Introduction to PL/SQL stored procedures and triggers

Databases 2018-2019

Jesús Correas – jcorreas@ucm.es

Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid

Bibliography

- Basic Bibliography:
  - http://www.plsqltutorial.com/
  - **Oracle Database PL/SQL Language Reference 11g Release 2 (11.2).** E25519-13. https://docs.oracle.com/cd/E11882_01/appdev.112/e25519.pdf
- Additional Bibliography:
  - R. Elmasri, S.B. Navathe. **Fundamentals of Database Systems** (6th ed). Addison-Wesley, 2010. (in Spanish: **Fundamentos de Sistemas de Bases de Datos** (5a ed). Addison-Wesley, 2007).
    Chapter 13 (6th ed.)

DB applications programming

- **SQL** is a very powerful query language, but **it is not a programming language.**
- We will need a **procedural programming language** for programming complex operations on the DB.
- It is usual to access databases from **applications** (external programs) that **connect** to the DB manager system and **submit** SQL sentences for their execution:
  - Such programs are usually programmed in general-purpose programming languages (Java, C++, PHP, Javascript, etc.).
  - They access to the DB using libraries as JDBC, ODBC, ADO.NET.
  - They use a **client/server architecture:** each SQL sentence is sent through the network to be executed by the DBMS.
- This approach **is inappropriate** for some complex operations on the database:
  - It is **inefficient.**
  - The resulting program is very complex and error prone because of the **different approach** of SQL with respect to procedural languages.

Advantages of stored procedures

- Most current DBMS provide a **procedural programming language executing inside the DB manager.**
- With this language a series of DB operations can be grouped in a **block** that can be executed on the DB **with no need of communication with client application** for each DB operation.
- This language is designed to seamlessly intersperse SQL sentences with procedural operations (if-then-else, loops, etc.)
  - Reduces the *Impedance Mismatch* of SQL and a procedural language.
- Procedures and function definitions can be compiled and stored in the DB together with other DB elements (tables, indices, constraints).
  - It is more efficient and provides a high security level.
  - Reusable from different client applications.
- Procedures code is **optimizable** using DB statistics.
- The code is **portable** to any other DBMS installation (same provider).
- We can use it to program **triggers,** pieces of code that automatically execute in response to DB events.

Advantages of stored procedures

- Most DBMS provide a language to write stored procedures:
  - Oracle: **PL/SQL.**
  - MySQL: *Stored Procedure Support* (from version 5.0.0).
  - Microsoft: **TransactSQL.**
  - PostgreSQL: **PL/pgSQL.**
  - IBM DB2: **SQL Procedural Language.**
- **SQL:1999** and **SQL:2003** standards specify a language (SQL/PSM) for creating functions and procedures, but each DBMS implementation provides dialects not very compatible with the rest.
- All these languages are characterized by having **two levels of procedure representation:**
  - The procedural language.
  - The SQL data access language.
- Although the integration of these levels is facilitated in these languages, **we always have to consider the level in which every piece of code is located.**

PL/SQL language: blocks, functions, procedures, triggers

- The procedural language for Oracle is **PL/SQL.**
  - It is an imperative procedural language with **local variables, control structures, procedures and functions with parameters and exception handling.**
- There are **four basic types of code blocks in PL/SQL:**
  - **Anonymous block:** a code fragment without a name that is executed just once.
  - **Procedure:** a named code fragment that can have **parámetros:** input, output, or input/output.
  - **Function:** a named code fragment with parameters that returns a value when executed.
  - **Trigger:** a named code fragment that **automatically executes** when some event takes place, usually when some data in DB changes.
- All types of blocks except anonymous blocks are **compiled** and **stored** as DB objects in the database.
- In this course we will see a **brief introduction to this language.**

Anonymous PL/SQL blocks

- An anonymous block is declared and **executed only once**: **as it is unnamed, we cannot store it nor invoke it.**
- **example01.sql:** A simple example of anonymous PL/SQL block is:

```
-- example01.sql
DECLARE -- Declarations section
 varGreetings VARCHAR2(20);
BEGIN   -- Instructions section
 varGreetings := 'Hello World';
 DBMS_OUTPUT.PUT_LINE(varGreetings);
END;
/  -- In SQL*Plus and SQLDeveloper we must finish the block
   -- with a slash character to execute it
```

- We can **declare nested blocks**, just like in Java o C++.
- Blocks are composed of **sections**. The only mandatory section is the **instructions section**, enclosed by **BEGIN** and **END**.
- A block can have up to three sections: **declarations** (**DECLARE**), instructions and **exceptions** (**EXCEPTION**).

Declarations section

- It must start with the keyword **DECLARE**.
- We can declare variable names local to this block, as well as their types and initial value.
- Allowed types are the used for table columns as well as other types specific to PL/SQL. Most important basic types are:
  - **`VARCHAR2(n)`** variable-length text variable with up to `n` characters.
  - **`NUMBER(p,s)`** numeric Variable with `p` digits, `s` of them follow the decimal point.
  - **`INTEGER`** (16-bit integer variables), **`DATE`**, **`BOOLEAN`**, etc.
- We will see that we can declare variable of other types: **records**, *arrays*, **cursors** and user-defined exceptions.
- We can assign an initial value to variables in the declaration.
- We can declare constants using the keyword **CONSTANT.**
- Examples:

```
DECLARE
  name VARCHAR2(50) := 'Valor inicial.';
  greeting CONSTANT VARCHAR2(12) := 'Hello World!';
  v_amount NUMBER(12,2);
```

Declarations section: %TYPE

- We can declare variables (and parameters) whose type **is referenced to the type of a column in a table** or another variable.
- This way we do not need to look for the table description to define a local variable with the appropriate type.
- Furthermore, **if the column type changes**, we just have to recompile the PL/SQL block to have the variable updated to the new type.
- For example, if the table `Client` have the following definition:

```
CREATE TABLE Client(
  clientId VARCHAR2(9) PRIMARY KEY,
  name CHAR(35) NOT NULL,
  maxAmount NUMBER(12,2)
);
```

- We can declare PL/SQL variables as follows:

```
DECLARE
  v_client_id Client.clientId%TYPE;
  total_amount Client.maxAmount%TYPE := 0.0;
```

Instructions section: assignment

- The **instructions section** must start with the keyword **BEGIN**. This is the only mandatory section in an anonymous block.
- In this section we can include assignments **to local variables**, control-flow instructions and calls to procedures and functions, combined with other data access instructions (SQL and cursor handling).
- The **assignment** syntax is:

```
var := expression; -- var must be a local variable!
```

- *expression* may combine **local variables**, literals (numéricos, texto entre comillas simples) and function calls by means of operators:
  - Numeric: `+, -, *, /, **`
  - Text concatenation: `||`

Instructions section: conditional statements

- Syntax:

```
IF condition THEN
  instructions
END IF;
```

```
IF condition1 THEN
  instructions1
ELSIF condition2 THEN
  instructions2
...
ELSE
  instructions
END IF;
```

- A condition can be any Boolean expressión combining **local variables** and literals (numeric, text enclosed by single quotes) with operators:
  - Comparison: `<, <=, >, >=, =, !=`,
  - Boolean: `AND, OR, NOT`
  - NULL check: `expr IS [NOT] NULL`
- column names **are only allowed if used in cursors or records.**

Instructions section: Multiple selection

- Similar to **`switch`** sentence in C++ or Java.
- But in PL/SQL it is not an instruction, it is **an expression that returns a value.**
- Ejemplo:

```
grade_alpha := CASE calif
  WHEN NULL THEN 'not marked'
  WHEN 'SB' THEN 'Outstanding'
  WHEN 'NT' THEN 'Good'
  WHEN 'AP' THEN 'Satisfactory'
  WHEN 'SS' THEN 'Failure'
END;
```

- We can also use it with any Boolean expression, just like a composite **`IF`** statement:

```
calif_alpha := CASE
  WHEN calif >= 9 AND homework = 'OK' THEN 'Outstanding'
  WHEN calif < 9 AND calif >= 7 AND homework = 'OK' THEN 'Good'
  WHEN calif < 7 AND calif >= 5 THEN 'Satisfactory'
END;
```

Instructions section: loops

(remember that square brackets stand for **optional** syntax)

- General **LOOP** statement:

```
LOOP
  instructions
  EXIT [WHEN condition] -- this can be anywhere in the loop!
  instructions
END LOOP;
```

- **WHILE** statement:

```
WHILE condition LOOP
  instructions
END LOOP;
```

- Numeric **FOR** statement:

```
FOR variable IN [REVERSE] minValue..maxValue LOOP
  instructions
END LOOP;
```

- We will see later another variant of the `FOR` loop to **iterate through the resulting rows of an SQL query.**

Functions and procedures from external packages

- Although we will not see it in this course, we can define packages in PL/SQL to build code libraries.
- Oracle provides a large number of packages to be used from PLSQL code. The complete list can be consulted here:

  `https://docs.oracle.com/cd/E11882_01/appdev.112/e40758.pdf`
- In this course we will use few basic packages included in the DBMS:
  - **DBMS_OUTPUT** contains procedures for writing text for **debugging:**

    ```
    DBMS_OUTPUT.PUT_LINE(text); -- Writes a text on the console.
    ```

    - PL/SQL is used for server-side programming and does not have any user interface.
    - We can activate console output in **SQLDeveloper** with the command: `SET SERVEROUTPUT ON;`
  - **DBMS_RANDOM** contains code for random values generation, e.g.:

    ```
    DBMS_RANDOM.SEED; -- Initializes random seed.
    v := DBMS_RANDOM.VALUE; -- Returns a value between 0 y 1.
    v := DBMS_RANDOM.VALUE(min,max); -- Returns value in range.
    v := DBMS_RANDOM.STRING(opt,len); -- Returns random text.
    ```

SQL inside PL/SQL

Access to DB data from PL/SQL

- We have seen that we can create with PL/SQL basic programs just like any other programming language.
- Nevertheless, PL/SQL **includes the SQL language** for accessing DB data in a simple way.
- The communication between PL/SQL and DB tables is made by means of SQL sentences and other specific statements.

> **PL/SQL code is programmed at two different levels: a procedural level** (PL/SQL) and **a data access level** (SQL sentences).

- It is crucial to take this into account when communicating data between PL/SQL local variables (procedural level) and SQL sentences (data access level).

PL $\Longrightarrow$ SQL communication

- Sending data from the procedural language to SQL:

> As a general rule, **we can use PL/SQL local variables inside an SQL sentence embedded in the same block.**

- **DML data modification sentences:** We can include **`INSERT`, `UPDATE` and `DELETE` sentences** in the instructions section of a block.
- **example01b.sql:** Given the table `parts(cod NUMBER(5),descr VARCHAR2(30), price NUMBER(11,2))`, we can insert data as follows:

```
BEGIN
  FOR X IN 1..15 LOOP
    INSERT INTO parts
    VALUES (X, 'part nr: '||X, X*10);
  END LOOP;
END;
```

> Local variable **`X`** is used inside the **`INSERT`** sentence, **with the current value of the variable for each iteration.**

SQL $\Longrightarrow$ PL communication

- Receiving data from SQL to the procedural language:

> Communication from DB tables and PL/SQL local variables **is not automatic:** we must use **specific mechanisms** based on the `SELECT` sentence.

- In a PL/SQL block **we cannot use table columns outside SQL sentences (and `%TYPE` definitions).**
- This communication produces what is called *"impedance mismatch":*
  - `SELECT` returns a **set of tuples**.
  - These sets can be **too large (millions of tuples)** to be handled by memory-based data structures (lists, arrays, etc.)
- There are two ways to perform this communication:
  - `SELECT ... INTO` sentence.
  - **Cursors.**

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

- We can use **SELECT ... INTO** when we want to make **a query that returns a single row.**
- This is a **SELECT** sentence with an additional **INTO** clause.
- We use the **INTO** clause to set **the local variables that will contain the (single-row) query result.**
- Number and type of local variables **must match the query result.**
- **example02.sql:** Query of a single row in parts table:

```
DECLARE
  v_cod parts.cod%TYPE := 7;
  v_descr parts.descr%TYPE;
  v_price parts.price%TYPE;
BEGIN
  SELECT descr, price INTO v_descr, v_price
  FROM parts WHERE cod = v_cod;
  DBMS_OUTPUT.PUT_LINE('Part : ' || v_cod || ' - ' || v_descr);
  DBMS_OUTPUT.PUT_LINE('Price: ' || v_price);
END;
```

SQL $\Longrightarrow$ PL communication – Cursors

- If a **SELECT ... INTO** sentence returns more than one row or no rows, **an exception is raised.**
- We can traverse the multiple rows produced by a **SELECT** sentence and process them one row at a time by means of a **cursor.**
- A cursor is an **element of PL/SQL language** of type **cursor** that is linked to a query. It must be declared in the declarations section:

```
DECLARE
 CURSOR cr_parts IS
  SELECT cod, descr, price FROM parts WHERE price > 100;
```

- Processing a cursor requires the following operations in the instructions section:
  - Open the cursor: **OPEN nombre_cursor;**
  - Fetch the next row in the cursor:
    FETCH *cursorName* INTO *variablesList*;
  - Close the cursor: **CLOSE *cursorName*;**

Cursors

- The **`FETCH`** statement retrieves the next row of the query results and assigns its values to the variables in the list.
- List variables must match the name and types of the values returned in each row.
- We can check the fetch status using **cursor attributes:**
  - **`cursorName %NOTFOUND`** returns **`true`** if last `FETCH` operation did not return any value (i.e., there are no more rows to retrieve).
  - **`cursorName %FOUND`** returns the opposite to the previous attribute.
  - **`cursorName %ROWCOUNT`** returns the number of rows read so far.
  - **`cursorName %ISOPEN`** returns **`true`** if the cursor has been opened. A cursor can be closed and opened again.
- We can use **variables of record type** for handling cursors.

Cursors

- **example03.sql:** Cursor traversal: lists the parts data for those parts with price greater than 100.

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

Cursors and record data type

- In order to avoid the declaration of as many local variables as columns in cursors or `SELECT ... INTO`, **we can use a single variable of RECORD type.**
- We can declare it associated to a table or cursor using **%ROWTYPE:**
- **example04.sql:**

```
DECLARE
  CURSOR cr_parts IS
    SELECT cod, descr, price FROM parts WHERE price > 100;
  r_parts cr_parts%ROWTYPE;
BEGIN
  OPEN cr_parts;
  LOOP
    FETCH cr_parts INTO r_parts;
    EXIT WHEN cr_parts%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE(TO_CHAR(r_parts.cod,'99999') || ' - ' ||
                         RPAD(r_parts.descr,25) || ' ' ||
                         TO_CHAR(r_parts.price,'99G999D99'));
  END LOOP;
  CLOSE cr_parts;
END;
```

Cursor FOR LOOP

- Cursors and loops for traversing them are very common in PL/SQL programming.
- **There exists a specific `FOR LOOP` for traversing cursors.**
- **`example05.sql:`** The block seen in previous slide can be much shorter, as follows:

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

- The cursor **`FOR LOOP`** automatically opens the cursor, fetches it before each iteration, and closes it at the end.
- The record type variable `r_parts` does not need to be declared.

Cursors for updating data

- We can use the **UPDATE** sentence to modify the contents of the **current row of a cursor.**
- We have to define the cursor with an extra clause:
  FOR UPDATE OF column
- In addition, the **UPDATE** sentence must refer the cursor as in the following example:
- **example06.sql:**

```
DECLARE
  CURSOR cr_parts IS
    SELECT cod, descr, price FROM parts WHERE price > 100
     FOR UPDATE OF price;
BEGIN
  FOR r_parts IN cr_parts LOOP
    UPDATE parts SET price = price * 0.95
    WHERE CURRENT OF cr_parts;
  END LOOP;
  COMMIT;
END;
```

Procedures and functions

Procedures and functions

- **Procedures** and **Functions** extend anonymous blocks with input/output parameters and a return value (functions).
- Functions can be invoked anywhere an expression is expected (e.g. inside an SQl query).
- Are compiled to improve their efficiency and are stored in the DB just like any other DB object (tables, indices, etc.): they can be reused once and again.
- **example07.sql:**

```
CREATE OR REPLACE PROCEDURE proc1(p_param VARCHAR2) IS
 v_local VARCHAR2(50) := 'My first procedure.';
BEGIN
 DBMS_OUTPUT.PUT_LINE(v_local || ' Parameter: ' || p_param);
END;
```

- To execute it we have to invoke it from another block or procedure:

```
BEGIN
 proc1('Hello World!');
END;
```

Procedures and functions: parameters and declarations

- Parameters are defined providing two pieces of information:
  - **Type: VARCHAR2, NUMBER, INTEGER. without the type length.**
  - **Mode:** input **IN**, output **OUT** or input-output**IN OUT**. (`IN` by default)
- The section of local variables does not start with the keyword **DECLARE**, **IS** keyword must be used instead.
- **example08.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('Input: p_in: ' || p_in);
  DBMS_OUTPUT.PUT_LINE('Input: p_out: ' || p_out);
  DBMS_OUTPUT.PUT_LINE('Entrada: p_inout: ' || p_inout);
  -- p_in cannot change its value as it is an input parameter.
  p_out := 'this is an output value.';
  p_inout := 'this is also an output value.';
END;
```

Functions: type and return value

- A function returns a value as the result of its invocation.
- **The return type of the function** must be declared in the function header.
- A function must contain at least one **RETURN** statement to **return a value.**
- **example09.sql:**

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

- A function can be used in any context where an expression (of the same type) is expected. For example, inside a SQL query:

```
SELECT fun1(Descr) FROM Parts where Price > 100;
```

Exceptions

Exceptions

- **Exceptions** are events that are thrown during the execution of a program and that prevent a program from resuming execution.
- Exceptions can be thrown due to several causes:
  - Errors detected by the Oracle execution environment (e.g. division by zero).
  - Anomalous conditions (disk access or DB access errors, network failures, etc.)
  - Conditions caused by the programmer.
- An exception thrown usually causes the program termination.
- The program can capture exceptions using a specific section **EXCEPTION** at the end of the block, procedure or function.
- This section must contain statements of the form:

```
WHEN exception1 [OR exception2...] THEN
  instructions
```

Exception handling

- **example10.sql:**

```
CREATE OR REPLACE PROCEDURE excepHandling IS
  v_cod Parts.cod%TYPE;
BEGIN
  SELECT cod INTO v_cod FROM Parts WHERE 1=2;
    -- Test exceptions: change WHERE condition to 1=1 or to 1=1/0
  DBMS_OUTPUT.PUT_LINE('everything OK.');
EXCEPTION
  WHEN TOO_MANY_ROWS THEN
    DBMS_OUTPUT.PUT_LINE('SELECT INTO returns several rows.');
  WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE('SELECT INTO returns no rows.');
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE('Another error : ' || SQLCODE);
    DBMS_OUTPUT.PUT_LINE('Error message : ' || SQLERRM);
END;
```

- For capturing all any other exception: **WHEN OTHERS THEN**. It must be the last `WHEN` clause in the block.
- We can get the error code and message: **SQLCODE**, **SQLERRM**.

Types of exceptions

- **Internal:** Have a code but no identifier (they should not be captured.)
- **Predefined:** Have an identifier. The most relevant ones are:

CURSOR_ALREADY_OPEN	Tried to open an already open cursor.
DUP_VAL_ON_INDEX	Duplicate primary/unique key value.
INVALID_NUMBER	It is not possible to convert texto to number.
NO_DATA_FOUND	`SELECT INTO` does not return any row.
ROWTYPE_MISMATCH	Variables do not match columns in a `SELECT INTO` or `FETCH`.
TOO_MANY_ROWS	`SELECT INTO` returns more than one row.
ZERO_DIVIDE	Division by zero.

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

- **User-defined:** Are created by the programmer in the program code.
  - Must be declared with: **`nombre_exc EXCEPTION;`**
  - Can be thrown with: **`RAISE nombre_exc;`**
  - **example11.sql.**

Triggers

Triggers

- There are some cases in which we require to execute some code when **a DB event takes place.**
  - For example, when we have to add an **integrity constraint** that cannot be represented by the relational model, or a **semantic contraint.**
  - For auditing or access control (recording who has modified specific data on the DB).
- Oracle (and almost all DBMS) can associate code fragments (**triggers**) to specific events of the DB.
- There are three types of events to associate triggers:
  - **Table triggers:** when the data in a table is modified (inserted, deleted, updated).
  - **View triggers:** when the data in a view is modified.
  - **System triggers:** when a system event takes place (user logs in, deletion of a DB object, etc.)
- We will **see table triggers in more detail.**

Table triggers - basic concepts

- It is important to set some concepts before defining a trigger:
  - **When is the trigger code executed:** just *before* or just *after* the event takes place:
    - **`BEFORE:`** The table data has not been modified yet; such data can be modified by the trigger before making the change on the table (changing `:NEW` record).
    - **`AFTER:`** the table has been already modified.

    When a trigger is executed it is said that *"the trigger fires."*
  - Which is the **event** that produces the trigger execution: **`INSERT`**, **`UPDATE`**, **`DELETE`**.
  - What is **the table** whose modification will produce the trigger execution.
  - **How many times** the trigger is executed:
    - A **statement-level trigger** is executed once **for each DML sentence** that fires it (default case).
    - A **row-level trigger** is executed once **for each row affected by a DML sentence** that fires the trigger.

Table triggers - basic concepts

- Syntax of a table trigger creation sentence:

```
CREATE [OR REPLACE] TRIGGER triggerName
BEFORE|AFTER event [OR event [OR event]]
ON table
[FOR EACH ROW [WHEN condition]] -- Row-level trigger.
[DECLARE
  trigger declarations]
BEGIN
  trigger body
END;
```

- **BEFORE|AFTER**: When the trigger fires.
- *event* may be one of: **INSERT**, **DELETE** or **UPDATE [OF *columns*]**
- **ON** *table*: table that makes the trigger fire when its data is modified.
- **FOR EACH ROW**: used to create **row-level triggers.** If omitted, it is a **statement-level trigger** by default.
- **DECLARE ... BEGIN ... END**: Trigger code.

Table triggers – Statement-level trigger

- A statement-level trigger executes once **for each DML sentence** that makes the trigger fire.
  - The trigger is executed once even though the DML sentence affects several rows in the table.
  - The trigger is executed **even though there are no rows affected by the DML sentence.**
- This is the default trigger type when declaring it (we have to OMIT the keywords `FOR EACH ROW`).
- Minimal example of a statement-level trigger: **example12.sql**

Table triggers – Row-level trigger

- A row-level trigger fires once **for each row affected by a DML sentence**.
- We can define a row-level trigger adding the clause **`FOR EACH ROW`**.
- **A row-level trigger is costly:** if a DML sentence changes many rows (millions), the trigger will fire the same number of times...
  - We should adjust it to avoid executing it unnecessarily.
  - If it is associated to an **`UPDATE`** event, we can indicate the column affected by the trigger: **`UPDATE OF`** *`column`*
    The trigger will fire only if *`column`* changes.
  - The **`WHEN`** clause allows us to specify more accurately under what condition the trigger fires.
- Example of row-level trigger: **example13.sql**

Table triggers – PL/SQL-specific elements

- PL/SQL has some language elements specific to trigger programming. We will see two of them:
  - We can use some **predicates** to know which was the DB operation that fired the trigger.
  - We can access a **record containing the data** of the particular row whose change fired a row-level trigger.

1. **Statement identification predicates:** If a trigger fires due to several modification events of a table, we can use the following predicates in the trigger for checking which event produced its execution: **`INSERTING`**, **`UPDATING`** and **`DELETING`**. For example:

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

Example of statement identification predicates in a statement-level trigger: **example14.sql**

Table triggers – PL/SQL-specific elements

2. **Records with the data of the modified row:** in the code of a **row-level trigger** we can access the data of the row whose modification produced the execution of the trigger.
   - We can access the data **before** and **after** the modification.
   - There are two special variables of record type named **:OLD** and **:NEW**, respectively.
   - These variables contain the column values of the row being modified **just before (:OLD) and just after (:NEW)** the modification of the row that made the trigger fire.
   - **Example13.sql:**

```
CREATE OR REPLACE TRIGGER test
AFTER UPDATE ON parts FOR EACH ROW
BEGIN
   DBMS_OUTPUT.PUT_LINE('Updating a row of parts table.');
   DBMS_OUTPUT.PUT_LINE(' Old value: ' || :OLD.cod || '-'
                        || :OLD.descr || '-' || :OLD.price);
   DBMS_OUTPUT.PUT_LINE(' New value: ' || :NEW.cod || '-'
                        || :NEW.descr || '-' || :NEW.price);
END;
```

Table triggers – Trigger execution order

- We can create several triggers associated to the same table.
- The triggers execute in a specific order:
  1. Statement-level `BEFORE` triggers.
  2. For each row: row-level `BEFORE` triggers.
  3. The row is modified.
  4. For each row: row-level `AFTER` triggers.
  5. Statement-level `AFTER` triggers.
- Other operations on triggers:

```
DROP TRIGGER triggerName;           -- removes trigger
ALTER TRIGGER triggerName DISABLE; -- deactivates trigger
ALTER TRIGGER triggerName ENABLE;  -- reactivates trigger
ALTER TABLE table {DISABLE|ENABLE} ALL TRIGGERS;
-- deactivates/reactivates all triggers associated to a table.
```