Advanced Database

Course Code - ST2BDA

Specialty : IT

Course Credits: Jean-Charles Huet & Hanen OCHI

Dr. Lilia Sfaxi
Assistant professor

efrei PARIS
Engineering school of digital technologies

CM2 – PL/SQL

Databases for IT Engineers

Dr. Lilia Sfaxi
Assistant Professor

01

Introduction

01

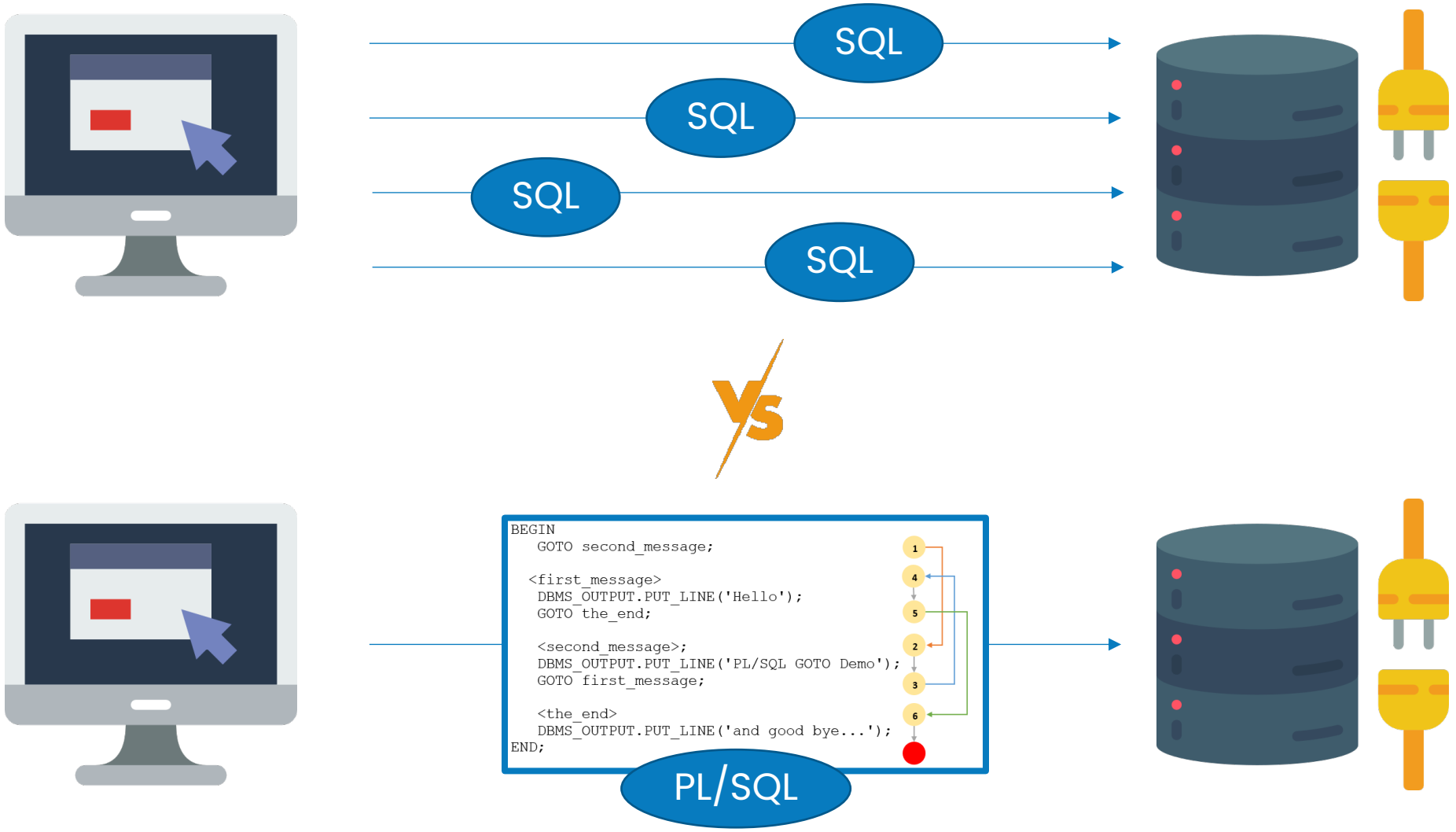
What is PL / SQL?

- The **Procedural Language for SQL**
- The PL / SQL is a language providing a procedural interface to the Oracle DBMS.
- It integrates SQL.
- It allows algorithmic processes (which SQL does not allow).
- It offers the most conventional programming mechanisms (loops, conditions, variables, etc.).

Advantages of PL / SQL

- PL / SQL Complete SQL that is not procedural, it offers:
  - Repetitive structures (WHILE, FOR, ...).
  - Conditional structures (IF THEN ELSE, CASE).
  - The declaration of cursors and tables
  - The declaration of variables
  - Assigning values to variables
  - Connections (GOTO, EXIT)
  - Exceptions (EXCEPTION).

Advantages of PL / SQL

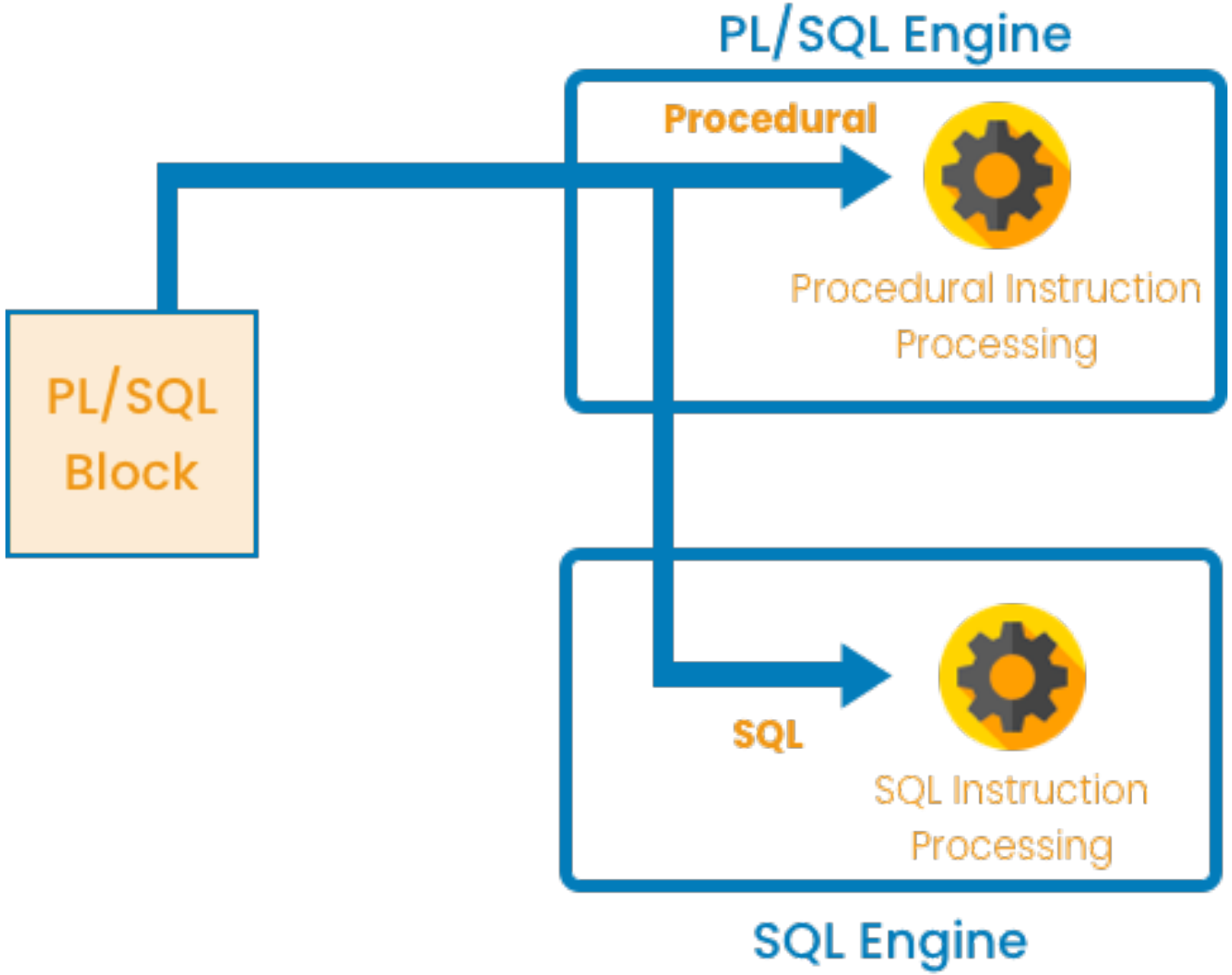


02

The PL/SQL Block

02

PL / SQL Environment



Structure of a PL / SQL Instruction

```
DECLARE
        Variables, cursors, exceptions,…
BEGIN
        SQL and PL/SQL instructions
        Possible nested blocks
EXCEPTION
        Exception Handling (error handling)
END ;
/
```

- Each PL / SQL block may consist of three sections:
  - An optional section for declaration and initialization of types, variables and constants DECLARE.
  - A mandatory section containing the execution instructions BEGIN.
  - An optional section Error Handling EXCEPTION.

Structure of a PL / SQL Instruction

Minimal Block

```
BEGIN
        null;
END ;
/
```

- The key word BEGIN determines the beginning of the executable code section.
- The key word END; indicates the end of the executable code section.
- Only one instruction in this block: Null; which generates no action.
  - This PL / SQL does absolutely nothing!

Example of a PL / SQL Instruction

```
DECLARE
        var_x    VARCHAR2 (5);
BEGIN
        SELECT column_name
        INTO var_x
        FROM table;
EXCEPTION
        WHEN exception_name THEN
        ……;
END  ;
/
```

PL / SQL blocks Type

- Named Blocks
  - Blocks with a header or a label
  - Can be:
    - Procedure
    - Function
    - Package
    - Trigger
- Anonymous Block
  - Blocks with no headers are known as anonymous blocks.
  - These blocks do not form the body of a function or triggers or procedure.

Syntax of a PL / SQL block

- **PL / SQL recognizes the SQL functions:**
  - The functions on numbers, strings, dates.
  - Data type conversion functions.

Examples:

- **Compose new employee's address:**
  - V_AdrComplete : = V_Rue || ' ' || V_Ville || ' ' || V_CodePostal ;
- **Convert the name in capital letters:**
  - V_Nom: = UPPER (V_Nom);
- **Extract a portion of a string:**
  - V_chr: = SUBSTR ('PL / SQL', 4,3);
- **Replace a string with another:**
  - V_chr: = REPLACE ('Serv1 / Prod / tb_client', 'Broadcast' Valid);

Syntax of a PL / SQL block

- **Operators in PL / SQL:**
  - Logic
  - Arithmetic
  - Concatenation
  - Parenthesis to control the order of operations
  - Exponentiation operator **

Same as SQL

- **Insert a comment:**
  - Precede a comment written on one line with '--'.
  - Place a comment written on several lines between symbols: / * And * /.
- **Example:**

```
DECLARE
          v_sal     NUMBER (9,2);
BEGIN
          / * This is a comment that can be
written on several lines * /
END ; -- This is a comment on one line
/
```

Declaration Section

- The declarative section (optional) of a block begins with the keyword DECLARE.
- It contains all declarations of the variables used locally by the executable section, and their possible initialization.
- This section can not contain executable instructions.
  - However, it is possible to define in this section procedures or functions containing an executable section.
- **A variable must be declared before it can be used in the executable section**

Execution Section

- Delimited by keywords BEGIN and END;
- Contains instructions to be run on the PL/SQL program
  - statements and iteration instructions,
  - the call of procedures and functions,
  - the use of native functions,
  - SQL commands, etc…
- Each statement must be followed by '; '.

```
BEGIN
        null;
END ;
/
```

Error Section

- Error Handling section (optional) starts with the keyword EXCEPTION.
- It contains executable code used to manage errors.
- When an error occurs in the execution, the program is stopped and the error code is transmitted to this section.

```
DECLARE
       ch VARCHAR2(15) := 'Hello World' ;
BEGIN
       DBMS_OUTPUT.PUT_LINE( ch ) ;
EXCEPTION
       WHEN OTHERS THEN Null ;
END ;
/
```

Error Section

- Errors must be intercepted with the keyword WHEN associated to error code.
- Here, the code OTHERS defines all other errors that are not defined on the previous WHEN clauses.
- DBMS_OUTPUT allows to display the values of the variable.

```
DECLARE
        ch VARCHAR2(15) := 'Hello World' ;
BEGIN
        DBMS_OUTPUT.PUT_LINE( ch ) ;
EXCEPTION
        WHEN OTHERS THEN Null ;
END ;
/
```

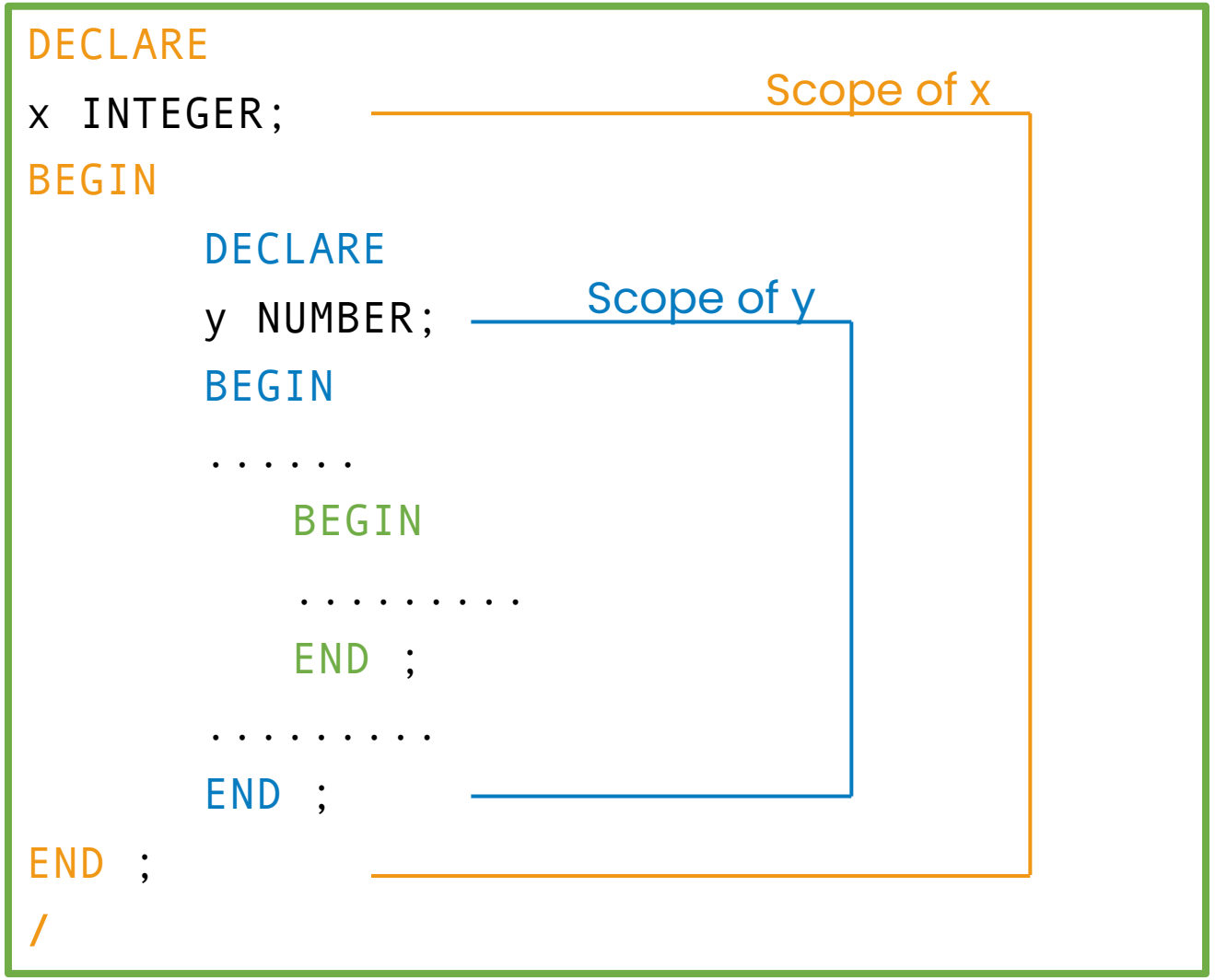
Nested blocks

- The PL / SQL blocks can be nested.

```
DECLARE
.....
BEGIN
        DECLARE
        .....
        BEGIN
        ......
            BEGIN
            .........
            END ;
        .........
        END ;
END ;
/
```

Nested blocks

- Nested blocks and variables:



Script

- The PL / SQL blocks can be placed in text files with .sql extension which will then be loaded into SQL * Plus.
- You need to insert the line: SET SERVEROUTPUT on to use the command DBMS_OUTPUT.PUT_LINE ();.
- The slash (/) At the end of the PL / SQL block forces its execution.

```
SET serveroutput on
DECLARE
       ch VARCHAR2(15) := 'Hello World' ;
BEGIN
       DBMS_OUTPUT.PUT_LINE( ch ) ;
END ;
/
```

03

Declaration and use of variables

03

Declaration of a variable

- The declarative section contains all the declarations of variables to be used locally by the executable section, and their possible initialization.

```
DECLARE
        ch VARCHAR2(15) := 'Hello World' ;
BEGIN
        DBMS_OUTPUT.PUT_LINE( ch ) ;
END ;
/
```

Declaration of a variable

- Syntax
  - Name [CONSTANT] type [NOT NULL] [: = Expression];
- Example

```
DECLARE
        v_dateOfBirth DATE;

        v_dep Number(2) NOT NULL := 10;

        v_city VARCHAR2(13) := 'Paris';

        c_cumul CONSTANT NUMBER := 1000;
```

Declaration of a variable

- Syntax
  - Name [CONSTANT] type [NOT NULL] [: = Expression];
- **Name** : Is the name of the variable. The variable name can not exceed 30 characters
- **CONSTANT** : Indicates that the value can not be changed, in this case **expression** must be indicated.
- **NOT NULL** : Indicates that the variable can not be NULL, in this case **expression** must be indicated.
- **type** : Represents the type of the variable

Declaration of a variable

- The basic types:
  - VARCHAR2 (n): string of max size n
    - NULL and an empty string are considered the same in varchar2, contrary to varchar
  - NUMBER (n, m): Number of size n, m is the size of decimals.
  - DATE: A date.
  - CHAR (n): string with a size= n.
  - BOOLEAN: 0 or 1.
  - INTEGER: integer.

Declaration of a variable

- The attribute %TYPE gives the associated type to:
  - A column table in the database.
  - A previously defined variable.

- Examples:
  - v_name emp.name% TYPE; --emp is the name of a table
  - v_annual_salary NUMBER (7,2);
  - v_monthly_salary v_annual_salary % TYPE: = 2000;

Declaration of a variable

- To declare a global variable, it must be preceded by the reference ' : '.

- Examples:
  - Store the monthly salary in a global variable on SQL * Plus:
    - :g_monthly_salary := v_annual_salary / 12;
  - Use the monthly salary in a PL / SQL block:
    - IF :g_monthly_salary > 1200 THEN ....

Assigning a value

- With the operator := in the implementation section:
  - Identifier := expr;

- Examples
  - Assign a date of birth:
    - v_dateOfBirth := '23 -SEP-2004';
  - Set the name to 'Clement'
    - v_name := 'Clement';

Assigning a value

- Assigning values with SELECT INTO :

```
SELECT      select_list
INTO        var_name[,var_name2…]
FROM        table
WHERE       condition;
```

- **The query must return a single value per column!**

Assigning a value
Example 1

The following example uses a `SELECT INTO` statement to get the name of a customer based on the customer id, which is the primary key of the `customers` table.

```
DECLARE
  v_customer_name customers.name%TYPE;
BEGIN
  SELECT
    name
  INTO
    v_customer_name
  FROM
    customers
  WHERE
    customer_id = 100;

  DBMS_OUTPUT.PUT_LINE( v_customer_name );
END;
```

Because the `customers` table has a row with customer ID 100, the block displayed the customer name.

```
Verizon
```

Assigning a value
Example 2

```
DECLARE
        v_deptnum NUMBER(2);
        v_loc  VARCHAR2(15);
BEGIN
        SELECT deptnum, loc
        INTO   v_deptnum, v_loc
        FROM   dept
        WHERE  name_dep = 'INFO';

        DBMS_OUTPUT.PUT_LINE('v_deptnum'||'v_loc');

END ;
/
```

Assigning a value
Example 3

Returns the sum of the salaries of all employees of a given department.

```
DECLARE
        v_sum_sal emp.salary%TYPE;
        v_deptnum NOT NULL NUMBER := 10;
BEGIN
        SELECT SUM(salary)
        INTO   v_sum_sal
        FROM   emp
        WHERE  deptnum = v_deptnum;

        DBMS_OUTPUT.PUT_LINE('v_sum_sal');

END ;
/
```

Using a variable
Example 1

- Add a new employee information the *emp* table.

```
DECLARE
        v_empnum NOT NULL NUMBER := 105;
BEGIN
        INSERT INTO   emp
        VALUES (v_empnum, 'Clement','Manager',10);
END ;
/
```

Using a variable
Example 2

- Increase the salary of all employees who have a teaching position.

```
DECLARE
        v_augm emp.sal%TYPE := 2000;
BEGIN
        UPDATE emp
        SET    sal := sal + v_augm
        WHERE  job = 'Lecturer';
END ;
/
```

Using a variable
Example 3

- Remove lines belonging to the department 10.

```
DECLARE
        v_deptnum emp.deptnum%TYPE := 10;
BEGIN
        DELETE FROM   emp
        WHERE         deptnum = v_deptnum;
END ;
/
```

04

The Control Structures : Conditions and Loops

04

Control Structures in PL/SQL

- Conditional IF
  - IF …. THEN …. END IF;
  - IF …. THEN …. ELSE …. END IF;
  - IF …. THEN …. ELSIF …. THEN …. END IF;

- Loops
  - LOOP …… END LOOP;
  - FOR ….. LOOP …… END LOOP;
  - WHILE ….. LOOP …… END LOOP;

IF Statement

- Syntax

```
IF condition THEN
        statements;
[ELSIF condition THEN
        statements;]
[ELSE
        statements;]
END IF;
```

- *Example*: If the employee name is 'Clement', assign:
  - the position 'Teacher'
  - the department No. 102 and
  - a 25% commission on his current salary.

```
IF v_name = 'Clement' THEN
        v_pos := 'Teacher';
        v_deptnum := 102;
        v_sal := sal * 0.25;
END IF;
```

IF Statement

- *Example2* : If the employee name is 'Clement', then:
  - assign him the job 'Teacher', the department No. 102 and a 25% commission on his current salary.
  - otherwise display 'non-existent employee'.

```
IF v_name = 'Clement' THEN
        v_pos := 'Teacher';
        v_deptnum := 102;
        v_sal := sal * 0.25;
ELSE
        DBMS_OUTPUT.PUT_LINE('Non-existent employee');

END IF;
```

IF Statement

- If the IF is within a function, we can use RETURN to return a value.

```
IF v_beginning = >100 THEN
        RETURN (2*v_ beginning);
ELSIF v_beginning = >=50 THEN
        RETURN (5*v_ beginning);
ELSE
        RETURN (v_beginning);
END IF;
```

Basic LOOP

- Syntax

```
LOOP
        statement 1;
…
EXIT [WHEN condition];
END LOOP;
```

- *Example*: Insert in the "article" table 10 items numbered from 1 to 10 and with today's date.

```
DECLARE
        v_date DATE;
        v_count NUMBER(2) := 1;
BEGIN
        v_date := SYSDATE;
        LOOP
                INSERT INTO   article
                VALUES (v_count,v_date);
                v_count := v_count +1;
                EXIT WHEN v_count >10;
        END LOOP;
END ;
/
```

FOR LOOP

- Syntax

```
FOR index IN [REVERSE] inf…sup LOOP
        statement 1;
        statement 2;
        …
END LOOP;
```

- *Remarks* :
  - We don’t need to declare the index, it is declared implicitly.
  - The option REVERSE can browse the index backwards.

FOR LOOP

- *Example*: Create Nb articles indexed from 1 to Nb with the system date using the FOR loop.

```
DECLARE
        v_date DATE;
BEGIN
        v_date := SYSDATE;
        FOR i IN 1..&Nb LOOP
                INSERT INTO   article
                VALUES (i,v_date);
        END LOOP;
END ;
/
```

With &Nb, the system requires a value from the user at the beginning of the loop.

WHILE LOOP

- Syntax

```
WHILE condition LOOP
        statement 1;
        statement 2;
        …
END LOOP;
```

- *Note :*
  - The condition is evaluated before each iteration.

WHILE LOOP

- *Example*: Insert the "Item" table 10 items numbered from 1 to 10 and with today's date.

```
DECLARE
        v_date DATE;
        v_count NUMBER(2) := 1;
BEGIN
        v_date := SYSDATE;
        WHILE  v_count <10 LOOP
                INSERT INTO   article
                VALUES (v_count,v_date);
                v_count := v_count +1;
        END LOOP;
END ;
/
```

Nested Loops and Labels

- We can Nest loops to multiple levels.
- Use labels to distinguish between blocks and loops.
- Leave the outer loop with an EXIT referencing the label.
- The label is written as :

  << label_name >>

```
BEGIN
  << ext_loop >>
  LOOP
  v_count := v_count + 1;
  EXIT WHEN v_count > 10;
    << int_loop >>
      LOOP
      ...
      EXIT ext_loop WHEN total = 1;
      EXIT int_loop WHEN int_done = 1;
      ...
      END LOOP int_loop;
  END LOOP ext_loop;
END ;
/
```

05

Error Management

05

Handling exceptions

- Exception handling is a mechanism to handle errors encountered when running.
- This allows the execution to continue if the error is not important enough to terminate the program.
- If an error is encountered, the exception is processed, and the program beyond the current block and the execution process continues.
- Types of exceptions:
  - Predefined Oracle exceptions
  - Non-predefined Oracle exceptions
  - User-defined exceptions

Triggered implicitely

Triggered explicitely

Capturing Exceptions

- Syntax

```
EXCEPTION
        WHEN exception 1 [OR exception 11 …] THEN
        statement 1;
        statement 2;
        …
        [WHEN exception 2 [OR exception 22 …] THEN
        statement 3;
        statement 4;
        …]
        [WHEN OTHERS THEN
        statement 5;
        …]
```

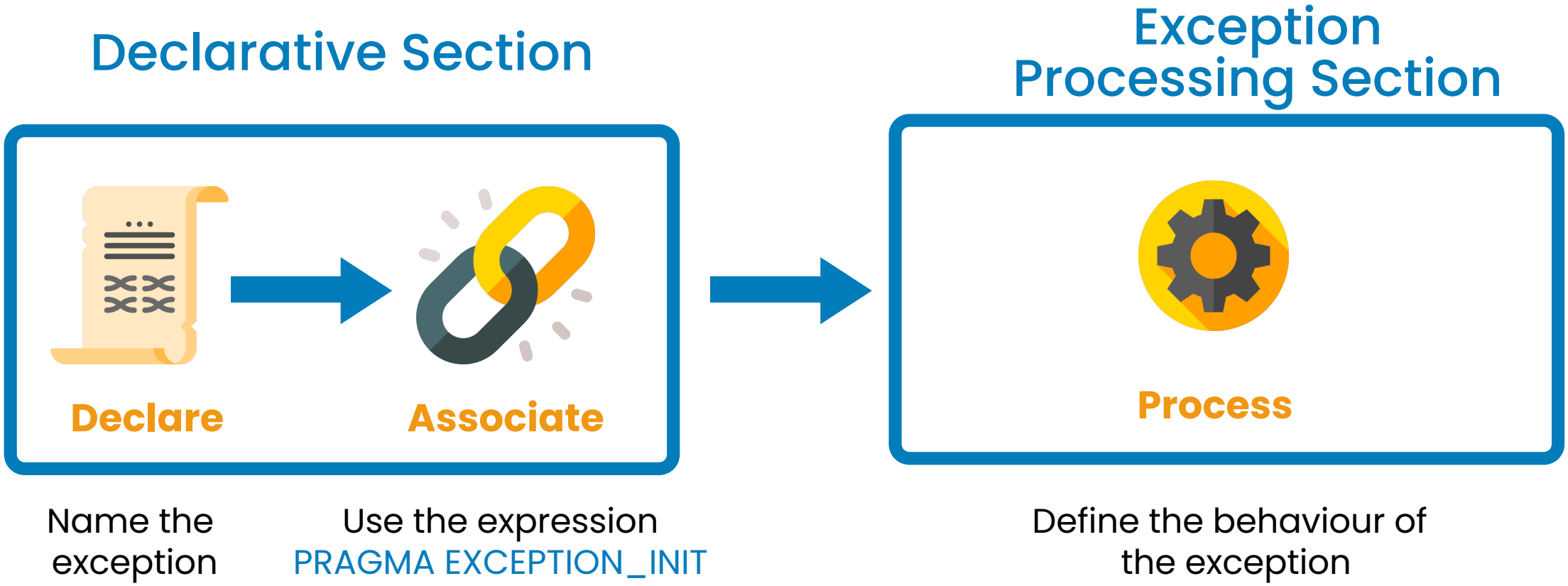
Predefined exceptions

- Refer to the name in the part where exceptions are processed.
- Some predefined exceptions:
  - NO_DATA_FOUND
  - TOO_MANY_ROWS
  - INVALID_CURSOR
  - ZERO_DIVIDE
  - DUP_VAL_ON_INDEX

Predefined exceptions
Example

```
BEGIN
        ...
        COMMIT ;
EXCEPTION
        WHEN NO_DATA_FOUND THEN
                DBMS_OUTPUT.PUT_LINE (TO_CHAR (studnum) || 'Invalid');
        WHEN TOO_MANY_ROWS  THEN
                DBMS_OUTPUT.PUT_LINE ('Invalid Data');
        WHEN OTHERS THEN
                DBMS_OUTPUT.PUT_LINE ('Other errors');
                ROLLBACK ;
END ;
/
```

Non-predefined exceptions



Non-predefined exceptions

- *Example*: Capture the Error No. 2291 (integrity constraint violation).

```
DECLARE
   e_integ_viol EXCEPTION;

   PRAGMA EXCEPTION_INIT (e_integ_viol, -2291);
BEGIN
        ...
EXCEPTION
    WHEN e_integ_viol THEN
      DBMS_OUTPUT.PUT_LINE ('violation of an integrity constraint');
END ;
/
```

User-defined exceptions

Declarative Section

Declare

Name the exception

Execution Section

Activate

Trigger the exception explicitely using the RAISE clause

Exception Processing Section

Process

Define the behaviour of the exception RAISE_APPLICATION_ERROR

User-defined exceptions

- The command RAISE_APPLICATION_ERROR displays a message and an error code for an exception defined by the user.

- Syntax:

```
RAISE_APPLICATION_ERROR(Error_code, message);
```

- The error code must be between -20000 and -20999.
- The error message will be displayed just like for a classic error.
- The PL / SQL code stops immediately and displays the error.

User-defined exceptions
Example

```
DECLARE
        x NUMBER;
        very_small_x  EXCEPTION;

BEGIN

        .............
        IF   x < 5 THEN RAISE very_small_x;
        END IF;
        .............

EXCEPTION
        WHEN very_small_x THEN
         RAISE_APPLICATION_ERROR (-20002, 'the value of x is too small !!');
END ;
/
```

The Capture functions

- SQLCODE
  - Returns the numeric value of the error code.
- SQLERRM
  - Returns the message associated with the error number.

```
DECLARE
        v_error_code NUMBER;
        v_error_message VARCHAR2(255);

BEGIN
        .............
EXCEPTION
        WHEN OTHERS THEN
        v_error_code:= SQLCODE;
        v_error_message := SQLERRM;
        INSERT INTO errors
        VALUES (v_error_code,
                v_error_message);
END ;
/
```

06

Triggers

06

Triggers

- A trigger is a PL / SQL program which runs automatically before or after a IUD operation (Insert, Update, or Delete).

- Unlike procedures, a trigger is raised automatically following an IUD order.

Event-Condition-Action

- A trigger is activated by an event:
  - Insertion, deletion or modification on a table

- If the trigger is activated, a condition is evaluated:
  - Predicate must return true

- If the condition is true, an action is performed:
  - Inserting, deleting, or modifying database

Trigger Components

- When is the trigger activated?
  - BEFORE : The code in the body of the trigger runs before IUD trigger events.
  - AFTER : The code in the body triggers runs after IUD trigger events.
- What IUD operations can cause the execution of the trigger?
  - INSERT
  - UPDATE
  - DELETE
  - The combination of these operations
- The body of the trigger is defined by a PL / SQL block:

```
DECLARE
BEGIN
EXCEPTION
END ;
/
```

Trigger components
Syntax

```
CREATE [OR REPLACE] TRIGGER <trigger_name>
[BEFORE | AFTER] [INSERT [OR] DELETE [OR] UPDATE]
ON <table_name>
[FOR EACH ROW][WHEN <condition>]
DECLARE
BEGIN
EXCEPTION
END ;
/
```

Trigger components
Example

```
CREATE OR REPLACE TRIGGER StartInvoice
AFTER INSERT ON Invoice
FOR EACH ROW
DECLARE
        v_nb_insert NUMBER := 1;
BEGIN
        SELECT Nb_Insert INTO v_nb_insert
        FROM Statistics
        WHERE Table_name = 'Invoice' ;
        UPDATE Statistics
        SET Nb_Insert = v_nb_insert + 1
        WHERE Table_name = 'Invoice' ;
EXCEPTION
        WHEN NO_DATA_FOUND THEN
                INSERT INTO Statistics VALUES ('Invoice',1) ;
END ;
/
```

Handling triggers

- Enable or disable a Trigger:
  - ALTER TRIGGER <Trigger_name> [ENABLE | DISABLE];
- Delete Trigger:
  - DROP TRIGGER <Trigger_name>;
- Identify the triggers on your DB:
  - SELECT <Trigger_name> FROM User_Triggers ;

:Old and :New Attributes

- These two attributes are used to manage the old and new used values.
- Beware of the ":" before Old and New in the execution section!

- *Example 1*: create a trigger that updates the class table after inserting a new student.
  - Student (Id_stu, Name, …, Id_Class)
  - Class (Id_Class, Nb_Stu)

```
CREATE OR REPLACE TRIGGER UpdateNbStudents
AFTER INSERT ON Student
FOR EACH ROW

BEGIN
        UPDATE Class
        SET Nb_Stu = Nb_Stu + 1
        WHERE Id_Class= :New.Id_Class ;
END ;
/
```

:Old and :New Attributes

- *Example 2*: create a trigger that updates the Class table after inserting a new student if she is more than 18 years old.
  - Student (Id_stu, Name, …, Id_Class)
  - Class (Id_Class, Nb_Stu)

```
CREATE OR REPLACE TRIGGER UpdateNbStudents
AFTER INSERT ON Student
FOR EACH ROW
WHEN (New.Age > 18)    -- no ":" in WHEN
BEGIN
        UPDATE Class
        SET Nb_Stu = Nb_Stu + 1
        WHERE Id_Class= :New.Id_Class ;
        -- ":" is mandatory in BEGIN
END ;
/
```

The “Inserting”, “Updating” and “Deleting” Predicates

- Inserting :
  - True: The trigger is enabled after insertion
  - False: Otherwise
- Updating :
  - True: the trigger is enabled due to an update
  - False: Otherwise
- Deleting :
  - True: the trigger is enabled after a deletion
  - False: Otherwise

The “Inserting”, “Updating” and “Deleting” Predicates
Example

```
CREATE OR REPLACE TRIGGER UpdateNbStudents
AFTER INSERT OR DELETE ON Student
FOR EACH ROW
BEGIN
        IF inserting THEN
                UPDATE Class
                SET Nb_Stu = Nb_Stu + 1
                WHERE Id_Class= :New.Id_Class ;
        END IF;
        IF deleting THEN
                UPDATE Class
                SET Nb_Stu = Nb_Stu - 1
                WHERE Id_Class= :Old.Id_Class ;
        END IF;
END ;
/
```

07

Functions and Procedures

07

Sub-Programs

- A subprogramm is a PL / SQL set of statements that has a name.
- There are two types of sub-programs:
  - The procedures
  - The functions
- A procedure is a subprogram that executes PL/SQL code and does not return a result.
- A function is a subprogram which performs a PL/SQL code and returns a result.

Procedures
Syntax

```
DECLARE
        …
        PROCEDURE <proc_name> [(P1,…,Pn)] IS
                [Local declarations]
        BEGIN
                …
        EXCEPTION
                …
        END;
BEGIN
        …
        <proc_name>[(A1,…,An)];
        …
EXCEPTION
        …
END ;
/
```

Syntax of P1…Pn:
<arg_name>[IN | OUT | IN OUT] <Type>
Where:
IN: input parameter
OUT: output parameter
IN OUT: Input/Output parameter
By default, the setting is IN

Procedures
Example

```
DECLARE
        VSAL NUMBER (7,2);
        PROCEDURE NouvSal (pnum IN Emp.Emp_Id%Type, PAug NUMBER) IS
                VSAL NUMBER (7,2);
                BEGIN
                        SELECT Sal INTO VSAL FROM Emp WHERE EMP_ID = pnum;
                        UPDATE Emp SET Sal = VSAL + Paug WHERE EMP_ID = pnum;
                        COMMIT ;
                EXCEPTION
                        WHEN NO_DATA_FOUND THEN
                                DBMS_OUTPUT.PUT_LINE( 'Non-existent employee') ;
                END ;
BEGIN
        NouvSal (7550, 500);
EXCEPTION
        WHEN OTHERS THEN DBMS_ OUTPUT.PUT_LINE(‘ERROR’);
END ;
/
```

Functions
Syntax

```
DECLARE
        [Global declarations]
        FUNCTION <func_name> [(P1,…,Pn)] RETURN Type
        IS
                [Local declarations]
        BEGIN
                …
                RETURN value;
        EXCEPTION
                …
        END;
BEGIN
        …
        var := <func_name>[(A1,…,An)];
        …
EXCEPTION
        …
END ;
/
```

Functions
Syntax

```
DECLARE
    VNomComplet VARCHAR2(40) ;
    VErr NUMBER;
    FUNCTION NomComplet(PNum Emp.Emp _Id%TYPE, PErr OUT NUMBER )
    RETURN VARCHAR2 IS
        VLastName Emp.Last_Name%Type ;
        VFirstName Emp.First_Name%Type ;
        BEGIN
                SELECT Last_Name, First_Name INTO VLastName, VFirstName
                WHERE Emp_Id=PNum ;
                PErr :=0 ;
                RETURN VLastName || ' ' || VFirstName ;
        EXCEPTION
                WHEN NO_DATA_FOUND THEN PErr :=1 ;
                RETURN Null ;
        END ;
BEGIN
    VNomComplet := NomComplet(&Num, VErr) ;
    IF VErr = 0 THEN
        DBMS_ OUTPUT.PUT_LINE('Nom Complet est : ' || VNomComplet);
    ELSE DBMS_ OUTPUT.PUT_LINE('Employé inexistant') ;
    END IF ;
END ;
/
```

Stored procedures and stored functions

- Are PL / SQL blocks that have names.
- Are used to store the compiled PL / SQL block into the database (CREATE).
- Can be reused without being recompiled (EXECUTE or EXEC).
- Can be called from any PL / SQL block.
- May be grouped together in a package.

Stored Procedures
Syntax

```
CREATE OR REPLACE PROCEDURE <proc_name> [(P1,…,Pn)]
IS
[Local declarations]
BEGIN
        …
EXCEPTION
        …
END;
/
```

Stored Procedures
Example

```
CREATE OR REPLACE PROCEDURE AddProd (PrefPro
Prod.RefPro%TYPE,…, PPriUni Prod.PriUni%TYPE, PErr OUT
Number)
IS
BEGIN
      INSERT INTO Prod VALUES(PrefPro,…,PPriUni) ;
      COMMIT ;
      PErr := 0 ;
EXCEPTION
      WHEN OTHERS THEN
            PErr := 1;
END;
/
```

Calling a Stored Procedure
Syntax

- The stored procedure is called by applications:
  - Using its name in a PL / SQL block
  - Using EXECUTE in SQL * Plus.
- In a PL / SQL block:

  BEGIN

  <Procedure-name> [(<P1> … <Pn>)];

  END ;

- Under SQL * PLUS:
  - EXECUTE <Procedure-name> [(<P1> … <Pn>)]

Calling a Stored Procedure

Example 1

```
ACCEPT VRefPro     -- request a value from the user
ACCEPT VUniPri
.......
DECLARE
      VErr NUMBER;
BEGIN
      AddProd(&VRefPro, ..., &VUniPri, VErr);
      IF  VErr = 0 THEN
            DBMS_ OUTPUT.PUT_LINE (
                  'Operation Performed ');
      ELSE DBMS_ OUTPUT.PUT_LINE ( 'error');
      END IF ;
END ;
/
```

Stored Procedure

Example

We're going to develop a procedure named `adjust_salary()` in HR sample database provided by Oracle. We'll update the salary information of employees in the `employees` table by using SQL UPDATE statement.

The following is the source code of the `adjust_salary()` procedure :

```
1  CREATE OR REPLACE PROCEDURE adjust_salary(
2      in_employee_id IN EMPLOYEES.EMPLOYEE_ID%TYPE,
3      in_percent IN NUMBER
4  ) IS
5  BEGIN
6     -- update employee's salary
7     UPDATE employees
8     SET salary = salary + salary * in_percent / 100
9     WHERE employee_id = in_employee_id;
10 END;
```

Now, we can call `adjust_salary()` procedure as the following statements:

```
1  -- before adjustment
2  SELECT salary FROM employees WHERE employee_id = 200;
3  -- call procedure
4  exec adjust_salary(200,5);
5  -- after adjustment
6  SELECT salary FROM employees WHERE employee_id = 200;
```

Stored Functions
Syntax

```
CREATE [OR REPLACE] FUNCTION <func_name> [(P1,…,Pn)]
RETURN Type IS
[Local declarations]
BEGIN
        …
        RETURN(Value)
EXCEPTION
        …
END;
/
```

Stored Functions

Example

```
CREATE OR REPLACE FUNCTION NbEmp (PNumDep Emp.Dept_Id%
Type, PErr OUT Number)
RETURN Number IS
      VNB Number(4);
BEGIN
      SELECT Count (*) INTO VNB FROM Emp
            WHERE Dept_id = PNumDep;
      PErr := 0;
      RETURN VNB;
EXCEPTION
      WHEN OTHERS THEN
            PErr := 1;
            RETURN Null;
END;
/
```

Calling a Stored Function
Syntax

- The stored function is called by applications:
  - In an expression in a PL / SQL block.
  - Using EXECUTE in SQL * Plus.
- In a PL / SQL block:

  BEGIN

  <Var> := <Function-name> [(<P1> ... <Pn>)];

  END ;

- Under SQL * PLUS:
  - EXECUTE <Var> := <Procedure-name> [(<P1> ... <Pn>)]

Calling a Stored Function

Example – In PL/SQL

```
ACCEPT Vdep          -- request a value from the user
DECLARE
      VErr NUMBER;
      VNB NUMBER(4);
BEGIN
      VNB := NbEmp(&VDep, VErr);
      IF  VErr = 0 THEN
            DBMS_ OUTPUT.PUT_LINE (
               'The number of employees is '||VNB);
      ELSE DBMS_ OUTPUT.PUT_LINE ( 'error');
      END IF ;
END ;
/
```

Calling a Stored Function

Example – In SQL*Plus

```
SQL> VARIABLE VNB

SQL> EXECUTE :VNB := NbEmp(& VDep, VErr);

PL / SQL procedure successfully completed.

SQL> PRINT VNB


VNB
----------
300
```

Stored Function

Example

We are going to create a function named `try_parse` that parses a string and returns a number if the input string is a number or `NULL` if it cannot be converted to a number.

```
CREATE OR REPLACE FUNCTION try_parse(
    iv_number IN VARCHAR2)
  RETURN NUMBER IS
BEGIN
   RETURN to_number(iv_number);
   EXCEPTION
     WHEN others THEN
        RETURN NULL;
END;
```

Let's create an anonymous block to use the `try_parse()` function.

```
SET SERVEROUTPUT ON SIZE 1000000;
DECLARE
  n_x number;
  n_y number;
   n_z number;
BEGIN
   n_x := try_parse('574');
   n_y := try_parse('12.21');
   n_z := try_parse('abcd');

   DBMS_OUTPUT.PUT_LINE(n_x);
   DBMS_OUTPUT.PUT_LINE(n_y);
   DBMS_OUTPUT.PUT_LINE(n_z);
END;
/
```

Procedures and functions: Useful Commands

- Deleting stored procedures and functions
  - DROP PROCEDURE procname
  - DROP FUNCTION funcname

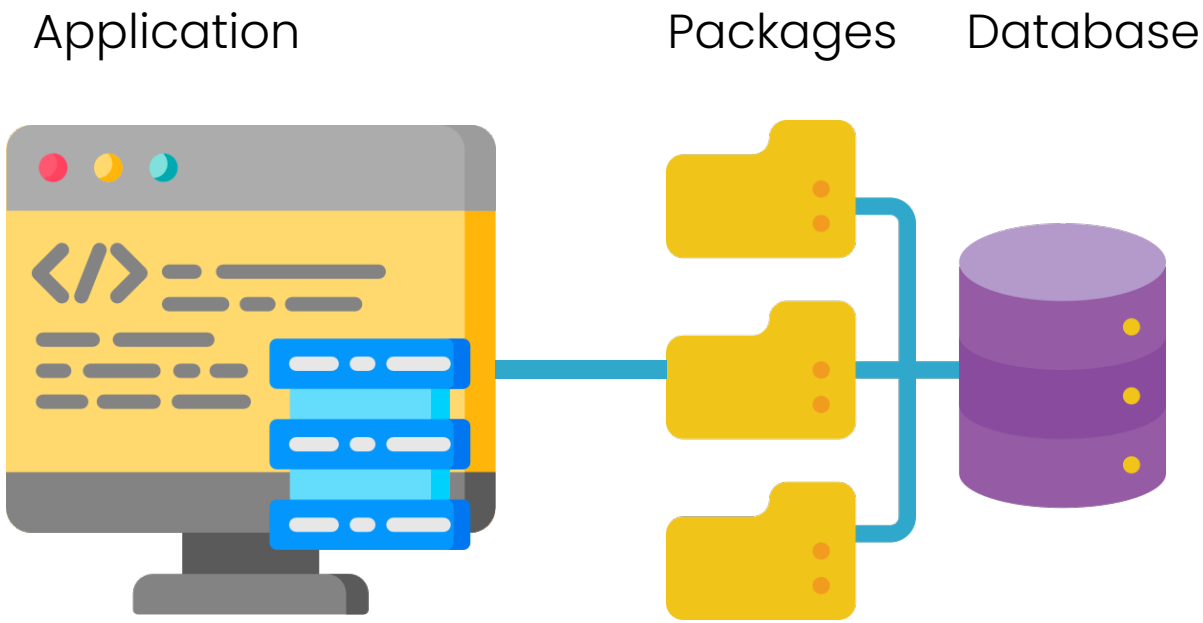
- Describing a procdure or a function
  - DESC procname
  - DESC funcname

Packages

- A PL / SQL object that stores other types of objects: procedures, functions, cursors, variables, ...
- Consists of two parts:
  - Specification (declaration)
  - Body (implementation)
- Can not be called or set or nested
- Allows Oracle to read multiple objects at once in memory

Packages

- A PL/SQL package has two parts: package specification and package body
  - A package specification is the public interface of your applications. The public means the stored function, procedures, types, etc. are accessible from other applicatons
  - A package body contains the code that implements the package specification.



Creating a Package Specification
Syntax

```
CREATE [OR REPLACE] PACKAGE <package_name>
IS
      [Declaration of variables and types]
      [Declarataion of cursors]
      [Declaration of procedures and functions]
      [Declaration of exceptions]
END [<package_name>];
/
```

Creating a Package Specification
Example

```
CREATE OR REPLACE PACKAGE PackProd
IS
      Cursor Cprod IS SELECT RefPro, DesPro FROM PRODUCT;
      Procedure AddProd(PrefPro Prod.RefPro%Type,…, Perr Out Number);
      Procedure ModifProd(PrefPro Prod.RefPro%Type,…, Perr Out Number);
      Procedure SuppProd(PrefPro Prod.RefPro%Type,…, Perr Out Number);
      Procedure DispProd;
END PackProd ;
/
```

Creating a Package Body
Syntax

```
CREATE [OR REPLACE] PACKAGE BODY <package_name>
IS
      [Procedures and Functions Implementations]
END [<package_name>];
/
```

Creating a Package Body

Example

```
CREATE OR REPLACE PACKAGE BODY PackProd
IS
      Procedure AddProd(PrefPro Prod.RefPro%Type,…, Perr Out Number)
      IS
      BEGIN
            INSERT INTO Prod VALUES (PrefPro,…,PUnitPri);
            COMMIT;
            PErr := 0;
      EXCEPTION
            WHEN Dup_Val_On_Index THEN PErr := 1;
            WHEN OTHERS THEN PErr := 1;
      End;
END PackProd ;
/
```

Using a Package
Syntax

```
<package_name>.<procname>[(params>];

Var := <package_name>.<funcname>[(params>];
```

Using a Package
Example

```
ACCEPT Vref Prompt '…';
ACCEPT VPri Prompt '…';
DECLARE
      VErr NUMBER;
BEGIN
      PackProd.ModifProd(&VRef, ..., &VPri, VErr);
      IF  VErr = 0 THEN
            DBMS_ OUTPUT.PUT_LINE (
                  'Operation Performed ');
      ELSE DBMS_ OUTPUT.PUT_LINE ( 'error');
      END IF ;
END ;
/
```

08

Cursors

08

Cursors

- A cursor is a pointer to a private SQL memory area allocated for the processing of a SQL statement. The cursor treats records (table rows) one by one.

- A cursor is a SELECT statement that is defined within the declaration section of your PL/SQL code.

Declaration of Cursors

```
CURSOR cursor_name IS SELECT_Statement;
```

- Do not include the INTO clause in the cursor declaration.
- If the rows processing must be done in a specific order, the ORDER BY clause is used in the query.

```
DECLARE
      CURSOR C1 IS
            SELECT RefArt, NameArt, QttArt
            FROM Article
            WHERE QttArt < 500;
```

Opening the Cursors

```
OPEN cursor_name
```

- Open the cursor to run the query and identify the active set.
- Use attributes of the cursor to test the result of FETCH.

Searching Data

```
FETCH cursor_name
INTO var1 [,var2,...]
```

- Find information and put them in variables.
- The lines are treated with the same order as the table, for each FETCH, the next row is treated.
- *Example*

```
FETCH C1
INTO V_RefArt, V_NameArt, V_QttArt;
```

Cursor Closure

```
CLOSE cursor_name;
```

- Close the cursor after the end of lines processing.
- Reopen the cursor, if necessary.
- You can not search for information in a cursor if it is closed.

Attributes of explicit cursors

- To get the cursor status information:

Attribute	Type	Description
%ISOPEN	Boolean	Is true if the cursor is open
%NOTFOUND	Boolean	Is true if the latest fetch returns no rows
%FOUND	Boolean	Is true if the latest fetch returns at least a row
%ROWCOUNT	Number	Returns the number of rows processed so far

Cursors
Complete Example

```
CREATE OR REPLACE FUNCTION FindCourse (name_in IN VARCHAR2)
RETURN Number IS
        cnumber Number(4);
        CURSOR C1 IS
                SELECT course_number
                FROM courses
                WHERE course_name = name_in;
BEGIN
        OPEN C1;
        FETCH C1 INTO cnumber;
        IF C1%NOTFOUND THEN
                cnumber := 9999;
        END IF;
        CLOSE C1;
        RETURN cnumber;
END;
/
```

Records

- A record is a variable that contains an entire row of a table.

```
DECLARE
CURSOR Stud_Curs IS
       SELECT       studno, name, age, addr
       FROM  stud WHERE age < 26;
Stud_Record Stud_Curs%ROWTYPE;      -- definition of the record

BEGIN
   OPEN Stud_Curs;
    ……..
   FETCH Stud_Curs INTO Stud_Record;       -- filling the record

   IF Stud_Record.age <18 THEN       -- using of the record
   ……
END;
```

FOR-LOOP for Cursor

Syntax

```
FOR name_record IN cursor_name
LOOP
      [information processing]
END LOOP;
```

- A shortcut to process explicit cursors.
- OPEN, FETCH and CLOSE are implicit.
- Do not declare the record, it is declared implicitly.

FOR-LOOP for Cursor

Example

```
DECLARE
CURSOR Cur_Stud IS
      SELECT * FROM Stud ;

BEGIN
      FOR Rec_Stud IN Cur_Stud  LOOP
            DBMS_OUTPUT.PUT_LINE(Rec_Stud.name ||' '||
                                 Rec_Stud.addr);

      END LOOP;
END;

/
```

Attributes of implicit cursors

- These are cursors implicitely created when running a SQL command
- Can be used to test the results of SQL statements

Attribute	Type	Description
SQL%ROWCOUNT	INTEGER	Number of rows affected by the most recent SQL statement
SQL%FOUND	Boolean	True if the most recent SQL statement affects one or more lines
SQL%NOTFOUND	Boolean	True if the most recent SQL statement does not affect any line
SQL%ISOPEN	Boolean	Always false, because PL/SQL closes implicit cursors immediately after their execution.

Attributes of implicit cursors
Example

- Remove from the ITEM table the lines corresponding to the lot 605.
- Display the number of deleted rows

```
DECLARE
        v_category NUMBER := 605;
BEGIN
        DELETE FROM Item WHERE category = v_category;
        DBMS_OUTPUT.PUT_LINE(SQL%ROWCOUNT || ‘ Lines Deleted’);

END;

/
```