificador (no deberían capturarse de forma habitual).
- **Predefinidas:** Tienen un identificador asociado. Las más relevantes:

CURSOR_ALREADY_OPEN	Se intenta abrir un cursor abierto.
DUP_VAL_ON_INDEX	Se intenta insertar fila con clave duplicada.
INVALID_NUMBER	No se puede convertir texto a número.
NO_DATA_FOUND	`SELECT INTO` no devuelve filas.
ROWTYPE_MISMATCH	No se correponden las variables con las columnas de `SELECT INTO` o `FETCH`.
TOO_MANY_ROWS	`SELECT INTO` devuelve varias filas.
ZERO_DIVIDE	División por cero.

`https://docs.oracle.com/cd/E11882_01/appdev.112/e25519.pdf`

- **Definidas por el programador:** se pueden crear en el programa.
  - Se deben declarar con: **`nombre_exc EXCEPTION;`**
  - Se pueden lanzar con: **`RAISE nombre_exc;`**
  - **ejemplo11.sql.**

Disparadores

Disparadores

- En determinadas circunstancias se necesita ejecutar un determinado código cuando ocurre **un evento en la BD.**
  - Por ejemplo, cuando se debe incluir una **restricción de integridad** que no se puede incluir en el modelo relacional.
  - Para auditoría (registro de quién ha modificado qué tablas/datos).
- En Oracle (y otros SGBD) se pueden asociar fragmentos de código (**triggers** o **disparadores**) a determinados eventos de la BD.
- Hay tres tipos de eventos a los que se pueden asociar:
  - **Disparadores de tabla:** cuando se produce una modificación de los datos de una tabla.
  - **Disparadores de vista:** cuando se ejecuta una modificación sobre los datos de una vista.
  - **Disparadores de sistema:** cuando ocurre un evento del sistema (conexión de un usuario, borrado de un objeto, etc.)
- Veremos en detalle los disparadores de tabla.

Disparadores de tabla - conceptos básicos

- Es necesario fijar una serie de conceptos para definir un disparador:
  - **Cuándo se ejecuta el código del disparador** exactamente: antes o después de que se produzca el evento:
    - ★ **`BEFORE:` antes** de realizar la modificación en la tabla.
    - ★ **`AFTER:` después** de modificar la tabla.

    Los disparadores `BEFORE` pueden cambiar el valor de la fila a modificar en la tabla (`:NEW`).
  - **Evento** que provoca la ejecución del disparador: **`INSERT`**, **`UPDATE`**, **`DELETE`**.
  - **La tabla** cuya modificación va a activar la ejecución del disparador.
  - **Cuántas veces** se ejecuta el disparador:
    - ★ **Disparador de instrucción:** Se ejecuta una vez **por cada instrucción DML** que lo dispara (opción por defecto).
    - ★ **Disparador de fila:** Se ejecuta una vez **por cada fila que se ve afectada.** Se debe indicar con **`FOR EACH ROW`**.

Disparadores de tabla - conceptos básicos

- Formato de la instrucción de creación de un disparador de tabla:

```
CREATE [OR REPLACE] TRIGGER nombreDisparador
BEFORE|AFTER evento [OR evento [OR evento]]
ON tabla
[FOR EACH ROW [WHEN condicion]] -- Disparador de fila.
[DECLARE
  declaraciones_del_disparador]
BEGIN
  cuerpo_del_disparador
END;
```

- **BEFORE|AFTER**: cuándo se ejecuta el disparador.
- *evento* puede ser **INSERT**, **UPDATE** o **DELETE**.
- **ON** *tabla*: indica la tabla que, cuando se modifican datos en ella, provoca la ejecución del disparador.
- **FOR EACH ROW**: para crear un **disparador de fila.** Si se omite, es un **disparador de instrucción.**
- **DECLARE ... BEGIN ... END**: es el código del disparador, que se ejecutará cuando se modifiquen los datos de *tabla*.

Disparadores de tabla – disparador de instrucción

- Un disparador de instrucción se ejecuta una vez **por cada instrucción DML** que lo dispara.
  - Aunque la instrucción afecte a varias filas de la tabla.
- El código del disparador se ejecuta **aunque no haya ninguna fila afectada por la instrucción.**
- Es la opción por defecto al declarar un disparador (para ello NO debe indicarse `FOR EACH ROW`).
- Ejemplo mínimo de disparador de instrucción: **ejemplo12.sql**

Disparadores de tabla – disparador de fila

- Un disparador de fila se ejecuta una vez **por cada fila afectada en una instrucción DML**.
- Se especifica indicando **`FOR EACH ROW`**.
- **Un disparador de fila consume recursos:** puede tener que ejecutarse millones de veces para una sola instrucción DML...
  - Se debe evitar ejecutarlo sin necesidad.
  - Si el evento es **`UPDATE`**, se puede indicar la columna afectada por el disparador: **`UPDATE OF`** *`columna`*
    Solo se ejecuta el disparador si se modifica *`columna`*.
  - La cláusula **`WHEN`** permite precisar con detalle cuándo se ejecuta el disparador.
- Ejemplo de disparador de fila: **ejemplo13.sql**

Disparadores de tabla – elementos específicos de PL/SQL

- PL/SQL dispone de elementos del lenguaje específicos para programar disparadores. Veremos dos:
  - **Predicados** para identificar la operación realizada.
  - **Registros con los datos** de la fila modificada en disparadores de fila.

1. **Predicados de identificación de sentencia:** Si un disparador se ejecuta por varios eventos, podemos utilizar los siguientes predicados para identificar el evento que produjo su ejecución: **`INSERTING`**, **`UPDATING`** y **`DELETING`**. Por ejemplo:

```
IF INSERTING THEN ...
ELSIF UPDATING THEN ...
ELSIF DELETING THEN ...
END IF;
```

Ejemplo de predicados en un disparador de instrucción: **ejemplo14.sql**

Disparadores de tabla – elementos específicos de PL/SQL

2. **Registros con los datos de la fila modificada:** en el código de un **disparador de fila** se puede acceder a los datos de la fila cuya modificación produjo la ejecución del disparador.
   - Es posible acceder a los datos **antes** y **después** de la modificación.
   - Existen dos variables especiales de tipo registro que se denominan **:OLD** y **:NEW**, respectivamente.
   - Estas variables contienen los valores de las columnas de la fila que se está modificando **antes y después de ejecutar la instrucción de modificación** que provocó la ejecución del disparador.
   - **Example13.sql:**

```
CREATE OR REPLACE TRIGGER test
AFTER UPDATE ON parts FOR EACH ROW
BEGIN
  DBMS_OUTPUT.PUT_LINE('Actualización de la tabla piezas.');
  DBMS_OUTPUT.PUT_LINE(' Valor antiguo: '||:OLD.cod||'-'
                       ||:OLD.descr||'-'|| :OLD.precio);
  DBMS_OUTPUT.PUT_LINE(' Valor nuevo : '||:NEW.cod||'-'
                       ||:NEW.descr||'-'||:NEW.precio);
END;
```

Disparadores de tabla – Orden de ejecución de disparadores

- Se pueden crear varios disparadores asociados a una misma tabla.
- Estos disparadores se ejecutan en un orden determinado:
  1. Disparadores `BEFORE` de instrucción.
  2. Por cada fila: disparadores `BEFORE` de fila.
  3. Se modifica la fila.
  4. Por cada fila: disparadores `AFTER` de fila.
  5. Disparadores `AFTER` de instrucción.
- Otras operaciones sobre disparadores:

```
DROP TRIGGER nombreDisparador;              -- elimina disparador
ALTER TRIGGER nombreDisparador DISABLE;     -- desactiva disparador
ALTER TRIGGER nombreDisparador ENABLE;      -- reactiva disparador
ALTER TABLE tabla {DISABLE|ENABLE} ALL TRIGGERS;
-- desactiva/reactiva todos los disparadores asociados a una tabla
```