The PL/SQL language

PL/SQL is a procedural language used in Oracle databases. This language is an extension of SQL, providing several instructions, such as loops, conditional statements, blocks, functions, procedures, ...

With PL/SQL we can, for instance:

- use loops and conditional statements;
- create functions, procedures, triggers and packages;
- improve the performance of database;
- create complex programs, to solve problems which are impossible or hard to solve using only SQL instructions;
- handle errors and exceptions.

Using PL/SQL we can write the following types of programs:

1. Anonymous blocks;
2. Stored procedures;
3. Triggers;
4. Functions.

Basic data types in PL/SQL. Numeric data

NUMBER[(*precision,scale*)]: to store numeric data: Fixed-point or floating-point numbers.

Precision is the total number of digits, *scale* is the number of digits to the right of the decimal point.
For precision, the maximum value is 38 (by default, 38).
Scale can be from -84 to 127 (default 0).

To define an integer, use: NUMBER(p) or INT (a subtype of NUMBER).

Fixed-point numbers are defined by giving both precision and scale, for example, NUMBER(6,3).

Floating-point numbers are defined as NUMBER, with no parameters.

Basic data types in PL/SQL. Numeric data

PLS_INTEGER: stores signed integers in the range -2,147,483,648 through 2,147,483,647, represented in 32 bits.
For efficiency, use PLS_INTEGER values for all calculations that fall within its range. It is faster than NUMBER or INT.

BINARY_FLOAT and BINARY_DOUBLE data types represent single-precision and double-precision floating-point numbers, respectively.

A BINARY_FLOAT literal ends with `f` (for example, `2.07f`).

A BINARY_DOUBLE literal ends with `d` (for example, `3.000094d`).

Basic data types in PL/SQL

CHAR(n) - Fixed-length character string with maximum size of 32,767 bytes, n must be from 1 to 32767. By default, n=1.

VARCHAR2(n) - Variable-length character string with maximum size of 32,767 bytes, n must be from 1 to 32767. The parameter n must be given.

BOOLEAN - data type that stores logical values, which can be used in logical operations. The logical values are the Boolean values TRUE and FALSE and the value NULL. SQL has no data type equivalent to BOOLEAN.

DATE - to store date and time.

TIMESTAMP - stores the year, month, day, hour, minute, and second.

Anonymous blocks in PL/SQL

In PL/SQL we can write anonymous blocks, without giving them any names. Such blocks are not stored in the database. They can be used for single execution of code. Simplified syntax:

```
DECLARE
        variables declarations
BEGIN
        executable code
END;
```

Anonymous blocks in PL/SQL

In PL/SQL we can write anonymous blocks, without giving them any names. Such blocks are not stored in the database. They can be used for single execution of code. Simplified syntax:

```
DECLARE
        variables declarations
BEGIN
        executable code
END;
```

In the next example each employee in table `employee` is given a rise of 100$.

```
DECLARE rise NUMBER(4);
BEGIN
    rise:=100;
    UPDATE employee SET salary=salary+rise;
END;
```

Constants and variables declarations

Constants and variables are declared in the DECLARE section of the anonymous block. To declare a variable write:

```
DECLARE variable data_type;
```

To declare a constant:

```
DECLARE const1 CONSTANT data_type := value;
```

For example:

```
DECLARE number1 NUMBER(4);
        integer1 INT :=90;
        integer2 PLS_INTEGER NOT NULL :=0;
        double1 BINARY_DOUBLE;
        text1 VARCHAR2(30);
        date1 DATE DEFAULT SYSDATE;
        date2 DATE := DATE '2017.04.25';
        boolean1 BOOLEAN;
        const1 CONSTANT CHAR(5) :='something';
```

In order to initialize a variable, we can use the statement: `variable1 data_type :=expression` or the DEFAULT clause: `variable1 data_type DEFAULT value`. Otherwise, the variable is, by default, initialized to NULL.

Using SQL statements in PL/SQL programs

In PL/SQL we can use the `INSERT, UPDATE, DELETE` statements, similarly, as in SQL. The `SELECT` statement is used together with the clause `INTO`.

Using SQL statements in PL/SQL programs

In PL/SQL we can use the `INSERT, UPDATE, DELETE` statements, similarly, as in SQL. The `SELECT` statement is used together with the clause `INTO`.

Example: using UPDATE in a block:

```
DECLARE
amount NUMBER(5):=200;
dept INT;
BEGIN
dept:=90;
UPDATE employees SET salary=salary+amount
WHERE department_id=dept;
END;
```

Using SQL statements in PL/SQL programs

In PL/SQL we can use the `INSERT, UPDATE, DELETE` statements, similarly, as in SQL. The `SELECT` statement is used together with the clause `INTO`.

Example: using INSERT in a block:

```
DECLARE
id char(2);
name varchar2(40);
region number:=1;
BEGIN
name:='Poland';
id:='PL';
INSERT INTO
countries(country_id,country_name,region_id)
VALUES (id,name,region);
END;
```

The DELETE statement is used analogously.

In case of the SELECT statement, the clause INTO is required. The SELECT statement must return **exactly one row!**
In the clause INTO we put the list of variables, to which the values returned by SELECT are assigned to.
The number and types of the variables and the values returned by the SELECT statement must match.

In case of the SELECT statement, the clause INTO is required. The SELECT statement must return **exactly one row!**
In the clause INTO we put the list of variables, to which the values returned by SELECT are assigned to.
The number and types of the variables and the values returned by the SELECT statement must match.

```
declare
v_sum numeric(10);
v_average numeric(10,2);
begin
select sum(salary), avg(salary) into v_sum,
v_average from employees;
DBMS_OUTPUT.put_line('total sum:  '||v_sum||'
average:  '||v_average);
end;
```

In case of the SELECT statement, the clause INTO is required. The SELECT statement must return **exactly one row!**
In the clause INTO we put the list of variables, to which the values returned by SELECT are assigned to.
The number and types of the variables and the values returned by the SELECT statement must match.

```
declare
v_sum numeric(10);
v_average numeric(10,2);
begin
select sum(salary), avg(salary) into v_sum,
v_average from employees;
DBMS_OUTPUT.put_line('total sum:  '||v_sum||'
average:  '||v_average);
end;
```

To test the PL/SQL programs, it is convenient to display the results on the console. It can be done using the function `put_line` from the package `DBMS_OUTPUT`, as follows:

```
DBMS_OUTPUT.Put_line(text);
```

In case of the SELECT statement, the clause INTO is required. The SELECT statement must return **exactly one row!**
In the clause INTO we put the list of variables, to which the values returned by SELECT are assigned to.
The number and types of the variables and the values returned by the SELECT statement must match.

```
declare
v_sum numeric(10);
v_average numeric(10,2);
begin
select sum(salary), avg(salary) into v_sum,
v_average from employees;
DBMS_OUTPUT.put_line('total sum:  '||v_sum||'
average:  '||v_average);
end;
```

The names of variables should be different from the names of the columns in the database tables.

Some conventions can be helpful, for instance, each variable name can start with `v_`, constants with `c_`, and so on.

PL/SQL Control Structures

The assignment operator :=

```
variable :=expression;
```

for example:

```
amount:=100;
actual_date:=sysdate;
```

Control Structures: conditional statement IF

The basic syntax of the conditional statement:

`IF...THEN...END IF.`

We can also use the `ELSE` and/or `ELSEIF` clauses:

```
IF condition THEN instructions;
ELSIF condition THEN instructions;
ELSE instructions END IF;
```

If the IF-condition is satisfied, then the first THEN-block of instructions is executed. For otherwise, and if the ELSIF-condition is fulfilled, then the second THEN-block of instructions is executed. For otherwise, the instructions from the ELSE-block are executed.
Both clauses ELSE and ELSEIF are not mandatory.

Example: solve the equation $ax + b = 0$.

```
DECLARE a number; b number;
BEGIN
    a:=1;
    b:=10;
    IF(a=0) THEN
        IF(b<>0) THEN dbms_output.put_line('no
solution');
        ELSE dbms_output.put_line('infinitely many
solutions');
        END IF;
    ELSE dbms_output.put_line('the root x='||-b/a);
    END IF;
END;
```

Control Structures: conditional statement CASE

CASE — two forms: simple case and searched case.
Searched CASE.
The **search conditions** are evaluated sequentially. If a search condition yields TRUE, its WHEN clause is executed. If any WHEN clause is executed, control passes to the next statement, so subsequent search conditions are not evaluated.

If none of the search conditions yields TRUE, the ELSE clause is executed. The ELSE clause is optional. But if none of the search conditions yields TRUE and there is no ELSE, CASE will return an error: exception `CASE_NOT_FOUND`.

```
CASE
WHEN condition1 THEN instructions1;
WHEN condition2 THEN instructions2;
...
ELSE instructions;
END CASE;
```

Using searched CASE:

```
DECLARE i NUMBER(1) := 3;
BEGIN
CASE
WHEN i=1 THEN DBMS_OUTPUT.PUT_LINE('i equals
1');
WHEN i>2 THEN DBMS_OUTPUT.PUT_LINE('i is
greater than 2');
ELSE DBMS_OUTPUT.PUT_LINE('i equals ' || i);
END CASE;
END;
```

Control Structures: conditional statement CASE

Simple CASE statement:
Such a CASE statement uses a **selector**: an expression whose value is used to select one of several alternatives. The selector expression is evaluated only once.
WHEN clauses are checked sequentially. The value of the selector determines which clause is executed. If the value of the selector equals the value of a WHEN-clause expression, that WHEN clause is executed.
If the CASE statement does not match any of the WHEN clauses, then the ELSE clause is executed, if present, otherwise, an exception `CASE_NOT_FOUND` is raised.

```
CASE selector
WHEN value1 THEN instructions1;
WHEN value2 THEN instructions2;
...
ELSE instructions;
END CASE;
```

Loop iterations: LOOP

Basic LOOP is defined within clauses LOOP and END LOOP.
Syntax:

```
LOOP
      sequence_of_statements
END LOOP;
```

To prevent an infinite loop, we must use either an EXIT WHEN or EXIT statement (with an IF statement).

For example:

```
DECLARE
   i INT :=1;
BEGIN
LOOP
   dbms_output.put_line('i= '||TO_CHAR(i));
   EXIT WHEN i=10;
   i:=i+1;
END LOOP;
END;
```

Loops: WHILE

The WHILE-LOOP statement executes the statements in the loop body as long as a condition is true:

```
WHILE condition LOOP
      sequence_of_statements
 END LOOP
```

The condition is evaluated before each iteration of the loop. If it is TRUE, the sequence of statements is executed, then control resumes at the top of the loop. If it is FALSE or NULL, the loop is skipped and control passes to the next statement.

The number of iterations depends on the condition and is unknown until the loop completes. The condition is tested at the top of the loop, so the sequence might execute zero times.

One of the statements inside the loop body must change the condition to FALSE or NULL, to avoid an infinite loop. To exit a WHILE loop we can also use EXIT WHEN.

Loops: FOR

The FOR loop can be used, when the number of iterations is known in advance. The FOR loop iterates over a specified range of integers `(lower_bound .. upper_bound)`. If lower_bound equals upper_bound, the loop body is executed once.

```
FOR counter IN lower_bound ..upper_bound LOOP
    sequence_of_statements
END LOOP
```

Inside a FOR loop, the counter can be read but cannot be changed.

By default, iteration proceeds upward from lower to higher bound. After each iteration, the loop counter is incremented by 1.
Use the keyword `REVERSE` to iterate downward from higher to lower bound. After each iteration, the loop counter is decremented. But, we must put first the smaller value and then the larger one.

```
FOR counter IN REVERSE 1..n LOOP
    sequence_of_statements
END LOOP.
```

Loops: FOR. The scope of the loop counter variable

```
FOR counter IN lower_bound ..upper_bound LOOP
    sequence_of_statements
END LOOP
```

- The loop counter variable is defined only within the loop (its scope is restricted to the loop).
- That variable cannot be referenced outside the loop.
- After the loop exits, the loop counter is undefined.
- The loop counter needs no declaration — it is implicitly declared as a local variable of type INTEGER.

It is better not to give a loop variable the same name as an existing variable, because the local declaration hides the global declaration, i.e., inside the loop only the local variable is visible (the loop counter) not the global variable with the same name.

Functions in PL/SQL

Function — a block of PL/SQL that has a name and is stored in the database. The function must return a value.
In most cases, functions can be used in SQL statement similarly like Oracle built-in functions.
Creating a function — basic syntax.

```
CREATE FUNCTION name
RETURN data_type
IS
        variables declarations
BEGIN
        executable statements
RETURN result;
END;
```

We can call a function from the SQL statement:

```
SELECT function_name FROM dual;
```

or from another PL/SQL block (or function, procedure, trigger).

To drop a function use

```
DROP FUNCTION name;
```

Example of a simple function

```
CREATE OR REPLACE FUNCTION summing
RETURN INT
IS
number1 INT:=90;
number2 INT:=100;
total INT;
BEGIN
total:=number1+number2;
RETURN total;
END;
```

Using parameters in functions

PL/SQL functions (and procedures) can have parameters of three types:

IN - input parameter, read-only, used to pass data to a function or procedure;

OUT - output parameter, can be used to return data from a procedure (or, rarely, function); with NULL value until initialize;

IN OUT - input/output parameter; can be used to pass data to a program and to return data from; can be used in cases when input data can be changed during the execution of the program.

Remark. By default, if we do not put the type of a parameter, it is treated as an IN parameter.

Input parameters in functions

In functions typically we use only input parameters (and RETURN clause to return data).
Syntax of the function definition with input parameters:

```
CREATE FUNCTION name
(parameter1 IN data_type,
parameter2 IN data_type,...)
RETURN data_type
IS
        variables declarations
BEGIN
        executable statements
RETURN result;
END;
```

Remark. When using data types `CHAR, NUMBER, VARCHAR2` for parameters, we do not put the range; however, we can use subtypes, like `INTEGER`, or types `PLS_INTEGER, BINARY_FLOAT`, ...

Example — function calculating an arithmetical mean of two numbers

```
CREATE OR REPLACE FUNCTION a_mean
(number1 IN NUMBER,
number2 IN NUMBER)
RETURN NUMBER
IS
mean NUMBER;
BEGIN
mean:=(number1+number2)/2;
RETURN mean;
END;
```

Function call:

```
SELECT a_mean(1,2) FROM dual;
```

Example — the function formatting postal codes

The function takes the chain of five integers, like 99999, representing a postal code and returns it in the form 99-999.

```
CREATE OR REPLACE FUNCTION postal_code
(code IN char)
RETURN char
IS
BEGIN
RETURN substr(code,1,2)||'-'||substr(code,3);
END;
```

Function call:

```
SELECT postal_code('56009') FROM dual;
```

Attributes %TYPE and %ROWTYPE

We can use the attribute %TYPE to declare a variable (or parameter) whose data type will be the same as the data type of the given column. It is useful in situation, when the variable is used to store data from the particular column.
For instance, to define a variable `v_last_name` with the same data type as column `last_name` in table `employee`, use the dot notation and attribute `%TYPE` as follows:

```
v_last_name employee.last_name%TYPE;
```

The advantages of using the attribute `%TYPE`:

- It is not necessary to know the data type of the column `last_name` to define a variable with the same data type;
- If the data type of the column `last_name` will be changed in the database, the data type of the variable `v_last_name` will be changed automatically.

Attributes %TYPE and %ROWTYPE

The attribute `%ROWTYPE` provides a record type that represents a row in a database table.
Such a record can store a row of data selected from the table or fetched from a cursor.
The fields in the record correspond to the columns of the given table (have the same names and data types).

For example, declare the variable `v_dept_rec` as a record for storing a row of data from table `department`. Fields of the record have the same names and data types as columns in table `department`.

```
v_dept_rec department%ROWTYPE;
```

To refer to a field in a record, we use dot notation:

```
v_deptid := v_dept_rec.dept_id;
```

Cursors

For each SQL statement (such as `INSERT, UPDATE, DELETE` and `SELECT`) used in a PL/SQL program, Oracle create a **cursor** (an unnamed work area) to store data necessary to execute the statement, among others, the status of the statement, and the set of selected rows (for SELECT statements).

When we execute a multi-row query, we can use a cursor to name this work area, access the information, and process the rows individually.

With the cursor, we can:

- name the work area,
- access it,
- fetch the rows from it,
- control the process of retrieving data.

Cursors

There are two types of cursors:
• **explicit** - declared in the PL/SQL program, used to read the set of records of data from the database,
• **implicit** - opened implicitly by Oracle for each SQL statement `UPDATE, INSERT, DELETE` and `SELECT INTO` that is used in the PL/SQL program.

To process an explicit cursor, we must first declare it. We use three commands to control a cursor:
• open the cursor (with `OPEN cursor_name`),
• fetch rows from the cursor (using `FETCH cursor_name INTO variable`),
• close the cursor (with `CLOSE cursor_name`).

Example

```
DECLARE
CURSOR my_cursor IS
SELECT * FROM employees WHERE
department_id='100';
vperson my_cursor%ROWTYPE;
BEGIN
OPEN my_cursor;
FETCH my_cursor INTO vperson;
dbms_output.put_line('Personal data:
'||vperson.last_name||'
'||vperson.first_name);
CLOSE my_cursor;
END;
```

Example

```
DECLARE
CURSOR my_cursor IS
SELECT * FROM employees WHERE
department_id='100';
vperson my_cursor%ROWTYPE;
BEGIN
OPEN my_cursor;
FETCH my_cursor INTO vperson;
dbms_output.put_line('Personal data:
'||vperson.last_name||'
'||vperson.first_name);
CLOSE my_cursor;
END;
```

Remark. The variable `vperson` is a record (declared using the attribute %ROWTYPE) with the fields that correspond to the columns of the cursor `my_cursor`.

Fetching rows from the cursor in a loop: cursor loop FOR

The cursor loop FOR can be used to fetch records from the cursor.
The sequence of statements inside the loop is executed once for each row that satisfies the query.
When using such a loop, we do not open and close the cursor explicitly.
When we leave the loop, the cursor is closed automatically (even if we use an EXIT statement to leave the loop before all rows are fetched, or an exception is raised inside the loop).

```
FOR iterator IN cursor_name LOOP
sequence of statements
END LOOP;
```

Fetching rows from the cursor in a loop: cursor loop FOR

```
DECLARE
CURSOR my_cursor IS
SELECT * FROM employees WHERE
department_id='100';
BEGIN
dbms_output.put_line('Employees from the
department 100:');
FOR person IN my_cursor LOOP
dbms_output.put_line(person.last_name||'
'||person.first_name);
END LOOP;
END;
```

Fetching rows from the cursor in a loop: cursor loop FOR

```
DECLARE
CURSOR my_cursor IS
SELECT * FROM employees WHERE
department_id='100';
BEGIN
dbms_output.put_line('Employees from the
department 100:');
FOR person IN my_cursor LOOP
dbms_output.put_line(person.last_name||'
'||person.first_name);
END LOOP;
END;
```

Remark. The iterator variable `person` for the FOR loop does not need to be declared in advance. It is a %ROWTYPE record whose field names match the column names from the query, and that exists only during the loop.

It is used to store the rows fetched from the cursor; we refer to the fields of this record variable inside the loop (with the dot notation).

Cursor attributes

Cursor attributes return useful information about the execution of a data manipulation statement.

`%ISOPEN` — returns TRUE, if the cursor is already opened, and FALSE otherwise.
`%ROWCOUNT` — returns the number of rows fetched so far; before the first fetch returns 0;
`%FOUND` — checks, if there are rows to fetch in the cursor; before the first fetch from an open cursor, it returns NULL; afterwards, it returns TRUE if the last fetch returned a row, or FALSE if the last fetch did not return a row.
`%NOTFOUND` — before the first fetch from an open cursor, it returns NULL; then, it returns FALSE if the last fetch returned a row, or TRUE otherwise.

If a cursor is not open, referencing it with `%FOUND`, `%NOTFOUND` or `%ROWCOUNT` raises the predefined exception `INVALID_CURSOR`.

We can use the cursor attributes in procedural statements, but not in SQL statements.

Example — using cursor attributes

```
DECLARE
CURSOR cursor1 IS SELECT * FROM countries;
rec_country countries%ROWTYPE;
vcount INT;
BEGIN
OPEN cursor1;
IF cursor1%ISOPEN THEN
Dbms_Output.put_line('cursor is opened');
vcount:=cursor1%ROWCOUNT;
Dbms_Output.put_line('The number of rows
fetched so far:  '||vcount);
FETCH cursor1 INTO rec_country;
vcount:=cursor1%ROWCOUNT;
CLOSE cursor1;
END IF;
Dbms_Output.put_line('The number of rows
fetched so far:  '||vcount);
END;
```

Cursors with parameters

The next code displays the data of each country from region 1.

```
DECLARE
CURSOR cur_country1 IS SELECT * FROM countries
WHERE region_id=1;
BEGIN
FOR country IN cur_country1 LOOP
Dbms_Output.put_line( country.country_id||'
'||country.country_name);
END LOOP;
END;
```

Cursors with parameters

Suppose that we want to display the data of countries from the given region, and we want to choose the region id.
In such a case we can use a parameter in our cursor. Such a parameter is passed to the cursor and is used in the WHERE clause.

```
DECLARE
CURSOR cur_country2(id INT) IS SELECT * FROM
countries WHERE region_id =id;
BEGIN
FOR country IN cur_country2(2) LOOP
Dbms_Output.put_line( country.country_id||'
'||country.country_name);
END LOOP;
END;
```

The cursor `cur_country2` is a cursor with parameter `id`.

Cursors with parameters

We can define more than one parameter for a cursor, for example:

```
CURSOR c1 (dept CHAR, sal NUMBER) IS
SELECT * FROM employee
WHERE dept_no=dept AND salary >=sal;
```

When the cursor has a parameter (parameters), then using the cursor (opening it or using with cursor loop FOR) we must pass the values for all parameters. For example, to open the above defined cursor, use:

```
OPEN c1('100',2500);
```

The cursor parameters may have default values:

```
CURSOR c1 (dept CHAR, sal NUMBER DEFAULT 2000)
IS
SELECT * FROM employee
WHERE dept_no=dept AND salary >=sal;
```

Cursors with parameters

When using cursors with parameters, we must take into account the following:

- Cursor becomes more reusable with cursor parameters.
- The mode of the parameters can only be IN, i.e., we can only pass values to the cursor; and cannot pass values out of the cursor through parameters.
- Default values can be assigned to cursor parameters.
- The scope of the cursor parameters is local to the cursor.
- Only the data type of the parameter is defined, not its length.

Stored procedures

The next type of a PL/SQL programs are **stored procedures**.
The stored procedure can have parameters of type:
IN, OUT or IN OUT.
The stored procedure can be used to make some operations on the data in a database or to choose data from the database as well.

```
CREATE PROCEDURE name(parameters list)
IS
        variables declarations
BEGIN
        statements
END;
```

To call a stored procedure from SQL we use `EXECUTE`.

```
CREATE PROCEDURE display_country(p_id IN INT)
is
CURSOR country2(id INT) IS SELECT * FROM countries
WHERE region_id =id;
BEGIN
FOR item IN country2(p_id) loop
Dbms_Output.put_line(item.country_name);
END LOOP;
END;
```

```
CREATE PROCEDURE display_country(p_id IN INT)
is
CURSOR country2(id INT) IS SELECT * FROM countries
WHERE region_id =id;
BEGIN
FOR item IN country2(p_id) loop
Dbms_Output.put_line(item.country_name);
END LOOP;
END;
```

To execute the procedure from SQL, for region with id=2, use:

```
EXECUTE display_country(2);
```

To call the procedure from another PL/SQL program, use:

```
DECLARE v_id INT;
BEGIN
v_id:=1;
display_country(v_id);
END;
```

Functions vs. procedures

Function

must return a value (has RETURN clause);

in most cases has only IN parameters;

typically, can be call from an SQL statement, for example

```
SELECT amean(1,3) FROM
dual;
```

can be called from another PL/SQL program, but the result of the function must be assign to a variable or parameter (of type OUT or IN OUT):

```
DECLARE i number;
BEGIN
i:=amean(1,4);
END;
```

Procedure

may return values;

can have parameters IN, OUT, IN OUT;

call from SQL with `EXECUTE`:

```
EXECUTE
display_country(3);
```

can be called from other PL/SQL programs:

```
DECLARE id INT;
BEGIN
id:=1;
display_country(id);
END;
```

Using parameters in procedures and functions

PL/SQL procedures (and functions) can have parameters of three types:

IN - input parameter, read-only, used to pass data to a function or procedure; if there is no default defined for this parameter, then calling a procedure or a function we must provide an actual value for it;

OUT - output parameter, can be used to return data from a procedure (or, rarely, function); with NULL value until initialize; calling a procedure or function, we cannot assign a constant or literal to that parameter (we must use a variable);

IN OUT - input/output parameter; can be used to pass data to a program and to return data from; can be used in cases when input data are to be changed during the execution of the program; we cannot define a DEFAULT value for such parameter; calling a procedure or function, we cannot assign a constant or literal to it.

Using parameters in procedures and functions

When we call a procedure (function) that has several input parameters, we have to provide values for each of them, but we may not give a value for these input parameters, which have the DEFAULT. There are three notations that can be used to call a procedure (function): positional, named and mixed.

Using parameters in procedures and functions

When we call a procedure (function) that has several input parameters, we have to provide values for each of them, but we may not give a value for these input parameters, which have the DEFAULT. There are three notations that can be used to call a procedure (function): positional, named and mixed.

```
CREATE OR REPLACE FUNCTION add
(a INT :=0, b INT :=0, c INT :0)
RETURN INT
AS
BEGIN
RETURN a+b/2+c/4;
END;
```

Using parameters in procedures and functions

When we call a procedure (function) that has several input parameters, we have to provide values for each of them, but we may not give a value for these input parameters, which have the DEFAULT. There are three notations that can be used to call a procedure (function): positional, named and mixed.

```
CREATE OR REPLACE FUNCTION add
(a INT :=0, b INT :=0, c INT :0)
RETURN INT
AS
BEGIN
RETURN a+b/2+c/4;
END;
```

Call this function from `SELECT` using positional, named and mixed notation:

```
select add(1,2,8) from dual; //a=1,b=2,c=8
select add(1,4) from dual; //a=1,b=4, c default
select add(c=>8,a=>2,b=>6) from dual;
select add(1,c=>4) from dual; //a=1, c=4, b default
```

Sequences

Sequence in Oracle database - a tool that can be used to generate, sequentially, integer numbers.

Creating a sequence:

```
CREATE SEQUENCE name
INCREMENT BY step
START WITH number
```

Description of the options:
`INCREMENT BY step` — the number the sequence will be incremented by (by default 1),
`START WITH liczba` — the start number (by default 1).

We can manipulate a sequence using the following methods:
`CURRVAL` — get the current value,
`NEXTVAL` — get the next value, i.e., after the incrementation by step.

Sequences — using to generate values for primary keys

In table employee we have the primary key on the column emp_no. We create a sequence which we can later use to generate values for this primary key:

```
CREATE SEQUENCE emp_no_seq
INCREMENT BY 1
START WITH 200
```

We can use the sequence in INSERT statement:

```
INSERT INTO employee
(emp_no, first_name, last_name, salary,
dept_no, hire_date, job_code, job_grade,
job_country)
VALUES(emp_no_seq.NEXTVAL, 'Anna',
'Kowalska',5000,
'000',sysdate,'Admin',4,'USA');
```

Sequences — methods

Every call of the method `NEXTVAL` results in incrementing the sequence by the step. If the generated value is not used (because, for example, the transaction was roll-backed or an error occurs when evaluating the INSERT statement), then the generated value is *lost*. The next call of `NEXTVAL` will return the next value.

The same sequence can be used to generate values for primary keys in several tables.

The method `CURRVAL` does not change the value of the sequence, only returns the current one.

We cannot use `CURRVAL` for the given sequence until we call `NEXTVAL` for that sequence at least once.

Exceptions

In PL/SQL, an error condition is called an **exception**.
Exception — an run-time error, occurs during the execution of the block (in PL/SQL program).

When an error occurs, an exception is **raised**. The normal execution of the block stops. The control transfers to the exception-handling part of the PL/SQL block, if it is present.

So, handling an exception, we can execute some additional statements before the block is finished.

Types of exceptions:

- internally defined (by the run-time system), most common have names (such as `NO_DATA_FOUND, ZERO_DIVIDE`), the other Oracle server exceptions can be given names;
- user-defined — defined by the user in the declarative part of the block, must be raised explicitly by `RAISE` statement, must be given names.

Types of exceptions

- **Internally define Oracle exceptions.**
  An internal exception is raised automatically (by the run-time system) if the PL/SQL program violates a database rule or exceeds a system-dependent limit.
  Each such exception has its code and message describing the error (a message also contains the code of the given error).
  - **Predefined exceptions with names.**
    About 20 most common errors; raised implicitly (automatically) by the run-time system; need no declaration.
  - **Non-predefined Oracle error**; should be declared if we want to handle it.
- **User-defined errors.** Need to be declared. To raise such an error we use the statement `RAISE`.

Handling error — section EXCEPTION of the block

When an exception is raised, normal execution of the PL/SQL block stops and control transfers to its exception-handling part:

```
DECLARE
   ...
BEGIN
   ...
EXCEPTION
WHEN exception1 THEN                          //handler for exception1
   statements;
WHEN exception2 THEN                          //handler for exception2
   statements;
WHEN OTHERS THEN                  //optional handler for all other errors
   statements;
END;
```

The raised exceptions are catched in exception handlers. Each handler consists of a `WHEN` clause, which specifies an exception, followed by a sequence of statements to be executed when that exception is raised. These statements complete execution of the block; control does not return to where the exception was raised.

Predefined Oracle exceptions

Exception	Number	Description
CASE_NOT_FOUND	ORA-06592	none of the choices in the WHEN clauses of a CASE statement is selected, and there is no ELSE clause
CURSOR_ALREADY _OPENED	ORA-06511	attempt to open an already open cursor
DUP_VAL _ON_INDEX	ORA-00001	attempt to store duplicate values in a column that is constrained by a unique index
INVALID_NUMBER	ORA-01722	in an SQL statement, the conversion of a character string into a number fails
NO_DATA_FOUND	ORA-01403	SELECT INTO statement returns no rows
ROWTYPE _MISMATCH	ORA-06504	incompatible return types
STORAGE_ERROR	ORA-06500	PL/SQL ran out of memory or memory was corrupted
TOO_MANY_ROWS	ORA-01422	SELECT INTO statement returns more than one row
VALUE_ERROR	ORA-06502	an arithmetic, conversion, truncation, or size-constraint error occurs
ZERO_DIVIDE	ORA-01476	attempt to divide a number by zero

User-defined exceptions

Declared in a given PL/SQL program. Must be raised explicitly. Have no specific code and message.
Can be used to react to specific situations during the execution of the program.
Creating and handling user-defined errors:

- Declaration of a variable of type `EXCEPTION` (in section DECLARE);
  `name_of_exception EXCEPTION`
- Raising an exception with the `RAISE` statement;
  `RAISE name_of_exception`
- Handling an exception (in section EXCEPTION).
  `WHEN name_of_exception THEN ...`

Using procedure RAISE_APPLICATION_ERROR

In Oracle PL/SQL we may associate an error-code and a message with an user-defined exception, using the procedure `RAISE_APPLICATION_ERROR`.
To invoke `RAISE_APPLICATION_ERROR`, use the following syntax:

```
RAISE_APPLICATION_ERROR( err_number number,
err_message varchar2, keep_errors boolean)
```

err_number – an integer from the range [-20000, -20999]
err_message – message, of length at most 512
keep_errors, if *true*, then the error is placed on the stack of previous errors, if *false* (by default), then the error replaces all previous errors.

When we invoke the procedure `RAISE_APPLICATION_ERROR` from a PL/SQL subprogram, the subprogram ends and returns a user-defined error number and message to the application.

The error number and message can then be trapped like any other Oracle database error.

Undefined Oracle errors

Each Oracle database error has its own code (error number, which is a negative integer) and the message describing a given error. The Oracle database run-time errors have prefix `ORA`. The compiler errors have prefix `PLS`.

Typical Oracle run-time errors:
violating of the primary or foreign key constraint;
violating of the column constraints (such as `CHECK, NOT NULL`).

To handle error conditions that have no predefined name, use the `OTHERS` handler (in section EXCEPTION) or the pragma `EXCEPTION_INIT`.

When we use the `OTHERS` handler, we can retrieve the error code with the built-in function `SQLCODE`.

The associated error message can be retrieved with either the packaged function `DBMS_UTILTY.FORMAT_ERROR_STACK` or the built-in function `SQLERRM`.

Undefined Oracle errors — using pragma EXCEPTION_INIT

The pragma EXCEPTION_INIT tells the compiler to associate an exception name with an Oracle database error number.

A **pragma** is a compiler directive that is processed at compile time, not at run time.

The pragma EXCEPTION_INIT is put in the declarative part of a PL/SQL block:

```
DECLARE
– declare an EXCEPTION type variable
exception_name EXCEPTION;
– associate an exception name with an Oracle error number
PRAGMA EXCEPTION_INIT(exception_name,
−error_number);
BEGIN
...
EXCEPTION
WHEN exception_name THEN ...
END;
```

Undefined Oracle errors

The errors that are not handled may be logged in special tables. For example:

```
EXCEPTION
...
WHEN OTHERS THEN
error_number := SQLCODE;
error_info := SQLERRM;
INSERT INTO error_table(user_who, data_when,
error_no, info)
VALUES (USER, SYSDATE,
error_number,error_info);
END;
```

Oracle errors

The list of Oracle pre-defined errors can be found here:

`https://docs.oracle.com/cd/B28359_01/appdev.111/b28370/errors.htm#i9355`

All Oracle errors can be found here:

`https://docs.oracle.com/cd/B28359_01/server.111/b28278/toc.htm`

Oracle DML triggers

Triggers are special type PL/SQL programs, which are fired **automatically** as a reaction to certain events in a database.

Triggers are often used to force data integrity and to ensure that the data stored in the database are consistent with the rules imposed when designing the database.

Each DML triggers is connected with a certain table (or view) and is fired as a reaction to the occurence of one of the following events: when the `INSERT, UPDATE` or `DELETE` statement is executed on a given table (or view).

Triggers — usage

The tasks that can be done using triggers:

- data integrity;
- checking the consistent of inserted or updated data with the rules;
- events logging; in the database we can create special log tables, inside which we can store information on changes that are done on more important or sensitive data in our database, the triggers may be used to insert such information to log tables;
- the data can be changed, for instance, properly formatted, before inserting into tables;
- generating data automatically, for instance, for **auto increment** fields, such as primary keys (using sequences);

Triggers — syntax

```
CREATE OR REPLACE TRIGGER name
BEFORE / AFTER / INSTEAD OF
INSERT / UPDATE / DELETE
ON table/view
FOR EACH ROW
```
trigger body

Description:
One of the options `BEFORE / AFTER / INSTEAD OF` should be chosen: when the trigger will be fired:

- **before the event (type BEFORE),**
- **after the event (type AFTER),**
- **instead of the event (type INSTEAD OF)** (this type can be used only for views).

For which DML statement the trigger will be fired: `INSERT / UPDATE / DELETE`.

In the clause `ON` we put the table name or view.

Triggers on the statement or row level

The DML trigger can work at two levels:

- at ROW level, defined with the clause `FOR EACH ROW`,
- at STATEMENT level (the default option).

The ROW trigger is fired for each row processed by the DML statement.
The trigger at the STATEMENT level is fired only once for the DML statement.

For example, if we have trigger for an INSERT statement defined at the ROW level, and 10 rows are inserted, this trigger will be fired 10 times. In the same situation, a STATEMENT level trigger would be fired only once.

Pseudorecords OLD and NEW in row-level DML triggers

In the body of the trigger one can use the same instructions, as in stored procedures.

In row-level triggers, there are also two **pseudorecords** available: `new` and `old`. These pseudorecords can be used to refer to the current or previous values in the inserted, deleted or updated rows.

If the trigger is fired for an INSERT statement, only the pseudorecord new is available; if for DELETE, then we can refer only to the pseudorecord old; in case of UPDATE, both.

For example, if the row-level trigger is working on the table employee, AFTER UPDATE, then in order to refer to the previous value in the column salary, we use `:old.salary`, while to read the new value, to which the salary is changed, we use `:new.salary`.

Restrictions

In DML triggers, we cannot use the instructions that

- change the structure of the table for which the trigger is defined;
- modify the data of the table for which the trigger is defined;
- commit or rollback the transaction;
- call subprograms (procedures, functions) containing such instructions.

DML triggers for multiple events

The DML trigger can be defined to be fired by several types of DML statements.
For example, for table employee to define a compound row-level trigger for statements UPDATE and INSERT we can use:

```
CREATE OR REPLACE TRIGGER new_trigger
AFTER INSERT OR UPDATE
ON employee
FOR EACH ROW
...
```

Conditional predicates: INSERTING, UPDATING, DELETING

The triggering event of a DML trigger can be composed of multiple triggering statements (i.e., INSERT, UPDATE, DELETE).
When one of them fires the trigger, the trigger can determine which one by using these conditional predicates: **I**NSERTING, UPDATING, DELETING, UPDATING(column). A conditional predicate can appear wherever a BOOLEAN expression can appear.
For example, in the `IF` or `CASE` statement, to decide which statement fired the trigger, and execute different block of commands:

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

Managing triggers

- To drop the trigger we use `DROP TRIGGER name`.

Often, we do not want to drop the trigger, but only deactivate it for a period of time (for example, we plan to evaluate a large amount of modifications; triggers extend the time of executing DML statements; we can deactivate the trigger first, do the modifications, and activate it at the end).

- To deactivate the trigger:
  `ALTER TRIGGER name DISABLE`.
- To activate it:
  `ALTER TRIGGER name ENABLE`.
- For a given table we can have several triggers of the same type; in this case they are fired sequentially.

Using sequences in triggers

Let `emp_no_seq` be a sequence. We want to use the values generated by this sequence as the values of the primary key `emp_no` of the table employee.

```
CREATE OR REPLACE TRIGGER insert_emp_no
BEFORE INSERT ON employee
FOR EACH ROW
BEGIN
IF (:NEW.emp_no IS NULL) THEN
:NEW.emp_no := emp_no_seq.NEXTVAL;
END IF;
END;
```

Packages

Packages are used to logically group PL/SQL objects such as subprograms (functions, procedures), types, variables, cursors and exceptions.

Package consists of:

specyfication (interface), and
body (implementation).

In the specifcation we put all declarions of types, variables, constants, cursors, exceptions and subprograms.
In the package body there is an implementation of the subprograms declared in the specification.

Package specification and body are stored separately in the data dictionary.

Packages

Packages can be used to:

1. store connected objects in one place,
2. make logical groups of objects with given functionality,
3. make it easier to design an application: specification and body are indepenently stored and compiled,
4. in the specification, which is public, we have only the interface, and the implementation part is hidden; it is possible to use private declarations in the body (which are not visible outside the package),
5. create variables that can be seen as global variables.

Packages

Advantages of using packages:

- increase performance (at the first call to the package, its whole code is loaded to the memory),
- additional functionality — global variables,
- procedures and functions overloading,
- the code is hidden — only the specification (public interface) is visible to the user, the implementation is hidden.

Packages — public and private declarations

Package specification	Variables declaration Functions and procedures declarations etc.	public
Package body	Variables declaration Functions and procedures definitions etc.	private

Package specification

Package specification:

- Contains the interface.
- Only the data about what is inside the package, but without an implementation (without the code of subprograms).
- Can contain the declarations of publicly visible types, variables, constants, exceptions, cursors.
- Declarations of all publicly visible subprograms.
- It should follow the same rules as the section DECLARE of the anonymous block.

Package specification — syntax

```
CREATE OR REPLACE PACKAGE package_name IS
declarations of publicly visible types,
variables, constants, exceptions, cursors
declarations of functions and procedures
END;
```

Example:

```
CREATE OR REPLACE PACKAGE country IS
TYPE CountryType IS RECORD
( country_ID CHAR(2),
country_name VARCHAR2(40),
region_ID INT);
CURSOR C_Country(r_id INT) RETURN CountryType;
vcount INT;
PROCEDURE insert_country(c_id IN CHAR, cname
VARCHAR2, cregion_id INT DEFAULT 1);
PROCEDURE display_country(c_id INT DEFAULT 1);
END;
```

Package specification

- In the specification we can initialize the variables, for otherwise they get NULL.
- All declarations of the specification are publicly visible to all the users that have the privilege to access the package.
- There are no restrictions on the order of declarations, but the elements, that we refer to, should be declared earlier.
- Declarations of subprograms in the specifications does not contain its code.
- It is possible to create packages that do not contain subprograms, but only cursors or types, or variables, etc.

Package body

- The package body is stored separately in the data dictionary.
- It contains the codes of implementations for all declarations from the specification.
- Each of the declarations from the specification must have its definition in the body.
- For procedures and functions, the name of the subprogram, parameters (names, order, data types) must be exactly the same in the specification and body.
- In the body we can put additional declarations, which were not present in the specification, such as types, variables, cursors or exceptions.
- The elements that are declared only in the body part are not visible outside of the package, only in its body.

Package — body

Body:

```
CREATE OR REPLACE PACKAGE BODY package_name IS
declarations of locally visible types,
variables, constants, exceptions, cursors
definitions of functions and procedures
BEGIN
initialization part of the package / optional
END;
```

np.

```
CREATE OR REPLACE PACKAGE BODY country IS
CURSOR C_Country(r_id INT) RETURN CountryType IS
SELECT * FROM countries WHERE region_id=r_id;
PROCEDURE insert_country(c_id IN CHAR, cname
VARCHAR2, cregion_id INT DEFAULT 1) IS
BEGIN
INSERT INTO countries VALUES
(c_id,cname,cregion_id);
END;
PROCEDURE display_country(c_id INT DEFAULT 1) IS
BEGIN
FOR item IN C_Country(r_id)
LOOP
Dbms_Output.put_line( item.Country_id||'
'||item.Country_name);
END LOOP;
END;
END;
```

Example (from Oracle documentation)

Specification:

```
CREATE OR REPLACE PACKAGE emp_actions IS
TYPE EmpRecTyp IS RECORD (emp_id INT, salary
REAL);
CURSOR desc_salary RETURN EmpRecTyp;
PROCEDURE hire_employee (
ename VARCHAR2,
job VARCHAR2,
mgr NUMBER,
sal NUMBER,
comm NUMBER,
deptno NUMBER);
PROCEDURE fire_employee (emp_id NUMBER);
END;
```

Body:

```
CREATE OR REPLACE PACKAGE BODY emp_actions IS
CURSOR desc_salary RETURN EmpRecTyp IS
SELECT empno, sal FROM emp ORDER BY sal DESC;
PROCEDURE hire_employee (
ename VARCHAR2,
job VARCHAR2,
mgr NUMBER,
sal NUMBER,
comm NUMBER,
deptno NUMBER) IS
BEGIN
INSERT INTO emp VALUES (empno_seq.NEXTVAL, ename,
job, mgr, SYSDATE, sal, comm, deptno);
END;
PROCEDURE fire_employee (emp_id NUMBER) IS
BEGIN
DELETE FROM emp WHERE empno = emp_id;
END;
END;
```

When we want to call a function or procedure which is defined in the package from outside the package, we should precede it with the package name (using dot notation):

```
EXECUTE country.display_country(2);
```

To compile a package, use:

```
ALTER PACKAGE name COMPILE PACKAGE | PACKAGE
BODY;
```

To remove a package, use:

```
DROP PACKAGE | PACKAGE BODY name;
```

Using packages for declaring global variables

If in the package specification there are no declarations of cursors or subprograms (procedures, functions), but only types, constants, variables and exceptions, we do not have to create the package body; for example:

```
CREATE OR REPLACE PACKAGE customerData IS
TYPE CustDataType IS RECORD
( n customer.cust_no%type,
name customer.customer%type,
phone customer.phone_no%type,
address VARCHAR2(60),
city VARCHAR2(35),
country customer.country%type,
code customer.postal_code%type);
min_payment CONSTANT NUMBER:= 100.00;
vcount INT;
no_paymenet EXCEPTION;
END;
```

In such a package we can define types and global variables, which can be referred to within a current session from other subprograms or triggers.

Overloading in packages

- In the package we can functions (or procedures) with the same name, but different parameters.
- It allows to use the same function (or procedure) to perform operations on objects of different types.
- The parameters should be distinct in number, order or types.

For example.

```
CREATE OR REPLACE PACKAGE emp_pack IS
FUNCTION avg_sal(id_dept NUMBER) RETURN
NUMBER;
FUNCTION avg_sal(id_dept NUMBER, id_manager
NUMBER) RETURN NUMBER;
END;
```

Overloading - restrictions

- We cannot create two subprograms with the same name, if their declarations differ only by the parameters names or type:

```
PROCEDURE A (p_1 IN NUMBER);
PROCEDURE A (p_1 OUT NUMBER);
```

- We cannot create two functions with the same name, if their declarations differ only by the type of returning value:

```
FUNCTION B RETURN NUMBER;
FUNCTION B RETURN DATE;
```

- Parameters cannot be distinguish if they belong to the same family of types, for example, such an overloading is not allowed:

```
PROCEDURE C (p_1 IN CHAR);
PROCEDURE C (p_1 IN VARCHAR2);
```