3.6. Subprograms

Subprograms (procedures and functions) available since Oracle 7, unlike anonymous blocks, are blocks with a name and can be stored in the database as database objects. Subprograms stored in the database can be referenced from other blocks (subprograms) and functions, additionally (since Oracle 7.3), can be referenced from the DML commands and the SELECT command. The functions stored in the database can therefore extend the capabilities of SQL language.

The procedure block is defined according to the syntax:

PROCEDURE procedure_name [({parameter [, ...]})]
[**AUTHID {<u>DEFINER</u> | CURRENT_USER**}]
{**IS | AS**}
 [-- *definitions of the PL/SQL objects for the procedure*]
BEGIN
  -- *executable part of the procedure*
 [**EXCEPTION**
  -- *exception handling*]
END [procedure_name];

The function block is defined according to the syntax:

FUNCTION function_name [({parameter [, …]})]
          RETURN return_type
[**AUTHID** {**<u>DEFINER</u> | CURRENT_USER**}]
{**IS | AS**}
  [-- *definitions of the PL/SQL objects for the function*]
BEGIN
  -- *executable part of the function with at least one*
  -- *command:* RETURN expression;
[**EXCEPTION**
  -- *exception handling*]
END [function_name];

The RETURN command without expression can also occur in the executable part of the procedure. Using this form of command causes the procedure to stop and pass control to the calling program.

A subprogram defined in this way can be an internal subprogram (DECLARE section) of another block (subprogram) or an element of the package (the package is a library of objects grouped under one name such as types, procedures, functions, variables, constants, cursors and exceptions).

The AUTHID clause specifies the rights with which the subprogram will be run (rights of user, who define subprogram or rights of user who call it - the default are the first rights). The parameter in both definitions has the syntax:

parameter_name [{<u>**IN**</u> | **OUT** [**NOCOPY**] | **IN OUT** [**NOCOPY**]}]
type [{:=|**DEFAULT**} initial_value];

Subprogram parameters, if they occur, play the role of formal parameters. The type of formal parameter (also defined by the %TYPE attribute) and the return type specified in the function definition means any predefined or defined type (without length!). Formal parameters can have one of three modes:

– IN - read-only parameter to which cannot be assigned a new value in the subprogram (this is the default mode),
– OUT - a write-only parameter which in a subprogram gets value (the value from the subprogram call is ignored) and which cannot be read,
– IN OUT - parameter for reading and writing (features of the IN and OUT modes).

Parameters in IN mode are passed by reference (indicator) and in OUT and IN OUT modes by value. The NOCOPY modifier (since Oracle 8i) causes the compiler to attempt to pass a parameter by reference rather than by value. This modifier is ignored if:

- actual parameter is an element of the index table,
- the type of the actual parameter has a length limit (however, this does not apply to parameters of character types), accuracy or the parameter has a NOT NULL limit,
- actual and formal parameters are records implicitly declared as control variables of the FOR loop or are declared using the %ROWTYPE pseudo-attribute, and the constraints of the corresponding fields in the records differ from each other,
- auto-conversion of type will occur when passing actual parameters.

The NOCOPY modifier is mainly used to speed up the transfer of large tables. In addition, it allows the parameter values determined in the subprogram to be retained in the event of an error (normally in such a situation, for passing by the value, the actual parameter retains the value of before the call).

The subprogram is called according to the syntax:

```
subprogram_name(argument [, ...])
```

Call arguments act as actual parameters. They can be specified in **positional notation** (suitability of actual and formal parameters) or in **name notation** (any order of actual parameters). In name notation, the parameter is specified according to the syntax:

```
formal_parameter_name=>actual_value
```

The following is an example of a procedure header and examples of calling it (correct and incorrect).

```
PROCEDURE something(order NUMBER, volume NUMBER:=450,
                    donor VARCHAR2:='Honey',
                    recipient VARCHAR2:='Drunkard');

something;                                   -- incorrect
something(221);                              -- correct
something(221,'Mite');                       -- incorrect
something(221,recipient=>'Mite');            -- correct
something(order=>221,recipient=>'Mite');     -- correct
```

The subprogram is saved as a database object using the DDL command of SQL language, according to the syntax:

CREATE [**OR REPLACE**] subprogram_block_definition;

Attention!!! Oracle also saves in the database a subprogram with compilation errors. A corresponding warning will then appear on the screen. The SHOW ERRORS command displays a list of errors.

A subprogram stored in the database can be called from within any PL/SQL block. The procedure stored in the database (preceded by the username) in SQL Developer environment is called by EXECUTE command. The stored function can be called as part of a DML command or as part of a SELECT command. In the event of such a call it must meet the following conditions:

- commands in function body cannot modify data of the database,
- function parameters must be passed in IN mode,
- function parameters and parameter returned by function must be of the simple type.

In addition, a DML command cannot use a function that works on relation whose attributes this DML command modifies.

Task. *Save in the database the function determining the minimum mice ration in a band specified by the function parameter and then use the defined function in the SELECT query and the DML command.*

```
CREATE OR REPLACE FUNCTION min_ration(bno NUMBER)
RETURN NUMBER
AS
  minr Cats.mice_ration%TYPE;
BEGIN
  SELECT MIN(mice_ration) INTO minr
  FROM Cats WHERE band_no=bno;
  RETURN minr;
EXCEPTION
  WHEN NO_DATA_FOUND THEN NULL;
  WHEN OTHERS THEN DBMS_OUTPUT.PUT_LINE(SQLERRM);
END;

FUNCTION min_ration compiled

SELECT nickname
FROM Cats
WHERE mice_ration=min_ration(2);

NICKNAME
---------------
MISS

UPDATE Cats
SET mice_ration=min_ration(3)
WHERE band_no=2;

ORA-04091: the Z.CATS table is mutating, the trigger/function
may not see it

ORA-06512: at "Z.MIN_RATION", line 12
Error report:
SQL Error: 06503. 00000 -  "PL/SQL: Function returned without
           value"
```

The defined function was called without a problem as part of the SELECT command, while an error occurred during the UPDATE command. It results from the mentioned prohibition of using in the DML command a function which works on relation whose attributes the DML command modifies - the `min_ration` function operates on the `Cats` relation which is modified by the UPDATE command

The subprogram is removed from the database using the DDL command with the syntax:

DROP {**PROCEDURE** | **FUNCTION**} subprogram_name;

A list of all user subprograms can be found in the USER_OBJECTS system view and their content in the USER_SOURCE view.

Subprograms in PL / SQL can be called recursively.

Task. *Determine recursively all superiors of the selected cat.*

```
CREATE OR REPLACE
FUNCTION superiors_rec(cat_nickname Cats.nickname%TYPE)
RETURN VARCHAR2
AS
  chief_ni Cats.chief%TYPE; chief_na Cats.name%TYPE;
BEGIN
  SELECT C1.chief,C2.name INTO chief_ni,chief_na
  FROM Cats C1,Cats C2
  WHERE C1.chief=C2.nickname AND C1.nickname=cat_nickname;
  DBMS_OUTPUT.PUT('Nickname of chief: ');
  DBMS_OUTPUT.PUT(RPAD(chief_ni,10));
  DBMS_OUTPUT.PUT_LINE(' Name: '||RPAD(chief_na,10));
  RETURN superiors_rec(chief_ni);
END superiors_rec;

FUNCTION superiors_rec compiled

DECLARE
  ni Cats.nickname%TYPE:='&nickname';
  cat_ni Cats.nickname%TYPE;
  cat_na Cats.name%TYPE;
BEGIN
  SELECT name INTO cat_na
  FROM Cats
  WHERE nickname=ni;
  DBMS_OUTPUT.PUT('Nickname of cat:  ');
  DBMS_OUTPUT.PUT(RPAD(ni,10));
  DBMS_OUTPUT.PUT_LINE(' Name: '||RPAD(cat_na,10));
  cat_ni:=superiors_rec(ni);
EXCEPTION
  WHEN NO_DATA_FOUND THEN DBMS_OUTPUT.PUT_LINE('The end');
END;

anonymous block completed

Nickname of cat:   ZERO       Name: LUCEK
Nickname of chief: HEN        Name: PUNIA
Nickname of chief: ZOMBIES    Name: KOREK
Nickname of chief: TIGER      Name: MRUCZEK
The end
```

3.7. Packages

PL/SQL gives the possibility of grouping certain objects (types, procedures, functions, variables, constants, cursors and exceptions) into libraries called packages. The package is another PL/SQL block saved as a database object. The package includes:

- specification **-** it contains declarations and some definitions (constants, types, cursors) of objects provided by the package (this is the interface of the package),
- body **-** it contains the package object definitions as well as declarations and definitions of internal body objects available only for the package.

The package specification is defined according to the syntax:

CREATE [**OR REPLACE**] **PACKAGE** package_name
[**AUTHID {DEFINER | CURRENT_USER**}]
{**IS** | **AS**}
/* *definitions of constants, types, cursors, subprograms declarations (their headers), variables, exceptions available from outside the package* */
END [package_name];

The package body is defined according to the syntax:

CREATE [**OR REPLACE**] **PACKAGE BODY** package_name
{**IS** | **AS**}
/* *definitions and declarations of objects, local for the package* */
/* *definitions of subprograms declared in the package specification* */
[**BEGIN**
/* *optional block initializing package variables at the time of the first reference to the package* */]
END [package_name];

The package body is an optional element (it does not appear if the specification does not contain subprogram declarations). In the event that a package subprogram refers to another package subprogram in the package body, then that subprogram must be defined before the subprogram from which it is referenced. If for some reason this is not possible, declare the called subroutine in the form of its header placed before the definition of the calling subprogram (early declaration). One can reference to the package objects according to the syntax:

package_name.package_object_name

Subprograms inside a package can be overloaded, i.e. there can be more than one procedure or function with the same name but different parameters. Overloading of subprograms hast the following restrictions:

- two subprograms may not be overloaded if their parameters differ only in name or passing mode,
- one cannot overload two functions that differ only in the type of the returned value,
- one cannot overload two functions for which parameters have types from the same family (e.g. CHAR and VARCHAR2).

If the SQL command or the PL / SQL block uses functions contained in the package, then the system, up to and including Oracle 8.1, cannot check the condition for their use (i.e. lack information of modification of the content of the relation by the function commands - only the package specification is available). Since Oracle 9i, this restriction only applies to package functions intended to be called from the block. In this case, in the package specification, after the function header, include relevant information about the so-called level of purity of the function. The compiler directive RESTRICT_REFERENCES is used for this purpose. Its syntax is as follows:

PRAGMA RESTRICT_REFERENCES(function_name ,
WNDS [, **WNPS**][, **RNDS**][, **RNPS**][,**TRUST**]);

This directive applies only to functions because only functions can be performed from SQL level. The meaning of parameters determining the level of function purity is presented below.

Parameter	Description
WNDS	The function does not modify the content of the relation (using the DML command).
WNPS	The function does not modify the values of package variables (package variables are not used by the assignment operator or by the FETCH command).
RNDS	The function does not read the content of the relation (using the SELECT command).
RNPS	The function does not read the values of package variables (package variables are not used on the right side of the assignment operator or as part of an SQL or PL/SQL expression).
TRUST	The function can call other functions with an unspecified level of purity.

Task. *Create a package containing two functions, one determining the minimum mice ration for a band specified by parameter, the other determining the average mice ration for a band specified by parameter. Use the package's functions to find cats whose mice ration is greater than the average ration in their bands, displaying additionally the difference between their mice ration and the minimum ration in their band.*

```
CREATE OR REPLACE PACKAGE package_of_functions AS
  FUNCTION min_ration(bno NUMBER) RETURN NUMBER;
  FUNCTION avg_ration(bno NUMBER) RETURN NUMBER;
END package_of_functions;

PACKAGE package_of_functions compiled
```

```
CREATE OR REPLACE PACKAGE BODY package_of_functions AS
  FUNCTION min_ration(bno NUMBER) RETURN NUMBER
  IS
    mr Cats.mice_ration%TYPE;
  BEGIN
    SELECT MIN(NVL(mice_ration,0)) INTO mr FROM Cats
    WHERE band_no=bno;
    RETURN mr;
  END min_ration;
  FUNCTION avg_ration(bno NUMBER) RETURN NUMBER
  IS
    ar NUMBER(10,3);
  BEGIN
    SELECT AVG(NVL(mice_ration,0)) INTO ar FROM Cats
    WHERE band_no=bno;
    RETURN ar;
  END avg_ration;
END package_of_functions;

PACKAGE BODY package_of_functions compiled


SELECT name "Name",band_no "Band No",NVL(mice_ration,0)-
             package_of_functions.min_ration(band_no) "Excess"
FROM Cats
WHERE mice_ration>package_of_functions.avg_ration(band_no)
ORDER BY band_no;

Name            Band No                Excess
--------------- ---------------------- ----------------------
MRUCZEK         1                      81
JACEK           2                      43
ZUZIA           2                      41
BOLEK           2                      48
PUNIA           3                      41
KOREK           3                      55
MELA            4                      11
PUCEK           4                      25
KSAWERY         4                      11

 9 rows selected
```

The package's functions were defined for their use from SQL level, so it was not necessary to specify their purity level.

3.8. Triggers

Triggers are another named block saved in the database as a database object. Unlike explicitly executed by calling subprograms, triggers are executed implicitly when a specific trigger event occurs. This event may relate to DML operations on relation (Oracle 7.0 and above), DML operations via the view (Oracle 8.0 and above), DDL operations as well as database events such as login (Oracle 8.1 and above). The trigger is defined according to the syntax:

CREATE [**OR REPLACE**] **TRIGGER** trigger_name
{**BEFORE** | **AFTER**} | **INSTEAD OF** triggering_event
ON {relations_name | **DATABASE**} | view_name
[**FOR EACH ROW**]
[**FOLLOWS** trigger_name]
[**WHEN** trigger_condition]
{ PL/SQL block | **CALL** procedure};

The trigger content size cannot exceed 32K. For a larger trigger, one can reduce its volume by moving some of the code in the form of subprograms to the package. Triggers can have the same names as subprograms or relations (they have a different namespace).

Below Oracle 8i version, the trigger content could only be a PL/SQL block. Since Oracle 8i, its content can also be a procedure stored in the database (not necessarily written in PL/SQL!), called with the CALL command.

The BEFORE type triggers are activated before the triggering event (DML operations on relations, DDL operations and database events), the AFTER type after the triggering event. The INSTEAD OF type triggers are performed instead of DML (trigger event) operations on the view.

There are the following DML trigger events for BEFORE | AFTER ... ON relation_name type trigger and for the INSTEAD OF ... ON view_name type trigger:

- INSERT - inserting a new row directly in the relation or indirectly in the relation (relations) *via* view,
- UPDATE [OF attribute_list] - modification of attribute values directly in relation or indirectly in relation (relations) *via* view (attribute_list defines the attributes whose modification activate the trigger; this list is not available for the INSTEAD OF trigger),
- DELETE - removing rows directly in a relation or indirectly in a relation (relations) *via* view.

There are the following DDL triggering events for BEFORE and AFTER ... ON DATABASE triggers:

- CREATE - creating a new database object,
- ALTER - change in an existing database object,
- DROP - removing the database object.

There are the following database triggering events for type triggers BEFORE AFTER ... ON DATABASE:

- SERVERERROR - appearing of server error message (type AFTER only),
- LOGON - user logging in (only AFTER type),
- LOGOFF - user logging out (only BEFORE type),
- STARTUP - opening the database (only AFTER type),
- SHUTDOWN - closing the database (only BEFORE type).

Trigger events can be combined using an OR logical operator.

The FOR EACH ROW clause applies only to triggers activated by the DML command and is placed when the trigger is to be activated for each modified row (so-called row trigger). Otherwise (no clause) the trigger is triggered only once, regardless of the number of lines modified by the DML command (so-called command trigger). The INSTEAD OF trigger is always a row trigger, hence the FOR EACH ROW clause over there does not exist.

The FOLLOWS clause, introduced since Oracle 11g version, allows the user to indicate a DML trigger of the same type for a given table, which is to be executed before the trigger just defined. In lower versions of Oracle this order was undefined.

The WHEN clause allows one to define an additional condition (in parentheses and without subqueries!) That limits the number of trigger calls. For row DML triggers (i.e. those with the FOR EACH ROW clause or INSTEAD OF triggers), this clause allows definition of the condition for selecting the rows whose modification will activate the trigger. Such triggers (row triggers, no other!) can additionally use two qualifiers in the WHEN clause and in their body, i.e. NEW and OLD (in the body of the trigger they must be preceded by a colon - :). They provide access to the new (corrected) and old value of the attribute, which is modified by DML command. This access is implemented in accordance with the syntax:

[**:**]**NEW**.attribute_name [**:**]**OLD**. attribute_name

An additional qualifier PARENT has been introduced since Oracle 8i version. It applies to triggers activated by a modification of the so-called nested table (this type of table will be discussed when presenting of the object extensions of Oracle).

For the INSERT and DELETE commands, the OLD and NEW qualifiers are NULL, respectively.

Triggers activated by DDL commands and database events (named together system triggers) have the following restrictions on the kind of condition in the WHEN clause:

- no conditions can be specified for STARTUP and SHUTDOWN triggers,
- for SERVERERROR triggers, one can only use the `ERRNO` variable specifying the server error number,
- for LOGON and LOGOFF triggers only the password and username can be checked (`ORA_DES_ENCRYPTED_PASSWORD` and `ORA_LOGIN_USER`).

– for DDL triggers, one can only check the object type and name (`ORA_DICT_OBJ_TYPE` and `ORA_DICT_OBJ_NAME`) as well as the password and username.

Creating system triggers is only possible with ADMINISTER DATABASE TRIGGER privileges.

Task. *Tiger decided to protect himself from removing him from the herd through deleting od his band (if the ON DELETE CASCADE restriction will be additionally defined for the foreign key `band_no` in the relation `Cats`). He also decided additionally to secure his henchman, Bald (band No 2). Define the trigger activated by deleting the row in the `Bands` relation, which implementing these protections.*

```
CREATE OR REPLACE TRIGGER or_removing_band
BEFORE DELETE ON Bands
FOR EACH ROW WHEN (OLD.band_no IN (1,2))
DECLARE
  how_many_members NUMBER(3):=0;
BEGIN
      SELECT COUNT(*) INTO how_many_members
      FROM Cats
  WHERE band_no=:OLD.band_no;
  IF how_many_members>0 THEN
     RAISE_APPLICATION_ERROR(-20105,
     'Band '||:OLD.name||' with members is irremovable!');
  END IF;
END;

TRIGGER or_removing_band compiled

DELETE FROM Bands WHERE band_no=1;

Error report:
SQL Error: ORA-20105: Band SUPERIORS with members is
           irremovable!

DELETE FROM Bands WHERE band_no=5;

1 rows deleted.

ROLLBACK;

rollback complete.;
```

Trigger has blocked the deletion of band 1. The band 5 has been deleted without any problem. The defined trigger is only meaningful if in CREATE TABLE command the ON DELETE CASCADE constraint has been defined for the foreign key `band_no` in the `Cats` relation. Otherwise, the bands will be protected by the reference constraint - error "`ORA-02292: integrity constraint (Z.CA_BNO_FK) violated - child record found`".

Task. *Using the `Kittens` view, which choose `nickname`, `name`, `in_herd_since`, `chief` from the relation `Cats` and `name` from the relation `Bands`, add a new cat to the relation `Cats`.*

```
CREATE OR REPLACE VIEW  Kittens AS
SELECT nickname,C.name cat_name,in_herd_since,
       B.name band_name,chief
FROM Cats C JOIN Bands B USING(band_no);
```

```
view KITTENS created.
```

```
CREATE OR REPLACE TRIGGER new_cat
INSTEAD OF INSERT ON Kittens
DECLARE
  bn NUMBER; l NUMBER;
BEGIN
  SELECT COUNT(*) INTO l FROM Bands WHERE name=:NEW.band_name;
  IF l=0
     THEN RAISE_APPLICATION_ERROR(-20001,'Invalid band name');
  END IF;
  SELECT band_no INTO bn FROM Bands
  WHERE name=:NEW.band_name;
  IF :NEW.in_herd_since>SYSDATE
    THEN RAISE_APPLICATION_ERROR(-20002,'Date above current');
  END IF;
  SELECT COUNT(*) INTO l FROM Cats
  WHERE nickname=:NEW.nickname;
  IF l=1
     THEN RAISE_APPLICATION_ERROR(-20003,'Existing nickname');
  END IF;
  SELECT COUNT(*) INTO l FROM Cats
  WHERE chief=:NEW.chief;
  IF l=0
    THEN RAISE_APPLICATION_ERROR(-20004,'Non-existent chief');
  END IF;
  INSERT INTO Cats (nickname,name,in_herd_since,band_no,chief)
  VALUES (:NEW.nickname,:NEW.cat_name,:NEW.in_herd_since,bn,
          :NEW.chief);
END;
```

```
TRIGGER new_cat compiled
```

```
ALTER SESSION SET NLS_DATE_FORMAT='YYYY-MM-DD';
session SET altered.

INSERT INTO Kittens
VALUES ('FAT','RYCHO','2020-04-17','BLACK KNIGHTS','BALD');
1 rows inserted.

ROLLBACK;
rollback complete.

INSERT INTO Kittens
VALUES ('TIGER','RYCHO','2020-04-17','BLACK KNIGHTS','BALD');
Error report:
SQL Error: ORA-20003: Existing nickname!
```

The INSTEAD OF trigger enabled modification relation `Cats` through the view `Kittens`, even though it is a non-modifiable view. Such a trigger is characterized by the fact that the operation causing its "firing" is ignored and replaced by the operation defined in his body.

System trigger events, i.e. DDL events and database events, have attributes that can be read within the trigger body. These attributes are:

– DICTIONARY_OBJ_NAME - the name of the database object used in the DDL command,
– DICTIONARY_OBJ_TYPE - type of object in the DDL command,
– DICTIONARY_OBJ_OWNER - owner of the object whose name was used in the DDL command,
– IS_ALTER_COLUMN (attribute IN VARCHAR2) - TRUE if the definition of the attribute has been changed,
– IS_DROP_COLUMN (attribute IN VARCHAR2) - TRUE if the attribute has been removed,
– IS_SERVERERROR (error_number IN NUMBER) - TRUE if an error with the given number occurred,
– LOGIN_USER - the name of the user whose action activated the trigger,
– SYSEVENT - name of the event whose occurrence triggered the trigger,
– CLIENT_IP_ADRES - client computer IP address.

Task. Task. *Define a trigger to monitor in the relation* `Events` *instances of DDL commands*

```
CREATE TABLE Events
(command VARCHAR2(10),
 user_name VARCHAR2(15),
 event_date DATE,
 object_type VARCHAR2(10),
 object_name VARCHAR2(14));

table EVENTS created.

CREATE OR REPLACE TRIGGER event_description
BEFORE CREATE OR ALTER OR DROP ON DATABASE
DECLARE
  com Events.command%TYPE;
  usn Events.user_name%TYPE;
  evd Events.event_date%TYPE;
  obt Events.object_type%TYPE;
  obn Events.object_name%TYPE;
BEGIN
  com:=SYSEVENT;
  usn:=LOGIN_USER;
  evd:=SYSDATE;
  obt:=DICTIONARY_OBJ_TYPE;
  obn:=DICTIONARY_OBJ_NAME;
  INSERT INTO Events VALUES (com,usn,evd,obt,obn);
END;

CREATE TABLE New_table(kolumn NUMBER);
table NEW_TABLE created.

SELECT * FROM Events;

COMMAND    USER_NAME  EVENT_DATE    OBJECT_TYPE OBJECT_NAME
---------- ---------- ------------- ----------- ------------
CREATE     Z          2020-04-18    TABLE       NEW_TABLE
```

When several different DML triggers, triggered by different events, need to be defined for one relation, it can be done within one trigger, the content of which will be divided into sections that will be activated separately upon the occurrence of a specific event. The INSERTING, UPDATING, DELETING predicates placed in the IF statement are used for this.

```
CREATE OR REPLACE TRIGGER something
BEFORE INSERT OR UPDATE OR DELETE ON Cats
BEGIN
  ...
 IF INSERTING THEN ...
    ...
 END IF;
  ...
 IF UPDATING THEN ...
    ...
 END IF;
 -- IF UPDATING('mice_ration') OR
 --    UPDATING('mice_extra') THEN ...
 --    ...
 -- END IF;
  ...
 IF DELETING THEN ...
    ...
 END IF
  ...
END;
```

3.8.1. Locking and removing triggers

Once defined, the trigger is normally ready for operation (unlocked). Locking or unlocking the trigger is carried out in accordance with the syntax:

ALTER TRIGGER trigger_name {**DISABLE | ENABLE**};

All triggers associated with a specific relation can be locked or unlocked according to the syntax:

ALTER TABLE relation_name {**DISABLE ALL TRIGGERS |**
ENABLE ALL TRIGGERS};

The trigger is removed with the command:

DROP TRIGGER trigger_name;

3.8.2. Restrictions on the use of DML triggers

In Oracle 7.0 version, only one command trigger and only one row trigger of each type: BEFORE INSERT, BEFORE UPDATE, BEFORE DELETE, AFTER INSERT, AFTER UPDATE, AFTER DELETE can be associated with a given relation. As of Oracle 7.3, multiple triggers of the same type can be associated with a relation. Additionally, triggers have the following restrictions:

- row triggers and all triggers triggered indirectly as a result of the ON DELETE CASCADE constraint or ON DELETE SET NULL constraint cannot read or change the content of the modified relation (mutating table). As modified relation is understood as the relation on which the action activates the trigger or the relation referring to the relation on which the action activates the trigger, all through the above-mentioned restrictions. This does not apply to INSTEAD OF triggers,
- triggers cannot execute DDL commands or DCL commands. The exceptions here are triggers with so-called autonomous transaction (available from Oracle 8i),
- no LONG or LONG RAW type variables can be declared in the trigger content. Also, OLD and NEW qualifiers cannot refer to attributes of these types if they are specified in the relation for which the trigger is defined,
- below Oracle 8 version one cannot refer in triggers to attributes of LOB (Large Objects) type. From Oracle 8 version one can do this, but cannot modify their value.

The only cases of row triggers that can read or modify a relation on which the action activates the trigger are BEFORE and AFTER triggers for an INSERT statement inserting one row only (e.g., not for the statement INSERT ALL or for the statement INSERT INTO relation_name SELECT... , even if it return only one row).

3.8.3. The order of execution of DML triggers

In the case where at least two triggers of different type related to the one relation are activated by DML events, the order of their execution is important. It is as follows:

1. The BEFORE command trigger (if any) is executed,
2. For each row, row trigger BEFORE is executed (if any),
3. The command activating the trigger is executed,
4. For each row, row trigger AFTER is executed (if any),
5. The AFTER command trigger (if any) is executed.

Up to Oracle 11g version, the order of execution of the same type of DML triggers associated with one relation was indefinite. From the Oracle 11g version, this is resulted by the presented earlier optional clause FOLLOWS of trigger header.

3.8.4. COMPOUND TRIGGER

Since the Oracle 11g version, actions related to one relation, implemented so far by many separate DML triggers (command or row triggers, in BEFORE and AFTER mode) can be defined in a one trigger named compound trigger. The big advantage of this solution is the ability to access, through each of these actions, to shared data stored in the local variables of the trigger. In earlier versions of Oracle, subsequent triggers could share data stored in variables, in defined for this purpose, specification of package.

There are two basic situations in which compound trigger can be used:

1. Preparation of data for mass processing (mass binding - this issue will be presented later in the lecture).
2. To avoid error ORA-04091: mutating-table error.

The compound trigger syntax is shown below.

```
CREATE [OR REPLACE TRIGGER] trigger_name
FOR DML_triggering_event
ON relations_name | view_name
[FOLLOWS trigger_name]
[WHEN trigger_condition]
COMPOUND TRIGGER
   [-- definitions and declarations of the PL/SQL objects for the trigger]

   BEFORE STATEMENT IS
   [-- definitions and declarations of the PL/SQL objects for the section]
   BEGIN
     -- sentences of the section implementing the command part BEFORE
   [EXCEPTION
  -- section exception handling sentences]
   END BEFORE STATEMENT;

   BEFORE EACH ROW IS
   [-- definitions and declarations of the PL/SQL objects for the section]
   BEGIN
     -- sentences of the section implementing the row part BEFORE
   [EXCEPTION
     -- section exception handling sentences]
   END BEFORE EACH ROW;

   AFTER EACH ROW IS
   [-- definitions and declarations of the PL/SQL objects for the section]
   BEGIN
     -- sentences of the section implementing the row part AFTER
   [EXCEPTION
     -- section exception handling sentences]
   END AFTER EACH ROW;
```

```
 AFTER STATEMENT IS
   [-- definitions and declarations of the PL/SQL objects for the section]
   BEGIN
     -- sentences of the section implementing the command part AFTER
   [EXCEPTION
     -- section exception handling sentences]
   END AFTER STATEMENT;

END trigger_name;
```

The trigger is activated by the indicated type of DML event (INSERT, DELETE or UPDATE [OF attribute_list]; events can be combined using the logical operator OR) on the indicated relation or view. It contains an optional declarative part in which data, among others, can be stored, shared by all sections of the trigger. Trigger sections perform the following actions: command part BEFORE, row part BEFORE, row part AFTER and command part AFTER. As part of the row sections one can use the qualifiers :OLD, :NEW and :PARENT. Sections can only appear in the order determined by above syntax. A trigger must contain at least one of these sections.

The use of the compound trigger has some limitations.

1. A trigger can only be activated by DML commands on a relation or view.
2. The trigger cannot contain a autonomous transaction (PRAGMA AUTONOMOUS_TRANSACTION directive in the declarative part - see the next section).
3. Exceptions must be handled within the section in which they occur.
4. Jumping (GOTO command) can only take place within a specific section.
5. Value :NEW can only be modified in the BEFORE EACH ROW section.
6. After a DML exception occurs, within one of the sections, the values of local variables of the sections are re-initialized (their previous values are lost), however, modifications made earlier are not rollback.

An example of a compound trigger that performs data preparation for mass processing will be presented in the part of the lecture on mass binding, while an example illustrating how to avoid the error *ORA-04091 error: mutating-table error* will be the subject of one of the tasks on project lists.

3.8.5. Triggers with an autonomous transaction

Since Oracle 8.1 version it is possible to use the so-called autonomous transaction. Such a transaction is an independent transaction embedded in the main transaction, performed during the suspended main transaction. After its completion, the main transaction continues. An autonomous transaction must always be completed (committed or rolled back) otherwise the exception `"ORA-06519: active autonomous transaction detected and rolled back"` will occur. Rollback of the main transaction does not affect the autonomous transaction. An autonomous transaction is defined using a compiler directive:

PRAGMA AUTONOMOUS_TRANSACTION;

Task. *Tiger decided to remember the history of mice ration changes (also such changes which was not committed) in the Change_history relation. Define a trigger that monitors each such change.*

```
CREATE TABLE Change_history
(change_no NUMBER(5),
 for_whom VARCHAR2(15),
 date_of_change DATE,
 ration NUMBER(5),
 extra NUMBER(5));

table CHANGE_HISTORY created.

CREATE SEQUENCE no_in_history;
sequence NO_IN_HISTORY created.
```

```
CREATE OR REPLACE TRIGGER about_mice
BEFORE INSERT OR UPDATE OF mice_ration,mice_extra
ON Cats FOR EACH ROW
DECLARE
  ni Cats.nickname%TYPE;
  mr Cats.mice_ration%TYPE;
  me Cats.mice_extra%TYPE;
  PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
  IF INSERTING
     THEN ni:=:NEW.nickname; mr:=:NEW.mice_ration;
          me:=:NEW.mice_extra;
     ELSE ni:=:OLD.nickname;
  END IF;
  IF UPDATING('mice_ration')
     THEN mr:=:NEW.mice_ration;
     ELSIF NOT INSERTING THEN mr:=:OLD.mice_ration;
  END IF;
  IF UPDATING('mice_extra')
     THEN me:=:NEW.mice_extra;
     ELSIF NOT INSERTING THEN me:=:OLD.mice_extra;
  END IF;
  INSERT INTO Change_history
         VALUES (no_in_history.NEXTVAL,ni,SYSDATE,mr,me);
  COMMIT;
END;

TRIGGER about_mice compiled

UPDATE Cats  SET mice_extra=50 WHERE nickname='LOLA';
1 rows updated.

ROLLBACK;
rollback complete.

CHANGE_NO FOR_WHOM    DATE_OF_CHANGE   RATION      EXTRA
--------- ----------- ---------------- ----------- -----------
1         LOLA        2020-04-20       25          50
```

By using the autonomous transaction, it was possible to roll back row modifications in the `Cats` relation without at the same time rollback changes in the `Change_history` relation (prohibited COMMIT operation for other triggers but required in triggers with autonomous).

Triggers in which autonomous transaction is defined can, in addition to DCL operations, perform DDL operations that are prohibited for other triggers.

Task. *Define a trigger creating a new database user (without permissions) in the form of a new member of the cat herd.*

```
CREATE OR REPLACE TRIGGER new_user
BEFORE INSERT ON Cats FOR EACH ROW
DECLARE
  PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
  EXECUTE IMMEDIATE 'CREATE USER '||:NEW.nickname||
                    ' IDENTIFIED BY '||:NEW.nickname;
  EXECUTE IMMEDIATE 'GRANT CREATE SESSION TO '||:NEW.nickname;
END;

TRIGGER new_user compiled

INSERT INTO
Cats(nickname,name,in_herd_since,chief,mice_ration,mice_extra)
VALUES ('FAT','RYCHO','2020-04-20','BALD',50,10);

1 rows inserted.

connect FAT/FAT;
Connected
```

The EXECUTE IMMEDIATE command used in the trigger is an element of the so-called internal (native) dynamic SQL. In this case, it the first command creates a new user and the second command gives him permissions to create a session. All this is specified by string expressions (the DDL command and the DCL command, both prohibited for other trigger are here executed!).

3.9. Internal dynamic SQL

SQL is called dynamic if the full command is not defined until program running. Only then is command created in the form of a string expression. This expression can contain any SQL statements, including also those not available in PL/SQL blocks. In Oracle there are two implementations of dynamic SQL: uncomfortable to use, although having a lot of capabilities DBMS_SQL package and introduced since Oracle 8.1. native dynamic SQL.

The basic command of native dynamic SQL is EXECUTE IMMEDIATE, which executes any SQL command (also a PL/SQL block) written as a string. This command has two forms:

EXECUTE IMMEDIATE string_expression_SQL_command
[[**INTO** {variable [, ...]}][**USING** {bound_argument [, ..]}]];

EXECUTE IMMEDIATE string_expression_PL/SQL_block
[**USING** {bound_argument [, ..]}];

The first command applies to dynamic SQL, the second to dynamic PL/SQL. The string defining the dynamic PL/SQL block must end with a semicolon (;). If a string defining dynamic SQL command is terminated with a semicolon, it will be treated as a PL/SQL block. A string expression (in a special case a variable or string constant) defines any SQL query. The INTO clause applies to the dynamic version of the SELECT statement in a PL/SQL block (this clause cannot be part of a SELECT statement defined as a string). The USING clause defines the so-called bound arguments. The corresponding to them variables, which de facto act as formal parameters of the dynamic query, are part of a string expression. For identification they are preceded there by a colon (:) and the binding takes place in the order in which they occur in the chain. They are passed in the IN, OUT, IN OUT modes known from subprograms (default IN). These arguments must have of the types allowed in SQL, not in PL/SQL, and must have names different from database object names.

Task. *The Tiger's thoughts were overcome by a conspiracy vision of the world. The conspiracy was to be established among cats who are old citizens of the European Union, and was to consist of forced exchange (under the cover of trade) of healthy Polish mice for artificially "driven", European ones. To remedy the danger, Tiger decided to set up members of his herd secret mice accounts where some of the hunted mice would be stored. Write a block implementing this task. On start, transfer to account of each cat a number of mice proportional to his position in the herd.*

```
DECLARE
  CURSOR kitties
  IS SELECT level,nickname
     FROM Cats
     START WITH chief IS NULL
     CONNECT BY PRIOR nickname=chief;
  dyn_string VARCHAR2(1000);
  maxl NUMBER(2):=0;
  how_many NUMBER(4);
BEGIN
  FOR ki IN kitties
  LOOP
    IF ki.level>maxl THEN maxl:=ki.level; END IF;
    SELECT COUNT(*) INTO how_many
    FROM USER_TABLES
    WHERE table_name=ki.nickname;
    IF how_many=1 THEN
       EXECUTE IMMEDIATE 'DROP TABLE '||ki.nickname;
    END IF;
    dyn_string:='CREATE TABLE '||ki.nickname||'
                (entry_date DATE,release_date DATE)';
    EXECUTE IMMEDIATE dyn_string;
  END LOOP;
  FOR ki IN kitties
  LOOP
    dyn_string:='INSERT INTO '||ki.nickname||
                ' (entry_date) VALUES (:en_da)';
    FOR i IN 1..maxl-ki.level+1
    LOOP
    EXECUTE IMMEDIATE dyn_string USING SYSDATE;
    END LOOP;
  END LOOP;
  FOR ki IN kitties
  LOOP
    dyn_string:='SELECT COUNT(*)-COUNT(release_date) FROM '
                ||ki.nickname;
    EXECUTE IMMEDIATE dyn_string INTO how_many;
    DBMS_OUTPUT.PUT_LINE(RPAD(ki.nickname,10)||
                ' - Number of available mice: '||how_many);
  END LOOP;
END;

anonymous block completed
```

```
TIGER      - Number of available mice: 4
BALD       - Number of available mice: 3
CAKE       - Number of available mice: 2
FAST       - Number of available mice: 2
MISS       - Number of available mice: 2
TUBE       - Number of available mice: 2
BOLEK      - Number of available mice: 3
LITTLE     - Number of available mice: 3
LOLA       - Number of available mice: 3
REEF       - Number of available mice: 3
EAR        - Number of available mice: 2
LADY       - Number of available mice: 2
MAN        - Number of available mice: 2
SMALL      - Number of available mice: 2
ZOMBIES    - Number of available mice: 3
FLUFFY     - Number of available mice: 2
HEN        - Number of available mice: 2
ZERO       - Number of available mice: 1
```

Task. *Fear of conspiracy caused the herd leader ordered that his nickname was secret. He did it so effectively that he finally forgot his nickname himself. Write a block that executes a dynamic PL/SQL block that finding the nickname of any cat based on his name.*

```
DECLARE
  na Cats.name%TYPE:='&cat_nmae';
  co NUMBER(2);
BEGIN
  SELECT COUNT(*) INTO co FROM Cats WHERE name=na;
  IF co=0 THEN
     RAISE_APPLICATION_ERROR(-20105,'Wrong name!');
  END IF;
  EXECUTE IMMEDIATE
    'DECLARE
       CURSOR namesakes IS
       SELECT nickname FROM Cats WHERE name=:na;
     BEGIN
       FOR i IN namesakes
       LOOP
         DBMS_OUTPUT.PUT_LINE
                   ('||'''Nickname - '''||'||i.nickname);
       END LOOP;
     END;'
  USING na;
END;
```

```
CAT_NAME - MRUCZEK
```

```
anonymous block completed
```

```
Nickname - TIGER
```

In static SQL, SELECT commands returning multi-row relations were supported either through declaring an explicit cursor or through cursor variables. Internal dynamic SQL uses a cursor variable whose value (SELECT command) is defined as a string. The new syntax element is here the USING clause in the OPEN command, with a list of bound arguments. The syntax for this command is:

OPEN cursor_variable **FOR** string_expression

The string expression defines the query SELECT of the cursor.

Task. *Tiger came to the conclusion that it is worth (as part of protection against conspiracy) to hide some mice rations by reducing the number of additional mice (mice_extra) taken into account in the function which displaying statistics of monthly consumption. The number of additional mice consumed by each cat included in the official statement would be equal to half the average value of additional consumption. Write a block displaying, for selected cats, modified mice ration.*

```
CREATE OR REPLACE PACKAGE cursor AS
  TYPE c IS REF CURSOR;
END cursor;

PACKAGE cursor compiled

CREATE OR REPLACE
FUNCTION about_cats(addition NUMBER,
                    WHEREcondition VARCHAR2)
RETURN cursor.c
AS
  cur cursor.c;
  query VARCHAR2(1000);
  BEGIN
    query:='SELECT nickname,NVL(mice_ration,0)+NVL(:do,0)
            FROM Cats WHERE '||WHEREcondition;
   OPEN cur FOR query
   USING addition;
   RETURN cur;
END about_cats;
```

```
DECLARE
  condition VARCHAR2(500):='&condition';
  rescur cursor.c;
  ad NUMBER(3);
  ni VARCHAR(15);
  cons NUMBER(3);
BEGIN
  SELECT ROUND(AVG(NVL(mice_ration,0))/2,0) INTO ad
  FROM Cats;
  rescur:=about_cats(ad,condition);
  DBMS_OUTPUT.PUT_LINE('Mouse consumption by selected cats');
  LOOP
    FETCH rescur INTO ni,cons;
    EXIT WHEN rescur%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE('  '||' Cat '||ni||' eats '||cons);
  END LOOP;
  CLOSE rescur;
EXCEPTION
  WHEN OTHERS THEN DBMS_OUTPUT.PUT_LINE(SQLERRM);
END;

CONDITION - function=''CAT''

anonymous block completed

Mouse consumption by selected cats
   Cat ZERO eats 69
   Cat EAR eats 66
   Cat SMALL eats 66

CONDITION - mice_ration>400

anonymous block completed

Mouse consumption by selected cats
   Cat CAKE eats 93
   Cat TUBE eats 82
   Cat ZERO eats 69
   Cat TIGER eats 129
   Cat BOLEK eats 76
   Cat ZOMBIES eats 101
   Cat BALD eats 98
   Cat FAST eats 91
   Cat REEF eats 91
   Cat HEN eats 87
   Cat MAN eats 77
   Cat LADY eats 77
```