3. PL/SQL language

PL/SQL (Procedural Language /SQL) is a procedural extension of the SQL language proposed by Oracle that allows the use of SQL commands in the structure of blocks constituting a transaction programming tool. As part of programs of this language, besides SQL commands, it is possible to use structures from procedural languages such as:

- variables and data types (predefined as well as user-defined),
- control structures such as conditional instructions and loop instructions,
- procedures and functions,
- methods and object types.

There were the following important reasons for extending the capabilities of SQL language:

- data manipulation carried out by SQL is not the only task of handling the database (additional tasks are carried out at the application level),
- business rules (also those regarding data security) controlled at the application level can be bypassed by using another database access tool. This exposes the data to intentional or accidental modifications,
- each SQL query requires a separate connection to the database server which increases the network load (it would be better to send many queries during one connection).

PL/SQL, by possibility of defining subprograms and packages stored in the database, enables "shifting" many tasks related to the broadly understood database service to the database server, reducing the client application to a typical screen interface. This allows faster access to data increasing, at the same time, their security. Sending multiple SQL commands within one PL/SQL block reduces network load and leads to faster application operation.

3.1 Basic information and concepts

The following is a set of introductory information and concepts for programming in PL/SQL.

3.1.1 Block structure of PL/SQL program

The PL/SQL program consists of one or more blocks. Blocks can be independent or nested one in another. There are two types of blocks: anonymous block and named block.

Anonymous block: unnamed PL/SQL block, declared in such the place of application where it will be executed.

The anonymous block is usually passed from a client-side program, to call subprograms stored in the database.

Structure anonymous block is as follows:

[**DECLARE**
-- *definitions and declarations of PL/SQL objects for a block*]
BEGIN
    -- *sentences of the executable part of the block*
[**EXCEPTION**
-- *exception handling sentences*]
END;
/

Definitions of objects for block, sentences of executable part (there must be at least one sentence here) and exception handling sentences are separated by a semicolon. The block terminates its operation, when all sentences of the executable part are done or an exceptional situation (error) occurs, handled or not in the exception handling part. Internal blocks may occur as part of the executable part as well as the part of exception handling. In the executable part, internal blocks are usually used to handle exceptions that are not to end the operation of the external block, and in the part of handle exceptions to handle exceptions to exceptions. Each object defined in a given block is available only in this block (and thus in its internal blocks). In the case of defining objects with the same names in the external block and internal blocks, the principle of covering names applies (for the time of operation of the internal block, the object defined in the external block is not available).

Named block: PL/SQL block to which name is assigned. This name one can use in PL/SQL program.

There are three types of named blocks:

– labeled block: labeled block: this is the anonymous block to which its label has been assigned, as the name. This name allows one to refer to the block variables from the internal block if in internal block exist variable of the same name,
– subprogram block: it is a procedure or function that can be explicitly called (by name) from each type of block. The subprogram can be stored in a database,
– trigger block: it is a block is stored in the database implicitly executed when it occurs event specified in trigger definition.

The structure of the labeled block is the same as the structure of the anonymous block. The structure of the subprogram and the trigger differ practically from the structure of the anonymous block only by the occurrence of the header. Therefore, anonymous blocks will be discussed first.

3.1.2. Displaying diagnostic messages on the screen

To display diagnostic messages from the PL/SQL block level, one can use the procedures from the DBMS_OUTPUT package named PUT_LINE, PUT and NEW_LINE. Procedure PUT_LINE places in the buffer a message with a maximum length of 255 (with a sign of go to a new line), PUT a message without going to a new line - this transition is explicitly performed by the NEW_LINE function. In order for the messages entered into the buffer to appear on the screen, in the SQL Developer environment should select the Dbms Output element in the View tab.

The PUT_LINE and PUT procedures are called according to the syntax:

PUT_LINE(message [, VARCHAR2|NUMBER|DATE]);
PUT(message [, VARCHAR2|NUMBER|DATE]);

Calling of procedures from packages is preceded by the package name, e.g .:

```
DBMS_OUTPUT.PUT_LINE('Wiva TIGER, Lord of Lords!!!');
```

The DBMS_OUTPUT package functions are only used to test the operation of PL/SQL blocks. The real output of information to the screen takes place as part of the on-screen interface of the database application.

Task. *Define an anonymous block inserting new rows into the `Cats` relation through the previously defined `Band4` view. In the case of entering incorrect data (e.g. a repeating cat name - a unique index should be put on the cat name) appropriate messages ought to appear on the screen.*

```
CREATE UNIQUE INDEX unique_name ON Cats(name);

unique index UNIQUE_NAME created.


BEGIN
 INSERT INTO Band4 VALUES ('&nickname','&name','&function',
                             &mice_ration,&band_no);
 COMMIT;
EXCEPTION
 WHEN DUP_VAL_ON_INDEX
 THEN DBMS_OUTPUT.PUT_LINE('Repeated pseudo or name!!! -
                            NO ENTRY!');
 WHEN OTHERS
      THEN DBMS_OUTPUT.PUT_LINE(SQLERRM);
END;

nickname - YUPITER
name - MRUCZEK
function - CAT
mice_ration - 40
band_no - 4

Repeated pseudo or name!!! - NO ENTRY!
```

The & sign before the variable means that its value is to be entered from the keyboard. The WHEN ... THEN command handles an exception. The exception is listed (its name) after WHEN and is handled as specified after THEN. DUP_VAL_ON_INDEX is a predefined exception indicating a violation of a unique index (this also applies to the nickname as the key attribute - the index is automatically applied to the primary key and uniqueness is a feature of this key). OTHERS specifies any exceptions other to those previously mentioned. SQLERRM is a function that displays a system message about the error (about exception).

3.1.3 Environment for PL/SQL

PL/SQL blocks are processed by the PL/SQL machine residing in the DBMS or in the utility program. If the block is called from the level of the application created via the utility with the PL/SQL machine implemented, the block is processed by this machine (execution on the client side), otherwise the block is processed by PL/SQL machine residing in SZBD (server-side execution). In both cases, however, the PL/SQL machine performs only procedural orders and sends SQL orders to the executor of SQL orders in the DBMS.

3.1.4 Identifiers in PL/SQL

The identifier in PL/SQL begins with a letter, followed by any sequence of characters consisting of letters, numbers, and characters '$', '_', and '#'. The identifier declared in apostrophes ("identifier") may contain any characters. The identifier can be up to 30 characters long.

3.1.5 Variables and constants

Within the PL/SQL blocks it is possible to declare the following types of variables:

- scalar: types known from the Oracle SQL dialect,
- complex: record type, array types called collections (index tables up to Oracle 7 version, nested tables and arrays of variable size available since Oracle 8 version, multi-level collections available since Oracle 9i version - collections collections),
- references (pointers: REF CURSOR available up to Oracle 7 version, REF object_type available since Oracle 8 version),
- LOB: BFILE, CLOB, NCLOB, BLOB (available through the DBMS_LOB package since Oracle 8 version, support large binary or character objects up to 4 GB without restrictions specific to LONG and LONG RAW, e.g. such a restriction as this, that only one attribute of this type may appear in relation or lack of possibility manipulation on them using triggers).

The variable is declared according to the syntax:

Variable_name type [[**NOT NULL]** **{DEFAULT** | :=} expression];

If NOT NULL appears in the declaration, it is mandatory to specify the default value.

A very often used mechanism is to declare a variable with a type compatible with the type of relation attribute. This is done in accordance with the syntax:

variable_name relations_name.attribute**%TYPE**;

The constant within the PL / SQL block is defined in accordance with the syntax:

constant_name **CONSTANT** type := expression;

The following are sample variable declarations and definitions of constants.

```
Surname   VARCHAR2(25):='Kowalski';
name      VARCHAR2(15):='Jan';
initial   VARCHAR2(4) :=
              SUBSTR(nazwisko,1,1)||'.'||SUBSTR(imie,1,1)||'.';
counter   INTEGER NOT NULL:=0;
pesel     NUMBER(11);
kids      BOOLEAN:=FALSE;
end       BOOLEAN;
date      DATE;
nick      Cats.nickname%TYPE;
pi        CONSTANT NUMBER(9,5):=3.14159;
```

3.1.6. Assignment operation

The assignment operation within the PL/SQL block is performed according to the syntax:

variable_name:=PL/SQL_expression;

As part of the PL/SQL expression, one can use all the functions known from SQL (except GRATEST, LEAST, DECODE and group functions), and additionally, if the expression occurs within the EXCEPTION section of the block, one can use the functions SQLCODE (returns the exception number) and SQLERRM (returns exception message including its number).

The following are sample assignments.

```
counter:=counter+1;
data:=counter||'. '||initial||' '||pesel;  -- auto-conversion
kids:=NOT kids;
end:=counter=100;
```

Operators in the PL/SQL expression are executed in a different order than in the SQL expression. The order of execution (priority) of operators in PL/SQL is presented below.

1. **, NOT
+, - (as number signs)
2. *, /
3. +, - , ||
4. =, !=, <, >, <=, >=, IS NULL, LIKE, BETWEEN, IN
5. AND
6. OR

For logical expressions occurring under PL/SQL commands in which the value NULL appears, the same rules (logic) apply as those applicable for logical expressions under SQL clauses.

3.1.7. SQL commands in PL/SQL

As part of the PL/SQL block it is allowed to directly use DML, DCL and SELECT commands, but SELECT with modified syntax returning only one row (the syntax of this command will be presented later). DDL commands can be placed in a block only through subprograms of the DBMS_SQL package implementing the so-called **dynamic SQL** or through the use of so-called **internal** (also called native) **dynamic SQL** (native dynamic SQL).

3.1.8. Cursor

Each SQL command placed in the PL/SQL program is processed in a memory area called a workspace or context area. The database server uses this area to store query result data and to store additional information regarding the status of the query, i.e. attributes. The cursor is an identifier for this area.

There are two types of cursors:

– implicit, used automatically when executing INSERT, UPDATE, DELETE and SELECT commands.
– explicit, used to handle queries operating on multiple rows (when these rows require "individual treatment") and in so-called loops with the cursor. Such a cursor is explicitly defined and operated by the programmer.

Explicit cursors will be presented later in the lecture.

The implicit cursor has the following attributes:

Attribute	Description
SQL%ROWCOUNT	Specifies the number of rows processed by the SQL command.
SQL%FOUND	Takes TRUE if the command processed at least one row, FALSE otherwise.
SQL%NOTFOUND	Takes TRUE if no rows have been processed, FALSE otherwise.
SQL%ISOPEN	Takes TRUE if the cursor is open, FALSE otherwise. The implicit cursor is automatically closed, hence the attribute is always FALSE.

Since the Oracle 8i version, there is, for the implicit cursor, an additional attribute SQL%BULK_ROWCOUNT and since Oracle 9i the SQL%BULK_EXCEPTION attribute, both related to the so-called primary mass binding (also named mass SQL - this issue will be presented later). Cursor attributes can be used in PL/SQL commands but not in the SQL commands themselves.

Task. *Modify the relation* `Cats` *so that each cat with an extra mice ration greater than 20 could receive one additional mouse as part of this ration. Display the number of modified rows.*

```
DECLARE
 number_mod NUMBER;
BEGIN
 UPDATE Cats SET mice_extra=mice_extra+1
 WHERE mice_extra>20;
 number_mod:=SQL%ROWCOUNT;
 DBMS_OUTPUT.PUT_LINE('Number modified rows: '||number_mod);
END;

anonymous block completed
Number modified rows: 6
ROLLBACK;
rollback complete.
```

It should be remembered that the PL/SQL block is treated as a transaction unit, therefore even after setting the AUTOCOMMIT parameter to ON, the DML commands from the block will be committed only after it is finished (unless the commits occur in the block). Similarly, if an unhandled error occurs, the block will not make and the DML commands from the block are rollbacked.

3.1.8. SELECT command

A SELECT command placed in a PL/SQL block can only return one row. If no rows will be returned results in the exception NO_DATA_FOUND will appear, if the number of rows returned will be greater than 1 the exception TOO_MANY_ROWS will be returned. The following clauses are allowed in such a command:

SELECT
INTO
FROM
[**WHERE**]
[**GROUP BY**]
[**HAVING**]
[**ORDER BY**]
[**FOR UPDATE** [**OF** {atrybut [, ...]}]] [**NOWAIT** | **WAIT** n]

The INTO clause, which does not exist in pure SQL, is followed by a list of variables to which the values selected in the SELECT clause are assigned. When the FOR UPDATE clause (blocking of rows/attributes to be corrected) is used, the GROUP BY and HAVING clauses and the DISTINCT qualifier are prohibited. Available since Oracle 9i version the WAIT clause specifies the maximum time of the command waits for access to a row/attributes (n is the number of seconds).

Task. *Calculate what percentage of cats receive an additional ration of mice.*

```
DECLARE
  n_cats NUMBER;
  n_with_extra NUMBER;
BEGIN
  SELECT COUNT(*) INTO n_cats
  FROM Cats;
  SELECT COUNT(mice_extra) INTO n_with_extra
  FROM Cats;
  DBMS_OUTPUT.PUT_LINE('*'||LPAD('*',54,'*')||'*');
  DBMS_OUTPUT.PUT_LINE('*'||LPAD(' ',54,' ')||'*');
  DBMS_OUTPUT.PUT_LINE('*  In herd '||
                   ROUND(n_with_extra/n_cats*100,2)||
                  '% cats have an additional mice ration  *');
  DBMS_OUTPUT.PUT_LINE('*'||LPAD(' ',54,' ')||'*');
  DBMS_OUTPUT.PUT_LINE('*'||LPAD('*',54,'*')||'*');
EXCEPTION
  WHEN ZERO_DIVIDE
  THEN DBMS_OUTPUT.PUT_LINE('No cats in the herd!!!');
  WHEN OTHERS
  THEN DBMS_OUTPUT.PUT_LINE(SQLERRM);
END;
```

```
anonymous block completed

********************************************************
*                                                      *
*  In herd 38,89% cats have an additional mice ration  *
*                                                      *
********************************************************
```

3.2. Exceptions

Block execution is always completed when an exception occurs. There are two classes of exceptions:

- predefined - having their numbers (codes), defined by the system constructor,
- user-defined.

The following are some predefined exceptions.

No	Name	Description
ORA-0001	DUP_VAL_ON_INDEX	Violation of the uniqueness constraint.
ORA-0051	TIMEOUT_ON_RESOURCE	The resource allocation has timed out.
ORA-0061	TRANSACTION_BACKED_OUT	Transaction rolled back due to deadlock.
ORA-1001	INVALID_CURSOR	Invalid cursor operation.
ORA-1012	NOT_LOGGED_ON	No database connection.
ORA-1017	LOGIN_DENIED	Unauthorized user or incorrect password.
ORA-1403	NO_DATA_FOUND	No data found.
ORA-1422	TOO_MANY_ROWS	SELECT INTO returns more than one row.
ORA-1476	ZERO_DIVIDE	Division by zero.
ORA-1722	INVALID_NUMBER	SQL error in conversion during numeric value.
ORA-6500	STORAGE_ERROR	Memory error or out of memory.
ORA-6501	PROGRAM_ERROR	Incorrect operation of the PL/SQL program.
ORA-6502	VALUE_ERROR	PL/SQL error during truncation or conversion.
ORA-6511	CURSOR_ALREADY_OPEN	Attempt to open cursor already open.
	OTHERS	Another exception - served last.

3.2.1. Exception handling

Predefined by the system constructor and user-defined exceptions can be handled (the handling is not mandatory) in the EXCEPTION section of the block using the command:

WHEN exception_identifier **THEN** action;

Exception identifiers in the WHEN clause can be combined with logical operators. To handle in the EXCEPTION section an user-defined exception, one must declare it in the DECLARE section of the block according to the syntax:

exception_identifier **EXCEPTION**;

A user defined exception is raised using the command:

RAISE exception_identifier;

Lack of handling in the EXCEPTION section of the exception raised causes the error ORA-06510 (handled or not).

Task. *For cats with the function specified by keyboard, change the ration of mice to the value specified by keyboard. Handle all exceptions.*

```
DECLARE
  maxm Functions.max_mice%TYPE;
  minm Functions.min_mice%TYPE;
  p1   Functions.function%TYPE:='&function';
  p2   Cats.mice_ration%TYPE:=&new_ration;
  too_little_or_too_much EXCEPTION;
BEGIN
  SELECT max_mice,min_mice INTO maxm,minm FROM Functions
  WHERE function=p1;
  IF p2 BETWEEN minm AND maxm
     THEN UPDATE Cats SET mice_ration=p2
          WHERE function=p1;
     ELSE RAISE too_little_or_too_much;
  END IF;
```

```
EXCEPTION
  WHEN NO_DATA_FOUND
       THEN DBMS_OUTPUT.PUT_LINE('Invalid function!!!');
  WHEN too_little_or_too_much
       THEN DBMS_OUTPUT.PUT_LINE('Not for this function!!!');
  WHEN OTHERS
       THEN DBMS_OUTPUT.PUT_LINE(SQLERRM);
END;

anonymous block completed

function - EATER
new_ration - 50

Invalid function!!!
```

To handle exceptions, one can use system error messages returned by the SQLERRM function. If any constraint defined as part of the CREATE TABLE command is violated, the constraint name will be part of the system message returned by the SQLERRM function. For example, assuming that when creating the `Functions` relation, the constraint `CHECK(max_mice<200)` was named `fu_maxm_ch`, in the case of a DML operation, which sets the value of `max_mice` above 199, the OTHERS exception can be handled as follows:

```
...
EXCEPTION
 WHEN OTHERS
      THEN IF UPPER(SQLERRM) LIKE '%FU_MAXM_CH%'
              THEN DBMS_OUTPUT.PUT_LINE('Ration>=200!!! ');
              ELSE DBMS_OUTPUT.PUT_LINE('Error: '||SQLERRM);
           END IF;
END;
```

Another way to handle errors is to use the RAISE_APPLICATION_ERROR function, introduced in Oracle 7 version, which allows one to create own error messages. The syntax for calling this function is:

RAISE_APPLICATION_ERROR(exception_number, message);

The function, unlike the exception supported in the EXCEPTION section, stops block operation, rollbacks all changes and displays the exception number and message specified by the programmer. The exception number is a parameter from -20000 to -20999. The block from the previous task, using the RAISE_APPLICATION_ERROR function, is following:

```
DECLARE
  maxm Functions.max_mice%TYPE;
  minm Functions.min_mice%TYPE;
  p1   Functions.function%TYPE:='&function';
  p2   Cats.mice_ration%TYPE:=&new_ration;
BEGIN
  SELECT max_mice,min_mice INTO maxm,minm FROM Functions
  WHERE function=p1;
  IF p2 BETWEEN minm AND maxm
     THEN UPDATE Cats SET mice_ration=p2
          WHERE function=p1;
     ELSE RAISE_APPLICATION_ERROR(-20001,
                                 'Not for this function!!!');
  END IF;
EXCEPTION
  WHEN NO_DATA_FOUND
       THEN DBMS_OUTPUT.PUT_LINE('Invalid function!!!');
  WHEN OTHERS
       THEN DBMS_OUTPUT.PUT_LINE(SQLERRM);
END;
```

```
anonymous block completed

function - BOSS
new_ration - 250

Not for this function!!!
```

3.2.2. PRAGMA directives

PL/SQL has a number of directives acting as instructions for the compiler (concept shared with ADA). The use of these directives announces in the declaration section of the block, according to the syntax:

PRAGMA directive_name;

Compiler directives will be discussed when they will appear during the lecture.

3.2.3. Spread of exceptions

If an exception occurs within the internal block that the block does not handle, the exception will spread. This means that the exception is passed through subsequent parent blocks until service is encountered. If no such handling exists in any of the blocks, then the exception passes to the external environment.

```
BEGIN
...
  BEGIN
    ...
-- exceptional situation A
    ...
  EXCEPTION
-- no exception handling A
  END;
...
EXCEPTION
-- here handling exception A
END;
```

3.2.4. Notes on exceptions

The following policies may be useful for handling exceptions.

1. A user-defined exception, like a variable, has its scope (block in which it was defined). If an exception has spread beyond its scope, one cannot refer to it by name. The solution to this problem is to define an exception in the package (packages will be discussed in the near future). Then its name will always be available (the package is an object stored in the database).
2. Exceptions related to all commands in block are handled in one exception handling section. This causes, when using several SQL commands of the same type (e.g. several SELECT ... INTO .. commands), difficulties in determining the instruction causing the error. The solution to the problem is to place each such command in a nested block (such block has its exception handling section) or mark each such command with a unique tag value that can be used in EXCEPTION section to handle the exception for particular command.
3. It is a good practice to avoid unhandled exceptions. This is usually achieved by using the OTHERS clause in the exception handling section.

3.3. Instructions

In the PL/SQL block (program), in addition to SQL commands, there may be control structures such as conditional statements and loop instructions. Their syntax is shown below.

3.3.1. Conditional statement IF

The syntax of conditional statement IF is as follows:

```
IF condition THEN {command; [...]}
            [ELSIF condition THEN {command; [...]}] |
            [ELSE {statement; [...]}
END IF;
```

The value TRUE of the condition causes execution commands after THEN, the value FALSE or NULL causes omission these commands and executing commands after ELSE, if this clause appears. Command means PL/SQL or SQL statement. If the next IF statement is to be included in the ELSE clause, it is more convenient to use the ELSIF clause (then there is no END IF at the end of each nested IF statement).

3.3.2. Conditional statement CASE (from Oracle 9i)

There are two forms of CASE statements: simple and searched.

Simple statement:

```
[<<label>>] CASE selector
                  {WHEN condition THEN {command; [...]} [...]}
                   [ELSE {command; […]}]
            END CASE [label];
```

Command means PL/SQL or SQL statement. A selector is an expression of any type whose value is compared with the values of expressions after WHEN clause (their type must match the type of the selector). Commands are executed after the first WHEN clause for which the value of the expression is equal to the value of the selector. If an expression with a value equal to the selector value is missing, the commands in the ELSE clause are executed (if the ELSE clause is omitted, error `ORA-6592 CASE_NOT_FOUND` is reported).

Searched statement:

CASE
  {**WHEN** condition **THEN** {command; [...]} [...]}
  [**ELSE** {command; […]}]
END CASE;

The searched statement executes the commands from the first WHEN clause for which the condition is TRUE. For this form of the CASE conditional statement, in the absence of the ELSE clause, it is required to use at least two WHEN clauses.

Similar to the simple statement, the inability to execute any command (e.g. with ELSE omitted) causes an exception.

3.3.3 CASE expression

In PL/SQL, the CASE keyword, in addition to instruction, can be act as function. Below is an illustrative such a piece of code.

```
ni:='&nickname';
SELECT gender INTO ge
FROM Cats
WHERE nickname=ni;
sex:=CASE ge
       WHEN 'M' THEN 'Male cat'
       WHEN 'F' THEN 'Female cat'
       ELSE 'Gender unknown'
     END;   -- attention!!! END instead of END CASE
...
```

The variable `sex` takes here value `'Male cat'`, `'Female cat'` or `'Gender unknown'`.

3.3.4. Loop instructions

The simplest type of loop is the so-called basic (straight) loop with syntax:

[<<label>>] **LOOP**
{command; [...]}
END LOOP [label];

Command here means PL/SQL or SQL statement. Exit from the loop follows by executing one of the commands EXIT, GOTO or the already known RAISE command.

EXIT [loop_label] [**WHEN** condition];

If EXIT appears as a stand-alone statement, then the WHEN clause with an exit condition is required. This clause is not necessary when EXIT appears as a command after the THEN clause of the IF conditional statement. In the case of nested loops, the loop from which exit follows is determined by its label.

GOTO label;

The above instruction defines an unconditional jump to the label <<label>> in the current block or external block (but not to inside another control instruction).

```
...
LOOP
  counter:=counter+1;
  ...
  IF counter=10 THEN EXIT;
  END IF;
  ...
END LOOP;
...
```

```
...
LOOP
  ...
  EXIT WHEN ration>maxm;
  ...
END LOOP;
...

...
<<ext>>LOOP
          ...
          LOOP
            counter:=counter+1;
            ...
            EXIT ext WHEN ration>maxm;
            EXIT WHEN counter=10;
            ...
          END LOOP;
         END LOOP ext;
...
```

The first condition in the example above cause exit from both loops, the second only from the inner loop.

The second type of loop in PL/SQL is the FOR loop. The FOR loop syntax is as follows:

FOR control_variable **IN** [**REVERSE**] start_value .. end_value
basic_loop;

The control variable of type BINARY_INTEGER or, since Oracle 9i version, PLS_INTEGER is implicitly declared and changes every 1 from the start_value to the end_value (every -1 from the final value to the initial value when REVERSE is used). The start and end values can be expressions of any type convertible to a numeric value.

The third type of loop in PL/SQL is the WHILE loop. The WHILE loop syntax is as follows:

WHILE condition
basic_loop;

The condition is checked at the beginning of each iteration. The loop terminates when the condition is set to FALSE or NULL.

FOR and WHILE loops can also be terminated by using the EXIT, GOTO or RAISE commands. Both of these loops can also be labeled (e.g. for nesting different types of loops).

3.3.5. NULL statement

If in the PL/SQL code needs to indicate that no command is to be performed, although the syntax requires it, then one can use in that place the NULL statement. This instruction serves only as a "filler". For example, if one need to handle an exception without any action, then the code after the EXCEPTION clause is as follows:

```
...
EXCEPTION
  ...
  WHEN OTHERS THEN NULL;
END;
...
```

3.3.6. Blocks with labels

Blocks can be marked with labels (without the outermost block - this can be bypassed by adding artificial external BEGIN and END).

```
BEGIN   -- artificial
  <<ext>>DECLARE
            x NUMBER;
            ...
          BEGIN
            ...
            <<ins>>DECLARE
                      x NUMBER:=10;
                      ...
                    BEGIN
                      ext.x:=ext.x+1;
                      /* the use of an ext label specifies
                         reference to variable x from the
                         external block*/
                      ...
                    END int;
...
          END ext;
END;    -- artificial
```

A block label placed in front of a variable indicates that the variable is from a block described with that label. This allows one to identify variables with the same names, coming from different blocks and available within one block.

3.4. Complex data types

Two types of complex data types can be used in PL/SQL blocks. The first is records, the second is collections. There are three types of collections: index tables, nested tables (since Oracle 8 version) and variable size tables (since Oracle 8 version). Since the Oracle 9i version, it is also possible to build multi-level collections (collections in which collections are their elements). Nested tables and variable size tables are elements of the object-oriented extension of the Oracle database, hence they will be part of the lecture on this topic.

3.4.1. Records

A record variable can be declared in two ways:

- by using the %ROWTYPE pseudo-attribute,
- by using an explicitly defined record type (TYPE ... IS RECORD ...).

The syntax for the record variable declaration using the %ROWTYPE pseudo-attribute is as follows:

record_variable_name object_name**%ROWTYPE**;

The structure of the record defined in this way is consistent with the structure of the object's row (relation, view, explicit cursor). The field names in the record are the same as the object attribute names. Such a record can be filled in the INTO clause of the SELECT command or by assigning values to individual fields. Access to the record field is based on the syntax:

record_variable_name.field_name

In the example below, the SELECT command fills with the Tiger data a record variable with the same structure as row of the relation `Cats`.

```
DECLARE
  cats_r Cats%ROWTYPE;
BEGIN
  SELECT * INTO cats_r
  FROM Cats
  WHERE nickname='TIGER';
...
END;
```

The only operation allowed on a record variable is to assign its value to another record variable of the same type.

The record type is explicitly defined according to the syntax:

TYPE type_name
IS RECORD ({field_name type [**NOT NULL**][:=expression] [, ...]});

The following is an example of definition of the record type and declaration of variable of this type.

```
DECLARE
  TYPE about_cats IS RECORD(nickname VARCHAR2(15),
                            sex VARCHAR2(1) NOT NULL:='M',
                            hunts_since DATE);
  about_cats_r about_cats;
  ...
END;
```

3.4.2. Index tables

Index tables are the first type of collection discussed in this lecture. The other two types (nested tables and variable size tables), due to the use of object elements, will be presented after discussing object extensions of the Oracle system.

The index table type is defined according to the syntax:

TYPE type_name **IS TABLE OF** table_element_type
INDEX BY index_type;

where index_type is an integer type (BINARY_INTEGER or, since Oracle 9i version, PLS_INTEGER) or a string type (VARCHAR2 with length limitation or LONG) and table_element_type is any scalar type, record type (since Oracle 7 version), object type (since Oracle 8 version) or any the collection (since Oracle 9i version).

Access to the field of index table is obtained according to the syntax:

variable_name(index_value)

One can use the %ROWTYPE pseudo-attribute to define the type of record index table. Below is an example of such a definition.

```
TYPE CATS_TABLE IS TABLE OF Cats%ROWTYPE
INDEX BY BINARY_INTEGER;
t_cats CATS_TABLE;
```

In terms of syntax, the index table is treated as an array, but it is actually similar to a database relation (it consists of two columns: KEY of type index_type and VALUE of type table element). This table has an unlimited size (the only limit is the size of the index type) and its elements do not have to be indexed sequentially (index value can be any expression of type index_type).

Task. *For each band, remember in the index table the data of the cat with the longest belonging to the herd (one representative).*

```
DECLARE
  TYPE rec_da IS RECORD (ni Cats.nickname%TYPE,
                         da DATE);
  TYPE tab_da IS TABLE OF rec_da INDEX BY BINARY_INTEGER;
  tab_re tab_da;
  i BINARY_INTEGER; nb NUMBER; n NUMBER;
BEGIN
  SELECT MIN(band_no),MAX(band_no) INTO n,nb
  FROM Cats;
  FOR i IN n..nb
  LOOP
    BEGIN
      SELECT nickname,in_herd_since INTO tab_re(i)
      FROM Cats
      WHERE band_no=i
            AND in_herd_since=(SELECT MIN(in_herd_since)
                               FROM Cats
                               WHERE band_no=i)
            AND ROWNUM=1;
      DBMS_OUTPUT.PUT('Longest in Band '||i||' - ');
      DBMS_OUTPUT.PUT(tab_re(i).ni||' since ');
     DBMS_OUTPUT.PUT_LINE(TO_CHAR(tab_re(i).da,'YYYY-MM-DD'));
    EXCEPTION
      WHEN NO_DATA_FOUND THEN NULL;
    END;
  END LOOP;
   -- continuation of the program using data from the tables
EXCEPTION
  WHEN NO_DATA_FOUND
       THEN DBMS_OUTPUT.PUT_LINE('No cats');
  WHEN OTHERS THEN DBMS_OUTPUT.PUT_LINE(SQLERRM);
END;
```

```
anonymous block completed

Longest in Band 1 - TIGER since 2002-01-01
Longest in Band 2 - FAST since 2006-07-21
Longest in Band 3 - ZOMBIES since 2004-03-16
Longest in Band 4 - REEF since 2006-10-15
```

Since Oracle 7.3 version, one can also use its so-called attributes to support the index tables. Those are:

- EXISTS (n) - returns TRUE or FALSE depending on whether the element with index n exists or not (ORA-1403 error is avoided),
- COUNT - returns the number of rows in the table,
- FIRST, LAST - returns the index value for the first and last row of the table, respectively,
- PRIOR (n), NEXT (n) - for a table row with the index n returns the index value of the previous and next row, respectively,
- DELETE, DELETE (n), DELETE (m, n) - deletes all rows of the table, deletes row with index n, deletes rows with indexes from m to n, respectively.

The attributes are used in accordance with the syntax:

table_name.attribute

3.5. Explicit cursor

The explicit cursor enables to formulate PL/SQL queries directed to many rows and their eventual handling. It is processed in the following steps:

1. Cursor definition.
2. Open the cursor.
3. Fetch the value from the current cursor row(s).
4. Close the cursor.

The explicit cursor is defined in the DECLARE section of the block according to the syntax:

CURSOR cursor_name [({parameter type [, ...]})] **IS**
SELECT_command;

The SELECT command of the cursor does not contain an INTO clause. Cursor parameters act as formal parameters and are (if they occur) used in the SELECT command.

The explicit cursor is opened with the OPEN command according to the syntax:

OPEN cursor_name [({argument [, ...]})];

The arguments of the cursor opening (if any) play the role of actual parameters. They correspond to the formal parameters in the cursor definition. The cursor parameter cannot be the name of relation or view. When cursor opened, the cursor pointer indicates the first row of the relation, which is returned by the SELECT command of cursor.

The value of the current row of the relation returned by the SELECT command of cursor (after opening this is the first row) is fetched by the FETCH command. This command has the syntax:

FETCH cursor_name
INTO {variable [, ...]} | record_variable;

Attribute values of the fetched row are inserted into the list of variables (of types compatible with the types of subsequent expressions of the cursor SELECT clause) or into the record variable (of type cursor_name%ROWTYPE). This fetch takes cursor pointer to the next cursor row. The first FETCH command with no row fetched (all already fetched) will not cause an error, but only target variables or record fields will be NULL. Since Oracle 8i, it is possible to fetch the entire cursor content once to the collection using the BULK COLLECT command. This is an element of the so-called primary mass binding - this topic will be discussed later in the lecture.

The cursor closes with the CLOSE command according to the syntax:

CLOSE cursor_name;

Like implicit cursors, explicit cursors have the attributes %FOUND, %NOTFOUND, %ROWCOUNT, %ISOPEN, in this case preceded by the cursor name. In addition to returning the standard TRUE, FALSE and NULL values, fetching the attribute may result in the exception `ORA-1001 INVALID_CURSOR`.

Attribute	Description
%ISOPEN	Returns TRUE if the cursor is open, FALSE otherwise.
%FOUND	Returns TRUE if the row was successfully fetched, FALSE if no row was fetched. Before the first row fetch NULL is returned. In case the cursor is not yet open or has already been closed, an INVALID_CURSOR exception is returned.
%NOTFOUND	Returns TRUE if no row was returned, FALSE if the row was successfully fetched. Before the first row fetch NULL is returned. In case the cursor is not yet open or has already been closed, an INVALID_CURSOR exception is returned.
%ROWCOUNT	Returns the number of cursor rows fetched so far. In case the cursor is not yet open or has already been closed, an INVALID_CURSOR exception is returned.

The first lack of fetched row will cause the attribute %NOT FOUND to be set to TRUE and the next lack will cause error ORA-1002.

Task. *Find the total number of mice consumed monthly by cats with individual rations greater than the average ration in the herd.*

```
DECLARE
  CURSOR overavr IS
  SELECT NVL(mice_ration,0) mr,
         NVL(mice_extra,0) me
  FROM Cats
  WHERE NVL(mice_ration,0)+NVL(mice_extra,0)>=
        (SELECT AVG(NVL(mice_ration,0)+
                    NVL(mice_extra,0))
         FROM Cats);
  sr NUMBER(4):=0; se NUMBER(4):=0; oa overavr%ROWTYPE;
  are_rows BOOLEAN:=FALSE;
```

```
BEGIN
  OPEN overavr;
  LOOP
    FETCH overavr INTO oa;
    EXIT WHEN overavr%NOTFOUND;
    IF NOT are_rows THEN are_rows:=TRUE; END IF;
    sr:=sr+oa.mr; se:=se+oa.me;
  END LOOP;
  CLOSE overavr;
  IF are_rows
  THEN
    DBMS_OUTPUT.PUT('Monthly consumption: ');
    DBMS_OUTPUT.PUT(TO_CHAR(sr+se,999));
    DBMS_OUTPUT.PUT(' (including additions: ');
    DBMS_OUTPUT.PUT_LINE(TO_CHAR(se,999)||')');
  ELSE
    DBMS_OUTPUT.PUT_LINE('No cats!!!');
  END IF;
END;
```

```
anonymous block completed

Monthly consumption:  650 (including additions:  156)
```

The use of the FOR UPDATE clause in the SELECT command will cause an exclusive lock (the lock is abolished after closing the cursor - the COMMIT command does not work to that moment) for update (UPDATE command) or to delete (DELETE command) rows in the modified relation. If the OF keyword followed by a list of attributes appears after FOR UPDATE, it will narrow down the lock to those attributes only. To the rows modified by the UPDATE or DELETE command which are indicated by the cursor one can reference in the WHERE of these commands clause using clause CURRENT OF (the clause available only in PL/SQL). The syntax of the WHERE clause with CURRENT is as follows:

CURRENT OF cursor_name

Task. *Provide staff for band 5 ('ROCKERS') by assigning to the band the cats with the smallest mice ration in their current band. Delegate a cat with nickname 'LOLA' as a head of the gang and fulfill her postulate that there should be no cats in the new band performing the function 'NICE'.*

```
DECLARE
  CURSOR for_mod IS
  SELECT nickname FROM Cats
  WHERE (((mice_ration,band_no) IN
        (SELECT MIN(NVL(mice_ration,0)),band_no
          FROM Cats
          GROUP BY band_no)) AND function<>'NICE')
        OR
        nickname='LOLA'
  FOR UPDATE OF band_no;
  re for_mod%ROWTYPE; are_rows BOOLEAN:=FALSE;
  no_cat  EXCEPTION;
BEGIN
  OPEN for_mod;
  LOOP
    FETCH for_mod INTO re;
    EXIT WHEN for_mod%NOTFOUND;
    IF NOT are_rows THEN are_rows:=TRUE;
    END IF;
    UPDATE Cats
    SET band_no=5
    WHERE CURRENT OF for_mod;
  END LOOP;
  CLOSE for_mod;
  IF NOT are_rows
     THEN RAISE no_cat;
  END IF;
  UPDATE Bands
  SET name='LOLAS',
      band_chief='LOLA'
  WHERE band_no=5;
-- COMMIT;
EXCEPTION
  WHEN no_cat THEN DBMS_OUTPUT.PUT_LINE('No cat');
  WHEN OTHERS THEN DBMS_OUTPUT.PUT_LINE(SQLERRM);
 END;

anonymous block completed

SELECT nickname,B.name
FROM Cats JOIN Bands B USING(band_no)
WHERE band_no=5;

NICKNAME        NAME
--------------- --------------------
LOLA            LOLAS
EAR             LOLAS
SMALL           LOLAS

ROLLBACK;

rollback complete.
```

As mentioned earlier, the cursor may have parameters. Below is a fragment of code with an example of such a cursor.

```
...
CURSOR choice (par1 NUMBER,par2 VARCHAR2) IS
SELECT name,mice_ration,in_herd_since
FROM Cats
WHERE band_no=par1 AND function=par2;
...
bno:=1;fu:='NICE';
...
OPEN choice(bno,fu);
...
```

The above defined and open cursor returns the data of cats with function `'NICE'`, belonging to the band No. 1.

3.5.1. FOR loop with cursor

Processing of cursor can be simplified by using so-called FOR loop with cursor. This loop has the following syntax:

FOR record_variable **IN** cursor_name[({parameter type [, ...]})]
basic_loop;

The number of loops in a loop is equal to the number of rows returned by the cursor. The record variable is implicitly declared as cursor_name%ROWTYPE type. The cursor opens implicitly when the loop is initialized. In each step of loop takes place implicit fetch row of cursor to a record variable which can be processed in the body of loop (basic loop). At the end implicit the cursor is closed. (also if the exit from the loop occurs via the EXIT, GOTO or RAISE command). Below is a fragment of code with an example of such a loop.

```
...
FOR no IN choice(bno,fu)
LOOP
  s:=s+no.przydzial_myszy   -- operation on the cursor field!
END LOOP;
...
```

The above loop supports the cursor named choice. It determines the sum of rations of mice for cats belonging to the band number `bno` and acting as `fu`. It should be noted here that in the case of such handling of field of record variable (in the example above it is variable `no`) they have names consistent with the names of the cursor fields specified within his definition (SELECT clause of the cursor command).

Instead of in the DECLARE section of the block, the cursor can also be defined directly in the FOR loop with the SELECT command of cursor. Such a cursor is operated according to the following syntax:

FOR record_variable **IN** (SELECT_command)
basic_loop;

The body (content) of the cursor is defined by the SELECT command after the keyword IN. It is also an explicit cursor, but it has no name (it has not been defined in the DECLARE section) so there is no access to its attributes and it cannot be parameterized.

Task. *Use the FOR loop with cursor to find the cats with the longest membership in their own bands.*

```
DECLARE
  are_rows BOOLEAN:=FALSE;
  no_cats EXCEPTION;
BEGIN
  FOR re IN (SELECT nickname,in_herd_since,band_no
             FROM Cats
             WHERE (in_herd_since,band_no) IN
                         (SELECT MIN(in_herd_since),band_no
                          FROM Cats
                          GROUP BY band_no))
  LOOP
    are_rows:=TRUE;
    DBMS_OUTPUT.PUT('Band '||re.band_no||' - longest ');
    DBMS_OUTPUT.PUT(re.nickname||' since ');
  DBMS_OUTPUT.PUT_LINE(TO_CHAR(re.in_herd_since,'YYYY-MM-
DD'));
  END LOOP;
  IF NOT are_rows
     THEN RAISE no_cats;
  END IF;
EXCEPTION
```

```
  WHEN no_cats THEN DBMS_OUTPUT.PUT_LINE('No cats');
  WHEN OTHERS THEN DBMS_OUTPUT.PUT_LINE(SQLERRM);
END;

anonymous block completed

Band 1 - longest TIGER since 2002-01-01
Band 2 - longest FAST since 2006-07-21
Band 4 - longest REEF since 2006-10-15
Band 3 - longest ZOMBIES since 2004-03-16
```

By using the explicit cursor within the PL/SQL block, it was possible to handle a SELECT query returning more than one row.

3.5.2. Cursor variables

The cursor variable is a reference type, i.e. a pointer to the memory area where the query result and its attributes are stored. The cursor used until now was the equivalent of the PL/SQL constant (static cursor). The syntax for defining the type of a reference variable for a cursor is as follows:

TYPE cursor_type_name **IS REF CURSOR**;

After defining the cursor variable type, one only need to declare variable of this type.

Like the static cursor, the cursor variable must be opened (*de facto* must be set its value). This is done according to the syntax:

OPEN cursor_variable **FOR** SELECT_command;

The SELECT command defines the cursor pointed to by the cursor variable. Fetching rows from the cursor variable and closing the cursor variable is done in a similar way to these operations for a static cursor.

The following is an example of using one cursor variable for its various values.

```
DECLARE
  TYPE cursor_type IS REF CURSOR;
  cursor_v cursor_type;
  ni Cats.nickname%TYPE;
  mr Cats.mice_ration%TYPE;
  me Cats.mice_extra%TYPE;
  en Enemies.enemy_name%TYPE;
  hd Enemies.hostility_degree%TYPE;
  relation_code VARCHAR2(2):='&1';
BEGIN
  IF relation_code ='CA'
     THEN OPEN cursor_v FOR
          SELECT nickname,mice_ration,mice_extra
          FROM Cats
          WHERE mice_ration>50;
  ELSIF relation_code ='WR'
        THEN OPEN cursor_v FOR
             SELECT enemy_name,hostility_degree
             FROM Enemies
             WHERE hostility_degree>5;
  ELSE
   RAISE_APPLICATION_ERROR(-20103,
                            'This relation is not supported');
  END IF;
  LOOP
    IF relation_code ='CA' THEN
       FETCH cursor_v INTO ni,mr,me;
       EXIT WHEN cursor_v%NOTFOUND;
       ...
    ELSE
       FETCH cursor_v INTO en,hd;
       EXIT WHEN cursor_v%NOTFOUND;
       ...
    END IF;
  END LOOP;
  CLOSE cursor_v;
END;
```

There are some restrictions on the use of the cursor variable (for Oracle 7.3 and above). They are presented below.

- one cannot use the %ROWTYPE attribute within the cursor variable,
- PL/SQL collections cannot store cursor variables,
- only since Oracle 8i version the SELECT command defining the cursor can contain the FOR UPDATE clause,
- cursor variables cannot be declared in the package (one can only define the cursor type in it).