PL/SQL

Structure of PL/SQL

- PL/SQL is Block Structured

  A block is the basic unit from which all PL/SQL programs are built. A block can be named (functions and procedures) or anonymous
- Sections of block

  1- Header Section

  2- Declaration Section

  3- Executable Section

  4- Exception Section

Structure of PL/SQL

HEADER

Type and Name of block

DECLARE

Variables; Constants; Cursors;

BEGIN

PL/SQL and SQL Statements

EXCEPTION

Exception handlers

END;

Structure of PL/SQL

```
DECLARE
        a number;
        text1 varchar2(20);
       text2 varchar2(20) := “HI”;
BEGIN
      --------- --------- ---------
END;
```

Important Data Types in PL/SQL include NUMBER, INTEGER, CHAR, VARCHAR2, DATE etc
to_date(‘02-05-2007','dd-mm-yyyy') { Converts String to Date}

Structure of PL/SQL

- Data Types for specific columns

Variable_name Table_name.Column_name%type;

This syntax defines a variable of the type of the referenced column on the referenced table

PL/SQL Control Structure

- PL/SQL has a number of control structures which includes:
- Conditional controls
- Iterative or loop controls.
- Exception or error controls

- It is these controls, used singly or together, that allow the PL/SQL developer to direct the flow of execution through the program.

Conditional logic –IF statement

```
IF condition1 THEN
        statements
[ELSIF condition2 THEN]
        statements
[ELSE
        last_statements]
END IF;
```

Examples

```
IF hourly_wage < 10 THEN
        hourly_wage := hourly_wage * 1.5;
ELSE
        hourly_wage := hourly_wage * 1.1;
END IF;

IF salary BETWEEN 10000 AND 40000 THEN
        bonus := 1500;
ELSIF salary > 40000 AND salary <= 100000
      THEN bonus := 1000;
ELSE bonus := 0;
END IF;
```

Comments

- The end of the IF statement is “END IF;” with a space in between.
- The “otherwise if” is ELSIF not ELSEIF
- You can put parenthesis around boolean expression after the IF and ELSIF but you don’t have to.
- You don’t need to put {, } or BEGIN, END to surround several statements between IF and ELSIF/ELSE, or between ELSIF/ELSE and END IF;

Conditional logic –Simple CASE statement

```
CASE selector
    WHEN expression_1 THEN statements
    [WHEN expression_2 THEN statements]
    [ELSE statements]
END CASE;
```

- *selector* can be an expression of any datatype, and it provides the value we are comparing.
- *expression_n* is the expression to test for equality with the selector.
- If no WHEN matches the selector value, then the ELSE clause is executed.
- If there is no ELSE clause PL/SLQ will implicitly supply:

  ELSE RAISE CASE_NOT_FOUND;

  which will terminate the program with an error (if the program ends up in the ELSE clause).

```
CREATE OR REPLACE PROCEDURE
    PrintQualification(grade IN CHAR) AS
BEGIN
CASE grade
  WHEN 'A' THEN
    dbms_output.put_line('Excellent');
  WHEN 'B' THEN
    dbms_output.put_line('Very Good');
  WHEN 'C' THEN
    dbms_output.put_line('Good');
  WHEN 'D' THEN
          dbms_output.put_line('Fair');
  WHEN 'F' THEN
          dbms_output.put_line('Poor');
  ELSE dbms_output.put_line('No such
    grade');
END CASE;
END;
/

BEGIN
    PrintQualification('B');
END;
/
```

Iterative Control: LOOP and EXIT Statements

- Three forms of LOOP statements:
  - LOOP,
  - WHILE-LOOP, and
  - FOR-LOOP.
- **LOOP**: The simplest form of LOOP statement is the basic (or infinite) loop:

```
LOOP
      statements
END LOOP;
```

- You can use an EXIT statement to complete the loop.
- Two forms of EXIT statements:
  - EXIT and
  - EXIT WHEN.

E.g. We wish to categorize salaries according to their number of digits.

```
CREATE OR REPLACE FUNCTION
     SalDigits(salary INT) RETURN INT
AS
  digits INT := 1;
  temp INT := salary;
BEGIN
  LOOP
    temp := temp / 10;

    IF temp = 0 THEN
       EXIT;
    END IF;
    --Or we can do: EXIT WHEN temp = 0
    digits := digits + 1;
  END LOOP;
  RETURN digits;
END;
/

BEGIN
dbms_output.put_line(SalDigits(150000));
END;
/
```

WHILE-LOOP

```
WHILE condition
LOOP
   statements
END LOOP;
```

E.g. We wish to categorize salaries according to their number of digits.

```
CREATE OR REPLACE FUNCTION
     SalDigits(salary INT) RETURN INT
AS
  digits INT := 1;
  temp INT := salary;
BEGIN

 WHILE temp > 0
  LOOP
     digits := digits + 1;
     temp := temp / 10;
  END LOOP;

  RETURN digits;
END;
/

BEGIN
dbms_output.put_line(SalDigits(150000));
END;
/
```

FOR Loops

- FOR loops iterate over a specified range of integers.

```
FOR counter IN [REVERSE] low .. high
LOOP
      sequence_of_statements
END LOOP;
```

```
for i in 1..1000 loop
      INSERT INTO a
      VALUES(i,i*2);
end loop;
```

Comments

The range is evaluated when the FOR loop is first entered and is never re-evaluated.

By default, iteration proceeds upward from the lower bound to the higher bound. However, if you use the keyword **REVERSE**, iteration proceeds downward from the higher bound to the lower bound.

Nevertheless, you write the range bounds in ascending (not descending) order.

Inside a FOR loop, the loop counter can be referenced like a constant but cannot be assigned values.

Cursors

Fundamental challenge: SQL is a **set oriented language** while procedure oriented languages like PL/SQL **are record (or tuple oriented).**

Solution: use cursors**. Example:**

```
CREATE OR REPLACE PROCEDURE favorite_play AS
    favorite_play_title VARCHAR(200);
    publication_date DATE;
    CURSOR bcur IS
          SELECT title, date_published
          FROM books
          WHERE UPPER(author)
              LIKE 'SHAKESPEARE%';
BEGIN
  OPEN bcur;
  LOOP
    FETCH bcur INTO favorite_play_title,
                    publication_date;
    EXIT WHEN bcur%NOTFOUND;

    /*Do something useful with
      favorite_play_title and publication_date */
  END LOOP;
  CLOSE bcur;
END;
/
```

- When we open a cursor, behind the scene, Oracle reads (parses) the statement, and associates with it those rows in the table that satisfy the query.
- After we open a cursor we fetch row by row. We need to make sure we list as many variables as there are columns in the tuples returned from the SQL statement.
  - The FETCH in the example retrieves one row.
  - A second FETCH would overwrite the values written by the first FETCH.
  - You cannot re-fetch a tuple already fetched. You should close and open again the cursor.

Cursor attributes

- General form is
  - cursor_name%ATTRIBUTE_NAME
  - This is similar to the accessing some attribute of some record with the dot notation. Here instead of a dot we use the % sign.
- cursor_name%FOUND is BOOLEAN and it's TRUE if the most recent fetch found a row to return; otherwise FALSE.
- cursor_name%NOTFOUND is BOOLEAN and it's the logical inverse of %FOUND.
- cursor_name%ROWCOUNT is NUMBER and it contains the number of rows fetched so far.

```
CREATE OR REPLACE PROCEDURE
    favorite_play AS

    favorite_play_title VARCHAR(200);
    publication_date DATE;
  CURSOR bcur IS
     SELECT title, date_published
     FROM books
     WHERE UPPER(author) LIKE
    'SHAKESPEARE%';
BEGIN
  OPEN bcur;
  LOOP
    FETCH bcur INTO favorite_play_title,
                    publication_date;
    EXIT WHEN bcur%NOTFOUND;

    /*Do something useful with
       favorite_play_title and
    publication_date */
  END LOOP;
  CLOSE bcur;
END;
/
```

Shortcut – Anchored declarations

variable_name
table_name.column_name%TYPE;

```
CREATE OR REPLACE PROCEDURE
    favorite_play AS

    favorite_play_title books.title%TYPE;
    publication_date DATE;
  CURSOR bcur IS
     SELECT title, date_published
     FROM books
     WHERE UPPER(author) LIKE
    'SHAKESPEARE%';
BEGIN
  OPEN bcur;
  LOOP
    FETCH bcur INTO favorite_play_title,
                    publication_date;
    EXIT WHEN bcur%NOTFOUND;

    /*Do something useful with
       favorite_play_title and
    publication_date */
  END LOOP;
  CLOSE bcur;
END;
/
```

Stored procedures

Syntax to create a stored procedure is as follows:

```
CREATE [OR REPLACE] PROCEDURE procedure_name
    [(parameter1 MODE DATATYPE [DEFAULT expression],
    parameter2 MODE DATATYPE [DEFAULT expression],
    …)]
AS
[variable1 DATATYPE;
variable2 DATATYPE;
…]
BEGIN
      statements
END;
/
```

- MODE can be IN for read-only parameters, OUT for write-only parameters, or IN OUT for both read and write parameters.
- The DATATYPE can be any of the types we already have mentioned but without the dimensions, e.g.
  - VARCHAR2, NUMBER,…, but not
  - VARCHAR2(20), NUMBER(10,3)…

Procedures

```
CREATE OR REPLACE Procedure HelloWorld IS
    msg VARCHAR(50); --a local variable
BEGIN
     msg := 'Hello world from a Procedure!';
     DBMS_OUTPUT.PUT_LINE(msg);
END;
/
```

To call it we need to use an anonymous block.

```
BEGIN
    HelloWorld;
END;
/
```

Variables

- Syntax for declaring variables:

variable_name DATATYPE

[CONSTANT]

[:= | DEFAULT initial_value]

- If keyword CONSTANT is present, the initial value of the variable can't be changed.
- := or DEFAULT are synonyms in for assigning an initial value to the variable.
- E.g.

name VARCHAR2 := 'Oracle';

name VARCHAR2 DEFAULT 'Oracle';

cur_date DATE := SYSDATE;

- Variables before being used should be declared in the declaration section of a block (not inside the block).
- PL/SQL data types are a superset of the Oracle data types, i.e. what most of the cases we use is:
  - VARCHAR2(n)
  - INT
  - NUMBER[(n,m)]
  - DATE
- There are also, other types that are only in PL/SQL:
  - PLS_INTEGER (smaller int)
  - BOOLEAN (TRUE, FALSE, or NULL)

Operators (Superset of those for SQL)

- := Assignment
- + Addition
- - Subtraction
- / Division
- * Multiplication
- ** Power
- AND, OR, NOT Logical operators
- = Equality
- !=, <>, ~=, ^= Inequality (four variants)
- <, > Less than, Greater than
- <=, >=
- IN membership in a set
- BETWEEN Range test
- IS NULL, IS NOT NULL
- LIKE (as in SQL for strings)
- || Concatenation of strings

Examples

```
square := x**2;

square_root := x**0.5;

order_overdue BOOLEAN :=
    ship_date > '28-Feb-2008' OR
    priority_level(company_id) = 'High';

full_name = 'Chris' || 'Smith';

IF number_of_pages IS NULL
THEN
     DBMS_OUTPUT.PUT_LINE('Unknown');
END;
```

Stored Functions

Syntax to create a stored function:

CREATE [OR REPLACE] FUNCTION *function_name*

[(*parameter1 MODE DATATYPE* [DEFAULT *expression*],

parameter2 MODE DATATYPE [DEFAULT *expression*],

…)] **RETURN *DATATYPE***

AS

[*variable1 DATATYPE*;

variable2 DATATYPE; …]

BEGIN

statements

RETURN *expression;*

END;

/

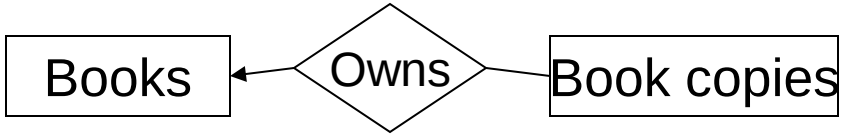
- If you omit it mode it will implicitly be IN.
- In the header, the RETURN DATATYPE is part of the function declaration and it is required. It tells the compiler what datatype to expect when you invoke the function.
- RETURN inside the executable section is also required.
- If you miss the the RETURN clause in the declaration, the program won't compile.
- On the other hand, if you miss the RETURN inside the body of the function, the program will execute but at runtime Oracle will give the error *ORA-06503: PL/SQL: Function returned without a value*.

Dropping stored programs

```
DROP PROCEDURE add_book;
```

An Example

- Let's build a system that will assist in the cataloging and searching of library books.
- For now, we'd like to address two requirements:
  1. Allow the creation of catalog entries for each newly acquired book.
  2. Provide means of counting how many copies of a particular book the library owns.
- A simple E/R schema would be:




```
CREATE TABLE books (
        isbn VARCHAR2(13) PRIMARY KEY,
        title VARCHAR2(200),
        summary VARCHAR2(2000),
        author VARCHAR2(200),
        date_published DATE,
        page_count NUMBER
);
```

```
CREATE TABLE book_copies (
        barcode_id VARCHAR2(100) PRIMARY
      KEY,
        isbn VARCHAR2(13) REFERENCES
                                books(isbn)
);
```

Implementing a **stored** procedure to add a book

```
CREATE OR REPLACE PROCEDURE add_book (
        isbn_in IN VARCHAR2,
        barcode_id_in IN VARCHAR2,
        title_in IN VARCHAR2,
        author_in IN VARCHAR2,
        page_count_in IN NUMBER,
        summary_in IN VARCHAR2 DEFAULT NULL,
        date_published_in IN DATE DEFAULT NULL
) AS
BEGIN
        /*Check for reasonable inputs*/
        IF isbn_in IS NULL THEN
                RAISE VALUE_ERROR;
        END IF;

        INSERT INTO books (isbn, title, summary, author, date_published, page_count)
        VALUES (isbn_in, title_in, summary_in, author_in, date_published_in, page_count_in);

        /*if barcode is supplied, put a record in the book_copies table*/
        if NOT(barcode_id_in IS NULL) then
                INSERT INTO book_copies (isbn, barcode_id)
                VALUES(isbn_in, barcode_id_in);
        end if;
END add_book;
/
```

Using the procedure to add a book

```
BEGIN
        add_book(
                '1-56592-335-9',
                '100000002',
                'Oracle PL/SQL Programming',
                'Feuerstein, Steven, with Bill Pribyl',
                987,
                'Reference for PL/SQL developers, ' ||
                'including examples and best practice recommendations.',
                TO_DATE('01-Sep-1997', 'DD-MON-YYYY')
                );
END;
/
```

Adding a book…

- **Parameter names**: It's good if you follow the convention to end the IN parameters with a suffix _in. (similarly _out for the OUT parameters, or _inout for the INOUT parameters).
- Such naming is not compulsory but helps in avoiding conflicts with the columns names. E.g.
- If we didn't put the _*in* suffix, then it is hard to read code like this:

```
INSERT INTO books (isbn, title, summary, author, date_published, page_count)
VALUES (isbn, title, summary, author, date_published, page_count);
```

- Are they column names or are they PL/SQL variables?
  - In this particular example it turns out that PL/SQL is able to interpret isbn…page_count of the first line as table columns, while in second as PL/SQL variables.
- But, what about this:

```
UPDATE Books
SET summary = summary
WHERE isbn = isbn;
```

- This won't work!

Creating a Function

```
CREATE [OR REPLACE] FUNCTION <function_name>
[(parameter_name [IN | OUT | IN OUT] type [, ...])]
RETURN return_datatype
{IS | AS}
BEGIN
   < function_body >
END <function_name>;
```

Creating a Function

```
CREATE OR REPLACE FUNCTION totalCustomers
RETURN number IS
   total number(2) := 0;
BEGIN
   SELECT count(*) into total FROM customers;
   RETURN total;
END;
/
```

Calling a Function

```
DECLARE
   c number(2);
BEGIN
   c := totalCustomers();
   dbms_output.put_line('Total no. of Customers: ' || c);
END;
/
```

```
DECLARE
   a number;
   b number;
   c number;
FUNCTION findMax(x IN number, y IN number)
RETURN number
IS
    z number;
BEGIN
   IF x > y THEN
      z:= x;
   ELSE
      Z:= y;
   END IF;
   RETURN z;
END;
BEGIN
   a:= 23;
   b:= 45;
   c := findMax(a, b);
   dbms_output.put_line(' Maximum of (23,45): ' || c);
END;
/
```

Retrieving a book count with a function

```
CREATE OR REPLACE FUNCTION book_copy_qty (isbn_in VARCHAR2) return NUMBER
AS
  number_of_copies NUMBER := 0;
    CURSOR bc_cur IS
          SELECT count(*) FROM book_copies WHERE isbn = isbn_in;

BEGIN

  IF isbn_in IS NOT NULL THEN
        OPEN bc_cur;

        FETCH bc_cur INTO number_of_copies;

        CLOSE bc_cur;
  END IF;

  RETURN number_of_copies;
END;
/
```

We can test this function as:

```
set serveroutput on

DECLARE
  how_many INT;

BEGIN
        dbms_output.put_line(
'Number of copies of 1-56592-335-9: ' ||
      book_copy_qty('1-56592-335-9'));
END;
/
```

PL/SQL – Triggers

- Triggers are stored programs, which are automatically executed or fired when some events occur. Triggers are, in fact, written to be executed in response to any of the following events:
  - A database manipulation (DML) statement (DELETE, INSERT, or UPDATE).
  - A database definition (DDL) statement (CREATE, ALTER, or DROP).
  - A database operation (SERVERERROR, LOGON, LOGOFF, STARTUP, or SHUTDOWN).
- Triggers could be defined on the table, view, schema, or database with which the event is associated.

Benefits of Triggers

- Triggers can be written for the following purposes:
- Generating some derived column values automatically
- Enforcing referential integrity
- Event logging and storing information on table access
- Auditing
- Synchronous replication of tables
- Imposing security authorizations
- Preventing invalid transactions

Creating Triggers

- The syntax for creating a trigger is:

```
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)
DECLARE
  Declaration-statements
BEGIN
  Executable-statements
EXCEPTION
  Exception-handling-statements
END;
```

Trigger Example:

- The following program creates a row level trigger for the customers table that would fire for INSERT or UPDATE or DELETE operations performed on the CUSTOMERS table. This trigger will display the salary difference between the old values and new values:

```
CREATE OR REPLACE TRIGGER display_salary_changes
BEFORE DELETE OR INSERT OR UPDATE ON customers
FOR EACH ROW
WHEN (NEW.ID > 0)
DECLARE
  sal_diff number;
BEGIN
  sal_diff := :NEW.salary  - :OLD.salary;
  dbms_output.put_line('Old salary: ' || :OLD.salary);
  dbms_output.put_line('New salary: ' || :NEW.salary);
  dbms_output.put_line('Salary difference: ' || sal_diff);
END;
/
```

- When the above code is executed at SQL prompt, it produces the following result:

```
Trigger created.
```

Triggering a Trigger

- Let us perform some DML operations on the CUSTOMERS table. Here is one INSERT statement, which will create a new record in the table:

  INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)VALUES (7, 'Kriti', 22, 'HP', 7500.00);

  When a record is created in CUSTOMERS table, above create trigger **display_salary_changes** will be fired and it will display the following result:

  Old salary:

  New salary: 7500

  Salary difference:

- Because this is a new record so old salary is not available and above result is coming as null. Now, let us perform one more DML operation on the CUSTOMERS table. Here is one UPDATE statement, which will update an existing record in the table:
- UPDATE customersSET salary = salary + 500WHERE id = 2;
- When a record is updated in CUSTOMERS table, above create trigger **display_salary_changes** will be fired and it will display the following result:

  Old salary: 1500

  New salary: 2000

  Salary difference: 500