Programming in Oracle with PL/SQL

Procedural Language Extension to SQL

Useful links (couple of slides are from here) :

http://plsql-tutorial.com/

http://www.ora-code.com/

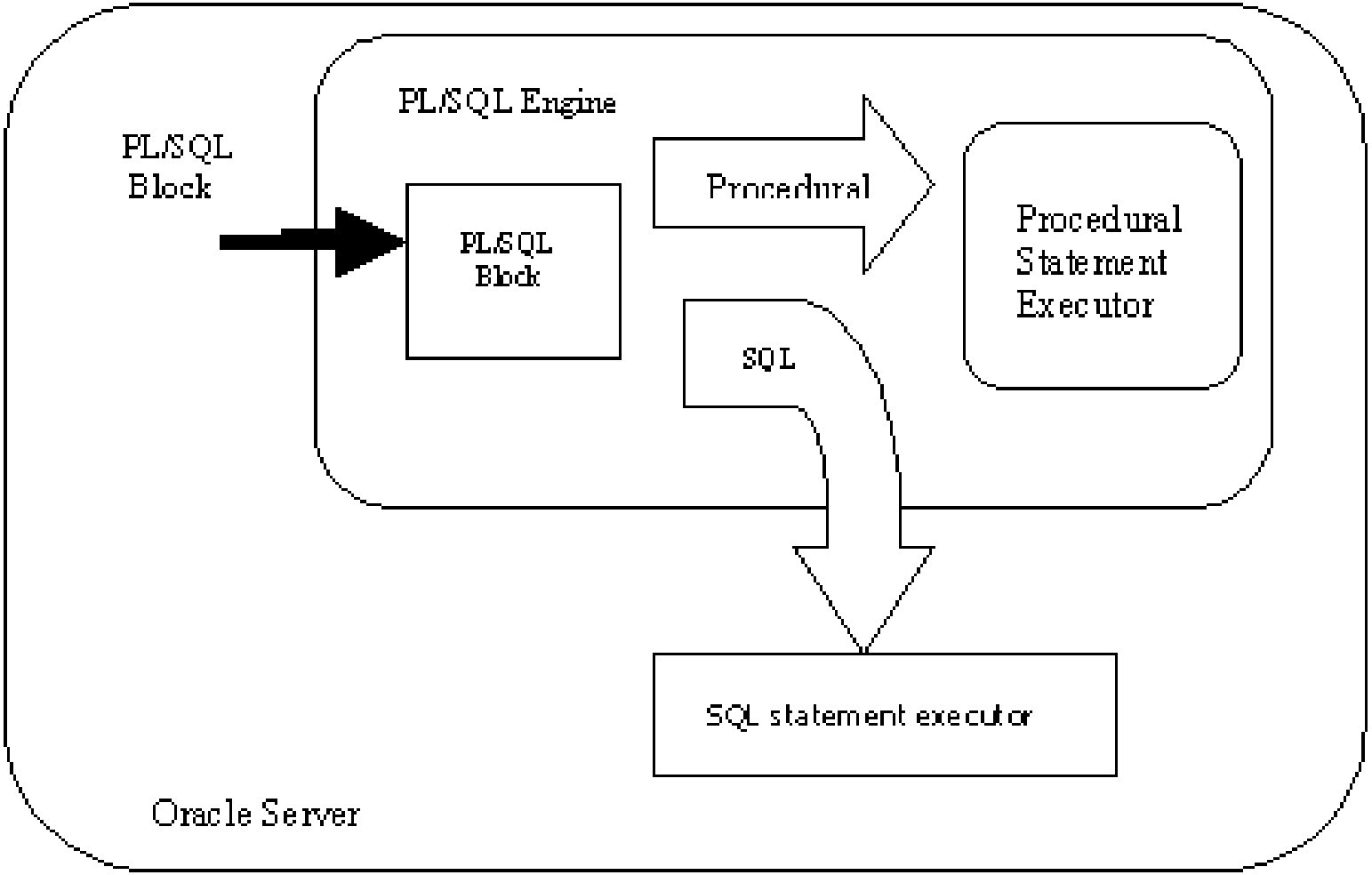
http://www.exforsys.com/tutorials/pl-sql.html

http://www.java2s.com/Tutorial/Oracle/0560__Trigger/Catalog0560__Trigger.htm

PL/SQL

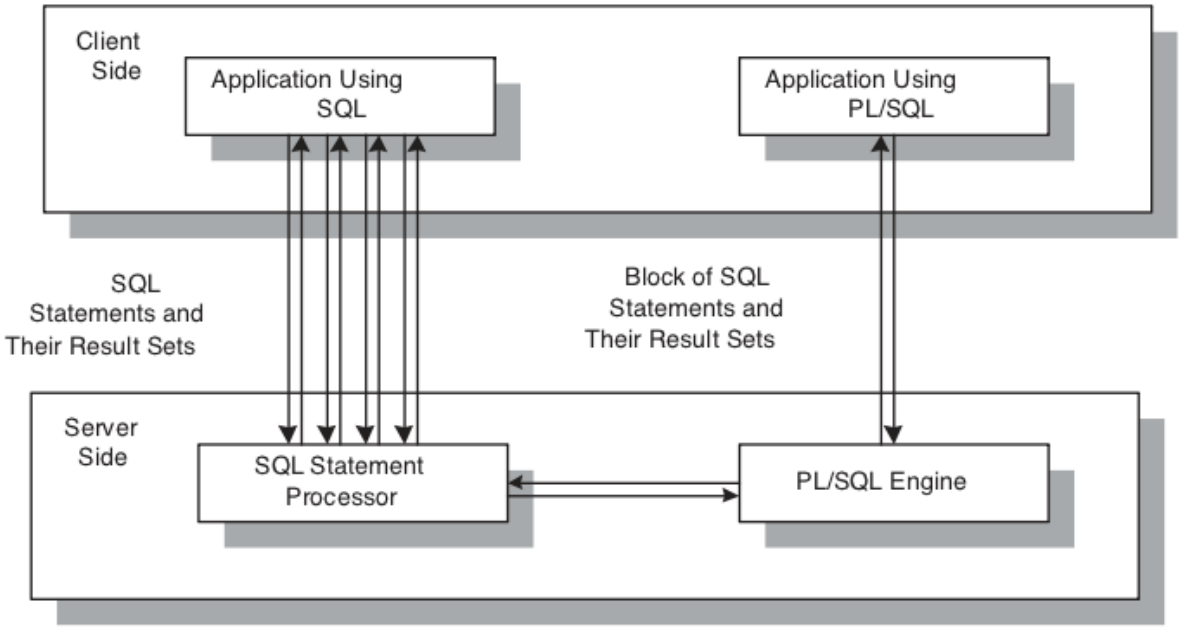
- Allows using general programming tools with SQL, for example: loops, conditions, functions, etc.
- This allows a lot more freedom than general SQL, and is lighter-weight than JDBC.
- We write PL/SQL code in a regular file, for example PL.sql, and load it with @PL in the sqlplus console.

PL/SQL


PL/SQL Engine
PL/SQL Block
PL/SQL Block
Procedural
Procedural Statement Executor
SQL
SQL statement executor
Oracle Server

PL/SQL in client-server architecture

When a SQL statement is issued on the client computer, the request is made to the database on the server, and the result set is sent back to the client.



As a result, a single SQL statement causes two trips on the network. If multiple SELECT statements are issued, the network traffic increases significantly very fast. For example, four SELECT statements cause eight network trips.

If these statements are part of the PL/SQL block, they are sent to the server as a single unit. The SQL statements in this PL/SQL program are executed at the server and the result set is sent back as a single unit.

There is still only one network trip made as is in case of a single SELECT statement.

PL/SQL Blocks

- PL/SQL code is built of Blocks, with a unique structure.
- There are two types of blocks in PL/SQL:

1. **Anonymous Blocks:** have no name (like scripts)
   - can be written and executed immediately in SQLPLUS
   - can be used in a trigger
2. **Named Blocks:**
   - Procedures
   - Functions

PL/SQL Blocks

Named blocks are used when creating subroutines.

These subroutines are procedures, functions, and packages.

The subroutines can be stored in the database and referenced by their names later on.

In addition, subroutines can be defined within the anonymous PL/SQL block.

Anonymous PL/SQL blocks do not have names. As a result,they cannot be stored in the database and referenced later.

Anonymous Block Structure:

DECLARE (optional)

/* Necessary variables are declared in this section*/

BEGIN (mandatory)

/* This section contains executable statements of SQL and PL/SQL (what the block DOES!)*/

EXCEPTION (optional)

/* Here you define the actions that take place if an exception is thrown during the run of this block */

END; (mandatory)

/

Always put a new line with only a / at the end of a block! (This tells Oracle to run the block)

A correct completion of a block will generate the following message:

PL/SQL procedure successfully completed

Printing Output

- You need to use a function in the DBMS_OUTPUT package in order to print to the output
- If you want to see the output on the screen, you must type the following (before starting):

```
set serveroutput on
```

- Then print using
  - dbms_output. put_line*(your_string);*
  - dbms_output.put(*your_string*);

Hello Word Example

Oracle SQL Developer : db2014

File Edit View Navigate Run Source Team Tools Window Help

Connections

- Connections
  - db2014
    - Tables (Filtered)
    - Views
    - Editioning Views
    - Indexes
    - Packages
    - Procedures
    - Functions
    - Queues
    - Queues Tables
    - Triggers
    - Crossedition Triggers
    - Types

Reports

- All Reports
  - Data Dictionary Reports
  - Data Modeler Reports
  - OLAP Reports
  - TimesTen Reports
  - User Defined Reports

Start Page | db2014

db2014

Worksheet | Query Builder

```
set serveroutput on;
begin
dbms_output.put_line('Hello');
end;
/
```

Script Output

Task completed in 0,005 seconds

```
anonymous block completed
Hello
```

Line 5 Column 2 | Insert | Modified | Windows: CF

DECLARE

Syntax

identifier [CONSTANT] *datatype* [NOT NULL]
[:= | DEFAULT *expr*];

Examples

Notice that PL/SQL includes all SQL types, and more…

```
Declare
  birthday    DATE;
  age         NUMBER(2) NOT NULL := 27;
  name        VARCHAR2(13) := 'Levi';
  magic       CONSTANT NUMBER := 77;
  valid       BOOLEAN NOT NULL := TRUE;
```

Declaring Variables with the %TYPE Attribute

This variable can accept one record, same type as

Tuple in the table Emp

Examples

```
DECLARE

   v_hold_one_row      Emp%ROWTYPE;
   v_name            Emp.name%TYPE;
   v_fav_emp           VARCHAR2(30);
```

Accessing column name in table Emp

Declaring Variables with the %ROWTYPE Attribute

Accessing table Reserves

Declare a variable with the type of a ROW of a table.

```
emp_record    emp%ROWTYPE;
```

And how do we access the fields in emp_record?

```
emp_record.id:=987;
emp_record.mid:=877;
```

Creating a PL/SQL Record

A record is a type of variable which we can define (like 'struct' in C or 'collection' in Java)

```
DECLARE
  TYPE emp_record_type IS RECORD
    (name            VARCHAR2(10),
     id          NUMBER(3),
     job         VARCHAR2(9),
     manager_id      NUMBER(3));

      emp_record  emp_record_type;
...
BEGIN
   emp_record.name:='peter';
  emp_record.manager_id:=45;
...
```

Cursor

- A cursor is a temporary work area created in the system memory when a SQL statement is executed. A cursor contains information on a select statement and the rows of data accessed by it.
- This temporary work area is used to store the data retrieved from the database, and manipulate this data. A cursor can hold more than one row, but can process only one row at a time. The set of rows the cursor holds is called the active set.
- There are two types of cursors in PL/SQL:
  - Implicit cursors
  - Explicit cursors

SQL Cursor

SQL cursor is automatically created after each SQL query. It has 4 useful attributes:

SQL%ROWCOUNT	**Number of rows affected by the most recent SQL statement (an integer value).**
SQL%FOUND	**Boolean attribute that evaluates to TRUE if the most recent SQL statement affects one or more rows.**
SQL%NOTFOUND	**Boolean attribute that evaluates to TRUE if the most recent SQL statement does not affect any rows.**
SQL%ISOPEN	**Always evaluates to FALSE because PL/SQL closes implicit cursors immediately after they are executed.**

Creating a Cursor

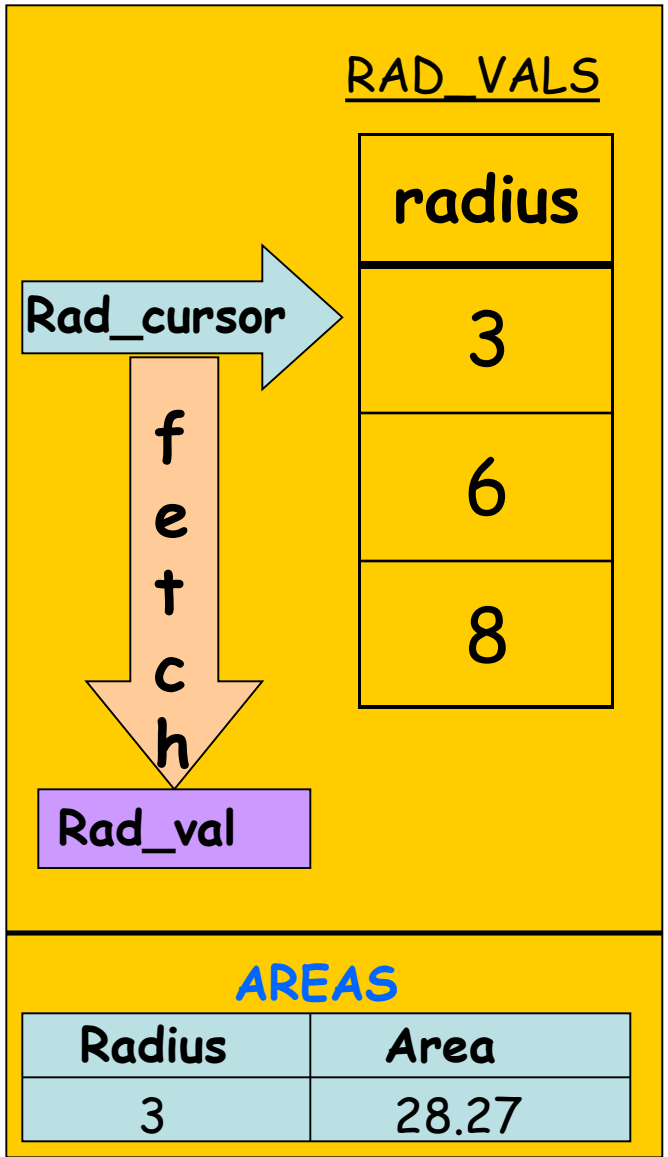
- We create a Cursor when we want to go over a result of a query (like ResultSet in JDBC)
- Syntax Example:

```
DECLARE
   cursor c is select * from emp;
   emp_duplicate emp_record_type;
 BEGIN
     open c;
     fetch c into emp_duplicate;
```

emp_duplicate is a variable that can hold a row from the emp table

Here the first row of emp inserted into **emp_duplicate**

Example (will be on seminars)




```
DECLARE
  Pi constant NUMBER(8,7) := 3.1415926;
  area NUMBER(14,2);
  cursor rad_cursor is select * from
   RAD_VALS;
  rad_value rad_cursor%ROWTYPE;

BEGIN
  open rad_cursor;
  fetch rad_cursor into rad_val;
  area:=pi*power(rad_val.radius,2);
  insert into AREAS values
   (rad_val.radius, area);
  close rad_cursor;
END;
/
```

SELECT Statements

```
DECLARE
  v_ename     VARCHAR2(10);
  v_id        NUMBER(4);
BEGIN
  SELECT     ename, empno
    INTO     v_ename, v_id
    FROM     emp
   WHERE     empno = 7900;
END;
/
```

- INTO clause is required.
- Query must return exactly one row.
- Otherwise, a NO_DATA_FOUND or TOO_MANY_ROWS exception is thrown

Conditional logic

Condition:

```
If <cond>
    then <command>
 elsif <cond2>
    then <command2>
 else
        <command3>
 end if;
```

Nested conditions:

```
If <cond>
   then
      if <cond2>
         then
            <command1>
       end if;
 else <command2>
 end if;
```

```
set serveroutput on;
declare
  v_x number(7,2);
  updated_salary NUMBER;
BEGIN
  SELECT  SAL
  INTO    v_x
  FROM    emp
  WHERE   empno=7788;
      dbms_output.put_line(The salay before raise: '||v_x);
  IF v_x < 3100 THEN
       updated_salary:=v_x*1.1;
  UPDATE emp
  SET sal = updated_salary
  WHERE empno = 7788;
     dbms_output.put_line(The salary after the raise: '|| updated_salary);
   ELSE  dbms_output.put_line('Salary has not changed.');
   END IF;
END;
/
update emp set sal=3000 where empno=7788;
```

IF-THEN-ELSIF Statements

```
. . .
IF mark<2 THEN
  v_message := 'You failed';
ELSIF mark < 4 THEN
  v_message := 'Not bad';
ELSE
  v_message := 'Pretty good';
END IF;
. . .
```

Loops: Simple Loop

```
create table number_table(
     num NUMBER(10)
);
```

```
DECLARE
  i  number_table.num%TYPE := 1;
BEGIN
  LOOP
    INSERT INTO number_table
      VALUES(i);
    i := i + 1;
    EXIT WHEN i > 10;
  END LOOP;
END;
```

Loops: Simple Loop

```
Set serveroutput on
begin
for i in 1..10 loop
dbms_output.put_line(i);
dbms_output.put_line
(this is a constant' || ' a');
end loop;
end;
/
```

Loops: Simple Loop with cursor

```
create table number_table(
      num NUMBER(10)
);
```

```
DECLARE
  cursor c is select * from number_table;
  cVal c%ROWTYPE;--cursor type!!
BEGIN
  open c;
  LOOP
    fetch c into cVal;
    EXIT WHEN c%NOTFOUND;
    insert into duplicate values(cVal.num*2);
  END LOOP;
  close c;
END;
```

Loops: FOR Loop

```
DECLARE
  i        number_table.num%TYPE;
BEGIN
  FOR i IN 1..10 LOOP
    INSERT INTO number_table VALUES(i);
  END LOOP;
END;
```

Notice that i is incremented automatically

Loops: For Cursor Loops

```
DECLARE
  cursor c is select * from number_table;

BEGIN
   for num_row in c loop
       insert into doubles_table
                       values(num_row.num*10);
   end loop;
END;
/
```

Notice that a lot is being done implicitly: declaration of num_row, open cursor, fetch cursor, the exit condition

Loops: Simple Loop with cursor

```
DECLARE
   v_ename   emp.ename%TYPE;

    CURSOR c_managers IS
       SELECT ename FROM emp WHERE job = 'MANAGER';
   BEGIN
     OPEN c_managers ;
     LOOP

       FETCH c_managers  INTO v_ename;
       DBMS_OUTPUT.put_line(v_ename);

       EXIT WHEN c_managers%NOTFOUND;

END LOOP;
END;
/
```

Loops: WHILE Loop

```
DECLARE
TEN number:=10;
i       number_table.num%TYPE:=1;
BEGIN
  WHILE i <= TEN LOOP
      INSERT INTO number_table
      VALUES(i);
      i := i + 100;
  END LOOP;
END;
```

Reminder- structure of a block

```
DECLARE                    (optional)
   /* Here you declare the variables you
    will use in this block */
BEGIN                      (mandatory)
   /* Here you define the executable
    statements (what the block DOES!)*/
EXCEPTION     (optional)
   /* Here you define the actions that take
    place if an exception is thrown during
    the run of this block */
END;                       (mandatory)
/
```

Trapping Exceptions

- Here we define the actions that should happen when an exception is thrown.
- Example Exceptions:
  - NO_DATA_FOUND
  - TOO_MANY_ROWS
  - ZERO_DIVIDE
- When handling an exception, consider performing a rollback

Trapping Exceptions

- The *exception-handling section* is the last section of the PL/SQL block.

- This section contains statements that are executed when a runtime error occurs within a block.

- Runtime errors occur while the program is running and cannot be detected by the PL/SQL compiler.

```
EXCEPTION
WHEN NO_DATA_FOUND THEN
DBMS_OUTPUT.PUT_LINE
(' There is no student with student
id 123 ');
END;
```

```
DECLARE
   num_row  number_table%ROWTYPE;
BEGIN
   select *
   into num_row
   from number_table;
   dbms_output.put_line(1/num_row.num);

EXCEPTION
   WHEN NO_DATA_FOUND THEN
     dbms_output.put_line('No data!');
   WHEN TOO_MANY_ROWS THEN
     dbms_output.put_line('Too many!');
   WHEN OTHERS THEN
     dbms_output.put_line(‘Error’);
end;
```

Exceptions Handling

```
DECLARE
v_empno NUMBER := &empno;
v_ename VARCHAR2(10);
v_job VARCHAR2(9);
BEGIN
SELECT ename, job
INTO v_ename, v_job
FROM emp
WHERE empno= v_empno;
DBMS_OUTPUT.PUT_LINE
('Emp data: '||v_ename||'
'||v_job);
EXCEPTION
WHEN NO_DATA_FOUND THEN
DBMS_OUTPUT.PUT_LINE('There is no such an
employee');
END;
/
```

User-Defined Exception

```
DECLARE
   e_number1  EXCEPTION;
   cnt        NUMBER;
BEGIN
   select count(*)
   into cnt
   from number_table;

   IF cnt = 1 THEN RAISE e_number1;
   ELSE dbms_output.put_line(cnt);
   END IF;

EXCEPTION
   WHEN e_number1 THEN
     dbms_output.put_line('Count = 1');
end;
```

Functions and Procedures

- Up until now, our code was in an anonymous block
- It was run immediately
- It is useful to put code in a function or procedure so it can be called several times
- Once we create a procedure or function in a Database, it will remain until deleted (like a table).

Functions and Procedures

- The header specifies
- : name and parameter list
  : return type (function headers)
  : any of the parameters can have a default value
  : modes - IN, OUT, IN OUT

```
CREATE FUNCTION get_department_no
( p_dept_name IN VARCHAR2 := null)
RETURN NUMBER
IS
DECLARE
   - - - - - - - -
BEGIN
  - - - - - - - - - -
   RETURN(l_dept_no);
EXCEPTION
   - - - - - - - - - -
END;
```

Function example

```
CREATE PROCEDURE department_change
(
 p_dept_number IN            NUMBER

p_new_name      IN OUT    VARCHAR2
)
AS
DECLARE

BEGIN     ……….. .
END;
```

- Procedure example

Creating Procedures

```
CREATE [OR REPLACE] PROCEDURE
procedure_name
 [(parameter1 [mode1] datatype1,
  parameter2 [mode2] datatype2,
  . . .)]
IS|AS
PL/SQL Block;
```

- Modes:
  - IN: procedure must be called with a value for the parameter. Value cannot be changed (like pass by value)
  - OUT: procedure must be called with a variable for the parameter. Changes to the parameter are seen by the user (like pass by reference)
  - IN OUT: value can be sent, and changes to the parameter are seen by the user
- Default Mode is: IN

Example- what does this do?

Table mylog

who	logon_ num
Pete	3
John	4
Joe	2

```
create or replace procedure
num_logged
(person IN mylog.who%TYPE,
 num OUT mylog.logon_num%TYPE)
IS
BEGIN
    select logon_num
    into num
    from mylog
    where who = person;
END;
/
```

Calling the Procedure

```
declare
    howmany  mylog.logon_num%TYPE;
begin
    num_logged(‘John',howmany);
    dbms_output.put_line(howmany);
end;
/
```

Errors in a Procedure

- When creating the procedure, if there are errors in its definition, they will not be shown
- To see the errors of a procedure called *myProcedure*,
- See Compiler Log window in sqldeveloper or
- type

  SHOW ERRORS PROCEDURE *myProcedure*
- For functions, type

  SHOW ERRORS FUNCTION *myFunction*

Creating a Function

- Almost exactly like creating a procedure, but you supply a return type

```
CREATE [OR REPLACE] FUNCTION
function_name
 [(parameter1 [mode1] datatype1,
  parameter2 [mode2] datatype2,
  . . .)]
RETURN datatype
IS|AS
PL/SQL Block;
```

Creating a function (see seminars):

```
create or replace function squareFunc(num in number)
return number
is
BEGIN
return num*num;
End;
/
```

Using the function:

```
BEGIN
dbms_output.put_line(squareFunc(3.5));
END;
/
```

Triggers

- Triggers are special procedures which we want activated (fire!) when someone has performed some action on the DB.
- For example, we might define a trigger that is executed when someone attempts to insert a row into a table, and the trigger checks that the inserted data is valid.

Triggers

- Triggers are stored procedures that execute automatically when something (event) happens in the database:
  - : data modification (INSERT, UPDATE or DELETE)
  - : schema modification
  - : system event (user logon/logoff)

- Types of triggers
  - : row-level triggers
  - : statement-level triggers
  - : BEFORE and AFTER triggers
  - : INSTEAD OF triggers (used for views)
  - : schema triggers
  - : database-level triggers

Triggers

- Trigger action can be any type of Oracle stored procedure
- PL/SQL trigger body is built like a PL/SQL procedure
- The type of the triggering event can be determined inside the trigger using conditional predicators

  IF inserting THEN … END IF;

- Old and new row values are accessible via :old and :new qualifiers
- If for each row clause is used the trigger will be a row-level one

Types of PL/SQL Triggers

There are two types of triggers based on the which level it is triggered.

1) **Row level trigger** - An event is triggered for each row updated, inserted or deleted.

2) **Statement level trigger** - An event is triggered for each sql statement executed.

PL/SQL Trigger Execution Hierarchy

The following hierarchy is followed when a trigger is fired.

1) BEFORE statement trigger fires first.
2) Next BEFORE row level trigger fires, once for each row affected.
3) Then AFTER row level trigger fires once for each affected row. This events will alternate between BEFORE and AFTER row level triggers.
4) Finally the AFTER statement level trigger fires.

How To know information about triggers?

We can use the data dictionary view 'USER_TRIGGERS' to obtain information about any trigger.
The below statement shows the structure of the view
'USER_TRIGGERS '

```
DESC USER_TRIGGERS ;
```

```
SELECT * FROM user_triggers
WHERE trigger_name =
'Before_Update_Stat_product ;'
```

The above sql query provides the header and body of the trigger 'Before_Update_Stat_product.'
You can drop a trigger using the following command.

```
DROP TRIGGER trigger_name ;
```

Triggers

CREATE [OR REPLACE] TRIGGER trigger_name}
BEFORE | AFTER | INSTEAD OF{
} INSERT [OR] | UPDATE [OR] | DELETE {
]OF col_name [ON table_name]
REFERENCING OLD AS o NEW AS n [
]FOR EACH ROW [WHEN (condition (BEGIN))
---sql statements END ;

Trigger example (log files)

```
CREATE TABLE T4 (a INTEGER, b CHAR(10));
insert into T4 values(4, ‘kicsi');
insert into T4 values(3, 'kicsi');
insert into T4 values(20, 'nagy');
select * from T5;


CREATE TABLE T5 (c CHAR(10), d INTEGER);
CREATE TRIGGER trig1
    AFTER INSERT ON T4
    REFERENCING NEW AS newRow
    FOR EACH ROW
    WHEN (newRow.a <= 10)
    BEGIN
        INSERT INTO T5 VALUES
                        (:newRow.b,
:newRow.a);
    END trig1;
Run;
```

```
1 rows inserted.
1 rows inserted.
1 rows inserted.
C          D
---------- ----------------------
kicsi      4
kicsi      3
kicsi      5
```

Trigger example (domain check and error message)

```
create table Person
(name varchar(20), age int);

CREATE TRIGGER PersonCheckAge
AFTER INSERT OR UPDATE OF age ON
Person
FOR EACH ROW
BEGIN
    IF (:new.age < 0) THEN
        RAISE_APPLICATION_ERROR(-
20000, 'no negative age allowed ;('
    END IF ;
END ;
RUN ;
```

Trigger example

```
CREATE OR REPLACE TRIGGER
Print_salary_changes
BEFORE DELETE OR INSERT OR UPDATE ON
Emp_tab

FOR EACH ROW WHEN (new.Empno > 0)

DECLARE sal_diff number;

BEGIN sal_diff := :new.sal -
:old.sal; dbms_output.put('Old
salary: ' ||:old.sal);
dbms_output.put(' New salary: '
|:new.sal); dbms_output.put_line('
Difference'||sal_diff);

END; /
```

SUBSTITUTION VARIABLES

- SQL Developer allows a PL/SQL block to receive input information with the help of substitution variables.
- Substitution variables cannot be used to output the values because no memory is allocated for them.
- SQL Developer will substitute a variable before the PL/SQL block is sent to the database.
- Substitution variables are usually prefixed by the ampersand(&) character or double ampersand (&&) character.

Example: SUBSTITUTION VARIABLES

```
--5_nevbekeres- not frequently used

SET SERVEROUTPUT ON
      ACCEPT nev PROMPT 'Kerem adja meg a nevét: '
      DECLARE
            szoveg varchar2(50);
      BEGIN
            szoveg := CONCAT('&nev',' sikeresen
végrehajtotta a programot!');
            DBMS_OUTPUT.PUT_LINE (szoveg);
      END;
```

Oracle SQL Developer : db2014_1

File Edit View Navigate Run Source Team Tools Window Help

Connections

- Connections
  - db2014
    - Tables (Filtered)
    - Views
    - Editioning Views
    - Indexes
    - Packages
    - Procedures
    - Functions
    - Queues
    - Queues Tables
    - Triggers
    - Crossedition Triggers
    - Types
    - Sequences
    - Materialized Views

Reports

- All Reports
  - Data Dictionary Reports
  - Data Modeler Reports
  - OLAP Reports
  - TimesTen Reports
  - User Defined Reports

Start Page | hello.sql | nevbekeres.sql

db2014_1

Worksheet | Query Builder

```
accept nev prompt 'Kérem adja meg a nevét!'
declare
  tarolo varchar2(50);
BEGIN
  tarolo := concat('&nev',' sikeresen végrehajtotta a programot.');
  dbms_output.put_line (tarolo);
END;
/
```

Dbms Output

Buffer Size: 20000

db2014_1

```
zseta sikeresen végrehajtotta a programot.
```

Opened nodes (101); Saved files(2) | Line 9 Column 4 | Insert | Modified | Windows: Cr

HOW PL/SQL GETS EXECUTED

- Every time an anonymous block is executed, the code
- is sent to the PL/SQL engine on the server where it is compiled.

- The named PL/SQL block is compiled only at the time of its creation, or if it has been changed.

- The compilation process includes syntax checking, binding and p-code generation.

- Syntax checking involves checking PL/SQL code for syntax or compilation errors.

- Once the programmer corrects syntax errors, the compiler can assign a storage address to program variables that are used to hold data for Oracle. This process is called Binding.

HOW PL/SQL GETS EXECUTED

- After binding, p-code is generated for the PL/SQL block.
  - P-code is a list of instructions to the PL/SQL engine.
  - For named blocks, p-code is stored in the database, and it is used the next time the program is executed.
- Once the process of compilation has completed successfully, the status for a named PL/SQL block is set to VALID, and also stored in the database.
- If the compilation process was not successful, the status for a named PL/SQL block is set to INVALID

Packages

- Logically connected Functions, Procedures, Variables can be put together in a package
- In a package, you can allow some of the members to be "public" and some to be "private"
- There are also many predefined Oracle packages
- Won't discuss packages in this course

Packages

- Many PL/SQL packages are provided within the Oracle Server
- Extend the functionality of the database
- Some example of such packages:
  - **: DBMS_JOB - for scheduling tasks**
  - **: DBMS_OUTPUT - display messages to the session output device**
  - **: UTL_HTTP - makes HTTP(S) callouts**
    Note: **can be used for accessing a web-service from the database**
  - **: PL/SQL web toolkit (HTP, HTF, OWA_UTIL, etc.)**
    Note: **can be used for building web-based interfaces**
    e.g. https://edms.cern.ch