Program Outline

- What is PL/SQL?
- PL/SQL Block
- Data Types
- Procedure
- Function
- Cursor
- Record
- Exception
- Trigger
- Transaction

What is PL/SQL?

- Procedural Language extension of SQL.
- Combination of SQL along with the procedural features of programming languages.
- Developed by Oracle Corporation in the early 90's to enhance the capabilities of SQL.

The PL/SQL Engine:

- Oracle uses a PL/SQL engine to processes the PL/SQL statements.
- A PL/SQL code can be stored in the client system (client-side) or in the database (server-side).

Features of PL/SQL

- PL/SQL is tightly integrated with SQL.
- Extensive error checking.
- Numerous data types.
- Variety of programming structures.
- Structured programming through functions and procedures.
- Object-oriented programming.

A Simple PL/SQL Block

- PL/SQL is a block-structured language
- PL/SQL programs are divided and written in logical blocks of code.

- **PL/SQL Block consists of three sections:**
  - The Declaration section (optional).
  - The Execution section (mandatory).
  - The Exception or Error Handling section (optional).

Sections & Description
Declarations This section starts with the keyword **DECLARE**. It is an optional section and defines all variables, cursors, subprograms, and other elements to be used in the program.
Executable Commands This section is enclosed between the keywords **BEGIN** and **END** and it is a mandatory section. It consists of the executable PL/SQL statements of the program. It should have at least one executable line of code, which may be just a NULL command to indicate that nothing should be executed.
Exception Handling This section starts with the keyword **EXCEPTION**. This section is again optional and contains exception(s) that handle errors in the program.

- Every PL/SQL statement ends with a semicolon **(;)**
- PL/SQL blocks can be nested within other PL/SQL blocks using **BEGIN** and **END**.
  - Here is the basic structure of a PL/SQL block:

```
DECLARE
   <declarations section>
BEGIN
  <executable command(s)>
EXCEPTION
  <exception handling>
END;
```

Example

```
DECLARE
message varchar2(20):= 'Hello, World!‘;
BEGIN
dbms_output.put_line(message);
END;
/
```

PL/SQL Identifiers

- Constants
- variables
- exceptions
- procedures
- cursors
- reserved words

Identifier

- a letter optionally followed by
  - more letters
  - numerals
  - dollar signs
  - Underscores
  - number signs
  - not exceed 30 characters
  - identifiers are not case-sensitive

PL/SQL Symbols

- Symbol with a special meaning

Delimiter	Description
+, -, *, /	Addition, subtraction/negation, multiplication, division
%	Attribute indicator
'	Character string delimiter
.	Component selector
(,)	Expression or list delimiter
:	Host variable indicator
,	Item separator
"	Quoted identifier delimiter
=	Relational operator
@	Remote access indicator
;	Statement terminator
:=	Assignment operator
=>	Association operator
||	Concatenation operator
**	Exponentiation operator
<<, >>	Label delimiter (begin and end)
/*, */	Multi-line comment delimiter (begin and end)
--	Single-line comment indicator
..	Range operator
<, >, <=, >=	Relational operators
<>, '=, ~=, ^=	Different versions of NOT EQUAL

The PL/SQL Comments

- single-line
- multi-line

```
DECLARE
    -- variable declaration
     message varchar2(20):= 'Hello, World!';
 BEGIN
    /*
     * PL/SQL executable statement(s)
    */
    dbms_output.put_line(message);
 END;
 /
```

PL/SQL Program Units

- PL/SQL block
- Function
- Package
- Package body
- Procedure
- Trigger
- Type
- Type body

PL/SQL - Data Types

Category	Description
Scalar	Single values with no internal components, such as a NUMBER, DATE, or BOOLEAN.
Large Object (LOB)	Pointers to large objects that are stored separately from other data items, such as text, graphic images, video clips, and sound waveforms.
Composite	Data items that have internal components that can be accessed individually. For example, collections and records.
Reference	Pointers to other data items.

PL/SQL Scalar Data Types and Subtypes

Date Type	Description
Numeric	Numeric values on which arithmetic operations are performed.
Character	Alphanumeric values that represent single characters or strings of characters.
Boolean	Logical values on which logical operations are performed.
Datetime	Dates and times

Numeric Data Types and Subtypes

Data Type	Description
PLS_INTEGER	Signed integer in range -2,147,483,648 through 2,147,483,647, represented in 32 bits
BINARY_INTEGER	Signed integer in range -2,147,483,648 through 2,147,483,647, represented in 32 bits
BINARY_FLOAT	Single-precision IEEE 754-format floating-point number
BINARY_DOUBLE	Double-precision IEEE 754-format floating-point number
NUMBER(prec, scale)	Fixed-point or floating-point number with absolute value in range 1E-130 to (but not including) 1.0E126. A NUMBER variable can also represent 0.
DEC(prec, scale)	ANSI specific fixed-point type with maximum precision of 38 decimal digits.
DECIMAL(prec, scale)	IBM specific fixed-point type with maximum precision of 38 decimal digits.
NUMERIC(pre, secale)	Floating type with maximum precision of 38 decimal digits.
DOUBLE PRECISION	ANSI specific floating-point type with maximum precision of 126 binary digits (approximately 38 decimal digits)
FLOAT	ANSI and IBM specific floating-point type with maximum precision of 126 binary digits (approximately 38 decimal digits)
INT	ANSI specific integer type with maximum precision of 38 decimal digits
INTEGER	ANSI and IBM specific integer type with maximum precision of 38 decimal digits
SMALLINT	ANSI and IBM specific integer type with maximum precision of 38 decimal digits
REAL	Floating-point type with maximum precision of 63 binary digits (approximately 18 decimal digits)

Example

```
DECLARE
     num1 INTEGER;
     num2 REAL;
     num3 DOUBLE PRECISION;
BEGIN
      null;
 END;
 /
```

Character Data Types and Subtypes

Data Type	Description
CHAR	Fixed-length character string with maximum size of 32,767 bytes
VARCHAR2	Variable-length character string with maximum size of 32,767 bytes
RAW	Variable-length binary or byte string with maximum size of 32,767 bytes, not interpreted by PL/SQL
NCHAR	Fixed-length national character string with maximum size of 32,767 bytes
NVARCHAR2	Variable-length national character string with maximum size of 32,767 bytes
LONG	Variable-length character string with maximum size of 32,760 bytes
LONG RAW	Variable-length binary or byte string with maximum size of 32,760 bytes, not interpreted by PL/SQL
ROWID	Physical row identifier, the address of a row in an ordinary table
UROWID	Universal row identifier (physical, logical, or foreign row identifier)

Datetime and Interval Types

Field Name	Valid Datetime Values	Valid Interval Values
YEAR	-4712 to 9999 (excluding year 0)	Any nonzero integer
MONTH	01 to 12	0 to 11
DAY	01 to 31 (limited by the values of MONTH and YEAR, according to the rules of the calendar for the locale)	Any nonzero integer
HOUR	00 to 23	0 to 23
MINUTE	00 to 59	0 to 59
SECOND	00 to 59.9(n), where 9(n) is the precision of time fractional seconds	0 to 59.9(n), where 9(n) is the precision of interval fractional seconds
TIMEZONE_HOUR	-12 to 14 (range accommodates daylight savings time changes)	Not applicable
TIMEZONE_MINUTE	00 to 59	Not applicable
TIMEZONE_REGION	Found in the dynamic performance view V$TIMEZONE_NAMES	Not applicable
TIMEZONE_ABBR	Found in the dynamic performance view V$TIMEZONE_NAMES	Not applicable

Large Object (LOB) Data Types

Data Type	Description	Size
BFILE	Used to store large binary objects in operating system files outside the database.	System-dependent. Cannot exceed 4 gigabytes (GB).
BLOB	Used to store large binary objects in the database.	8 to 128 terabytes (TB)
CLOB	Used to store large blocks of character data in the database.	8 to 128 TB
NCLOB	Used to store large blocks of NCHAR data in the database.	8 to 128 TB

User-Defined Subtypes

- A subtype has the same valid operations as its base type, but only a subset of its valid values.
  - SUBTYPE CHARACTER IS CHAR;
  - SUBTYPE INTEGER IS NUMBER(38,0);

- Example:

```
DECLARE
    SUBTYPE name IS char(20);
    SUBTYPE message IS varchar2(100);
    salutation name;
    greetings message;
BEGIN
    salutation := 'Reader ';
    greetings := 'Welcome to the World of PL/SQL';
    dbms_output.put_line('Hello ' || salutation || greetings);
END;
/
```

PL/SQL - Variables

- name given to a storage area that programs can manipulate
- Variable Declaration
  - declared in the declaration section
  - in a package as a global variable
  - Syntax :

variable_name [CONSTANT] data type [NOT NULL] [:= | DEFAULT initial_value]

  - Example :

```
sales number(10, 2);
pi CONSTANT double precision := 3.1415;
name varchar2(25);
address varchar2(100);
```

  - size, scale or precision limit is mentioned with the data type, it is called a constrained declaration.

Initializing Variables

- The **DEFAULT** keyword
- The **assignment** operator
- Example
  - counter binary_integer := 0;
  - greetings varchar2(20) DEFAULT 'Have a Good Day';
- specify that a variable should not have a **NULL** value using the **NOT NULL** constraint

Example

```
DECLARE
      a integer := 10;
      b integer := 20;
      c integer;
      f real;
BEGIN
      c := a + b;
      dbms_output.put_line('Value of c: ' || c);
      f := 70.0/3.0;
      dbms_output.put_line('Value of f: ' || f);
END;
 /
```

Variable Scope

- **Local variables** - variables declared in an inner block and not accessible to outer blocks.
- **Global variables** - variables declared in the outermost block or a package.

```
DECLARE
      -- Global variables
      num1 number := 95;
      num2 number := 85;
BEGIN
      dbms_output.put_line('Outer  Variable num1: ' || num1);
      dbms_output.put_line('Outer  Variable num2: ' || num2);
DECLARE
      -- Local variables
      num1 number := 195;
      num2 number := 185;
BEGIN
      dbms_output.put_line('Inner  Variable num1: ' || num1);
      dbms_output.put_line('Inner  Variable num2: ' || num2);
END;
END;
/
```

Assigning SQL Query Results to PL/SQL Variables

```
DECLARE
    c_id customers.id%type := 1;
    c_name customers.name%type;
    c_addr customers.address%type;
    c_sal customers.salary%type;
BEGIN
    SELECT name, address, salary INTO c_name, c_addr, c_sal
    FROM customers WHERE id = c_id;
    dbms_output.put_line ('Customer ' ||c_name || ' from ' ||
    c_addr || ' earns ' || c_sal);
END;
/
```

PL/SQL - Constants and Literals

- A constant holds a value that once declared, does not change in the program.
- A constant declaration specifies its name, data type, and value, and allocates storage for it.
- The declaration can also impose the NOT NULL constraint.
  - E.g. PI CONSTANT NUMBER := 3.141592654;

```
DECLARE
        -- constant declaration
       pi constant number := 3.141592654;
        -- other declarations
       radius number(5,2);
       dia number(5,2);
       circumference number(7, 2);
       area number (10, 2);
BEGIN
       -- processing
       radius := 9.5;
       dia := radius * 2;
       circumference := 2.0 * pi * radius;
       area := pi * radius * radius;
        -- output
       dbms_output.put_line('Radius: ' || radius);
       dbms_output.put_line('Diameter: ' || dia);
       dbms_output.put_line('Circumference: ' || circumference);
       dbms_output.put_line('Area: ' || area);
END;
 /
```

The PL/SQL Literals

- Numeric Literals
- Character Literals
- String Literals
- BOOLEAN Literals
- Date and Time Literals

Literal Type	Example:
Numeric Literals	050 78 -14 0 +32767 6.6667 0.0 -12.0 3.14159 +7800.00 6E5 1.0E-8 3.14159e0 -1E38 -9.5e-3
Character Literals	'A' '%' '9' ' ' 'z' '('
String Literals	'Hello, world!' '19-NOV-12'
BOOLEAN Literals	TRUE, FALSE, and NULL.
Date and Time Literals	DATE '1978-12-25'; TIMESTAMP '2012-10-29 12:01:01';

Example

```
DECLARE
    message varchar2(30):= ''That''s an example!'';
BEGIN
    dbms_output.put_line(message);
END;
/
```

Operators

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulation. PL/SQL language is rich in built-in operators and provides the following types of operators:

- Arithmetic operators
- Relational operators
- Comparison operators
- Logical operators
- String operators

Example

```
BEGIN
    dbms_output.put_line( 10 + 5);
    dbms_output.put_line( 10 - 5);
    dbms_output.put_line( 10 * 5);
    dbms_output.put_line( 10 / 5);
    dbms_output.put_line( 10 ** 5);
END;
/
```

```
DECLARE
        a number (2) := 21;
        b number (2) := 10;
BEGIN
        IF (a = b)
                then
                dbms_output.put_line('Line  1 - a is equal to b');
        ELSE
                 dbms_output.put_line('Line  1 - a is not equal to b');
        END IF;

        IF (a < b)
                 then
                dbms_output.put_line('Line  2 - a is less than b');
         ELSE
                dbms_output.put_line('Line  2 - a is not less than b');
         END IF;
         IF ( a <> b )
                THEN
                dbms_output.put_line('Line  6 - a is not equal to b');
        ELSE
                dbms_output.put_line('Line  6 - a is equal to b');
         END IF;
        END;
        /
```

Like - example

```
DECLARE
PROCEDURE compare (value varchar2, pattern varchar2 ) is
BEGIN
IF value LIKE pattern THEN
dbms_output.put_line  ('True');
ELSE
dbms_output.put_line  ('False');
END IF;
END;
BEGIN
compare(tharun', ‘t%u_i');
compare(‘bala', ‘b%l_i');
END;
/
```

Between - example

```
DECLARE
x number(2) := 10;
BEGIN
IF (x between 5 and 20) THEN
dbms_output.put_line('True');
ELSE
dbms_output.put_line('False');
END IF;
IF (x BETWEEN 5 AND 10) THEN
dbms_output.put_line('True');
ELSE
dbms_output.put_line('False');
END IF;
IF (x BETWEEN 11 AND 20) THEN
dbms_output.put_line('True');
ELSE
dbms_output.put_line('False');
END IF;
END;
/
```

In - example

```
DECLARE
letter varchar2(1) := 'm';
BEGIN
        IF (letter in ('a', 'b', 'c')) THEN
        dbms_output.put_line('True');
        ELSE
        dbms_output.put_line('False');
        END IF;
        IF (letter in ('m', 'n', 'o')) THEN
        dbms_output.put_line('True');
        ELSE
        dbms_output.put_line('False');
        END IF;
        IF (letter is null) THEN
        dbms_output.put_line('True');
        ELSE
        dbms_output.put_line('False');
        END IF;
END;
/
```

PL/SQL - Conditions

Statement	Description
IF - THEN statement	The **IF statement** associates a condition with a sequence of statements enclosed by the keywords **THEN** and **END IF**. If the condition is true, the statements get executed and if the condition is false or NULL then the IF statement does nothing.
IF-THEN-ELSE statement	**IF statement** adds the keyword **ELSE** followed by an alternative sequence of statement. If the condition is false or NULL , then only the alternative sequence of statements get executed. It ensures that either of the sequence of statements is executed.
IF-THEN-ELSIF statement	It allows you to choose between several alternatives.
Case statement	Like the IF statement, the **CASE statement** selects one sequence of statements to execute. However, to select the sequence, the CASE statement uses a selector rather than multiple Boolean expressions. A selector is an expression whose value is used to select one of several alternatives.
Searched CASE statement	The searched CASE statement **has no selector**, and it's WHEN clauses contain search conditions that yield Boolean values.
nested IF-THEN-ELSE	You can use one **IF-THEN** or **IF-THEN-ELSIF** statement inside another **IF-THEN** or **IF-THEN-ELSIF** statement(s).

Example

```
DECLARE
      c_id customers.id%type := 1;
      c_sal customers.salary%type;
BEGIN
      SELECT salary INTO c_sal FROM customers WHERE id = c_id;
IF (c_sal <= 2000)
      THEN
      UPDATE customers SET salary = salary + 1000 WHERE id = c_id;
      dbms_output.put_line ('Salary updated');
END IF;
END;
 /
```

```
DECLARE
grade char(1) := 'A';
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('Well done');
      when 'D' then dbms_output.put_line('You passed');
      when 'F' then dbms_output.put_line('Better try again');
      else dbms_output.put_line('No such grade');
END CASE;
END;
/
```

Searched CASE Statement

```
DECLARE
grade char(1) := 'A';
BEGIN
CASE
     when grade='A' then dbms_output.put_line('Excellent');
     when grade='B' then dbms_output.put_line('Very good');
     when grade='C' then dbms_output.put_line('Well done');
     when grade='D' then dbms_output.put_line('You passed');
     when grade='F' then dbms_output.put_line('Better try again');
     else dbms_output.put_line('No such grade');
END CASE;
END;
/
```

PL/SQL Loops

Loop Type	Description
PL/SQL Basic LOOP	In this loop structure, sequence of statements is enclosed between the LOOP and END LOOP statements. At each iteration, the sequence of statements is executed and then control resumes at the top of the loop.
PL/SQL WHILE LOOP	Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.
PL/SQL FOR LOOP	Execute a sequence of statements multiple times and abbreviates the code that manages the loop variable.
Nested loops in PL/SQL	You can use one or more loop inside any another basic loop, while or for loop.

Basic Loop - example

```
DECLARE
x number := 10;
BEGIN
LOOP
dbms_output.put_line(x);
x := x + 10;
IF x > 50 THEN
exit;
END IF;
END LOOP;
-- after exit, control resumes here
dbms_output.put_line('After  Exit x is: ' || x);
END
```

For - example

```
DECLARE
a number(2);
BEGIN
FOR a in 10 .. 20 LOOP
dbms_output.put_line('value of a: ' || a);
END LOOP;
END;
/
```

Reverse For - example

```
DECLARE
a number(2) ;
BEGIN
FOR a IN REVERSE 10 .. 20 LOOP
dbms_output.put_line('value of a: ' || a);
END LOOP;
END;
/
```

Labeling Loop - example

```
DECLARE
i number(1);
j number(1);
BEGIN
<< outer_loop >>
FOR i IN 1..3 LOOP
<< inner_loop >>
FOR j IN 1..3 LOOP
dbms_output.put_line('i is: '|| i || ' and j is: ' || j);
END loop inner_loop;
END loop outer_loop;
END;
/
```

The Loop Control Statements

Control Statement	Description
EXIT statement	The Exit statement completes the loop and control passes to the statement immediately after END LOOP
CONTINUE statement	Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.
GOTO statement	Transfers control to the labeled statement. Though it is not advised to use GOTO statement in your program.

String - example

```
DECLARE
greetings varchar2(11) := 'hello world';
BEGIN
dbms_output.put_line(UPPER(greetings));
dbms_output.put_line(LOWER(greetings));
dbms_output.put_line(INITCAP(greetings));
/* retrieve the first character in the string */
dbms_output.put_line ( SUBSTR (greetings, 1, 1));
/* retrieve the last character in the string */
dbms_output.put_line ( SUBSTR (greetings, -1, 1));
/* retrieve five characters,
starting from the seventh position. */
dbms_output.put_line ( SUBSTR (greetings, 7, 5));
/* retrieve the remainder of the string,
starting from the second position. */
dbms_output.put_line ( SUBSTR (greetings, 2));
/* find the location of the first "e" */
dbms_output.put_line ( INSTR (greetings, 'e'));
END;
/
```

PL/SQL - Arrays

- TYPE varray_type_name IS VARRAY(n) of <element_type>
  - *varray_type_name* is a valid attribute name,
  - *n* is the number of elements (maximum) in the varray,
  - *element_type* is the data type of the elements of the array.

  - E.g.
    - TYPE namearray IS VARRAY(5) OF VARCHAR2(10);
    - Type grades IS VARRAY(5) OF INTEGER;

Example 1

```
DECLARE
type namesarray IS VARRAY(5) OF VARCHAR2(10);
type grades IS VARRAY(5) OF INTEGER;
names namesarray;
marks grades;
total integer;
BEGIN
names := namesarray('Raj', 'Priya', 'Ram', 'Ashish', 'Tharun');
marks:= grades(98, 97, 78, 87, 92);
total := names.count;
dbms_output.put_line('Total '|| total || ' Students');
FOR i in 1 .. total LOOP
dbms_output.put_line('Student: ' || names(i) || '
Marks: ' || marks(i));
END LOOP;
END;
/
```

Example 2

```
DECLARE
    CURSOR c_customers is
    SELECT name FROM customers;
    type c_list is varray (6) of customers.name%type;
    name_list c_list := c_list();
    counter integer :=0;
BEGIN
    FOR n IN c_customers LOOP
        counter := counter + 1;
        name_list.extend;
        name_list(counter) := n.name;
        dbms_output.put_line('Customer('||counter||'):'||name_list(counter));

END LOOP;
END; /
```

PL/SQL - Procedures

- A **subprogram** is a program unit/module that performs a particular task.
- These subprograms are combined to form larger programs. This is basically called the 'Modular design'.
- A subprogram can be invoked by another subprogram or program which is called the calling program.

Types of subprograms

- PL/SQL subprograms are named PL/SQL blocks that can be invoked with a set of parameters.
- PL/SQL provides two kinds of subprograms:
  - **Functions**: these subprograms return a single value, mainly used to compute and return a value.
  - **Procedures**: these subprograms do not return a value directly, mainly used to perform an action.

Creating a Procedure

- A procedure is created with the CREATE OR REPLACE PROCEDURE statement.

```
CREATE [OR REPLACE] PROCEDURE procedure_name [(parameter_name
    [IN | OUT | IN OUT] type [, ...])] {IS | AS}
BEGIN
    < procedure_body >
END procedure_name;
```

- *procedure-name* specifies the name of the procedure.
- [OR REPLACE] option allows modifying an existing procedure.
- The optional parameter list contains name, mode and types of the parameters. IN represents that value will be passed from outside and OUT represents that this parameter will be used to return a value outside of the procedure.
- *procedure-body* contains the executable part.
- The AS keyword is used instead of the IS keyword for creating a standalone procedure.

Example

```
CREATE OR REPLACE PROCEDURE greetings
AS
BEGIN
dbms_output.put_line('Hello World!');
END;
/
```

Executing a Standalone Procedure

- A standalone procedure can be called in two ways:
  - Using the EXECUTE keyword
    - EXECUTE greetings;
  - Calling the name of the procedure from a PL/SQL block

```
BEGIN
greetings;
END;
/
```

Deleting a Standalone Procedure

- DROP PROCEDURE procedure-name;
  - Example :

```
BEGIN
DROP PROCEDURE greetings;
END;
/
```

Parameter Modes

IN -

1. **An IN parameter lets you pass a value to the subprogram.**
2. **It is a read-only parameter.**
3. **IN parameter acts like a constant.**
4. **It is the default mode of parameter passing.**
5. **Parameters are passed by reference.**

OUT –

1. **An OUT parameter returns a value to the calling program.**
2. **OUT parameter acts like a variable.**
3. **Its value can change and reference the value after assigning it.**
4. **The actual parameter must be variable and it is passed by value.**

IN OUT –

1. **An IN OUT parameter passes an initial value to a subprogram and returns an updated value to the caller.**
2. **It can be assigned a value and its value can be read.**
3. **The actual parameter corresponding to an IN OUT formal parameter must be a variable, not a constant or an expression.**
4. **Formal parameter must be assigned a value.**
5. **Actual parameter is passed by value.**

```
DECLARE
      a number;
      b number;
      c number;
PROCEDURE findMin(x IN number, y IN number, z OUT number) IS
BEGIN
      IF x < y THEN
      z:= x;
      ELSE
      z:= y;
      END IF;
END;
BEGIN
      a:= 23;
      b:= 45;
      findMin(a, b, c);
      dbms_output.put_line(' Minimum of (23, 45) : ' || c);
END;
/
```

```
DECLARE
    a number;
PROCEDURE squareNum(x IN OUT number) IS
BEGIN
    x := x * x;
END;

BEGIN
    a:= 23;
    squareNum(a);
    dbms_output.put_line('  Square of (23): ' || a);
END;
/
```

Methods for Passing Parameters

- Actual parameters could be passed in three ways:
  - Positional notation
    - findMin(a, b, c, d);
  - Named notation
    - findMin(x=>a, y=>b, z=>c, m=>d);
  - Mixed notation
    - positional notation should precede the named notation
    - findMin(a, b, c, m=>d); // legal
    - findMin(x=>a, b, c, d); //illegal

PL/SQL Function

- A PL/SQL function is same as a procedure except that it returns a value.

Creating a Function

- Syntax for the CREATE OR REPLACE PROCEDURE statement is as follows:

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

Example

The following example ,function returns the total number of CUSTOMERS in the customers table.

Select * from customers;

```
+----+----------+-----+----------+----------+
| ID | NAME | AGE | ADDRESS | SALARY |
+----+----------+-----+----------+----------+
| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |
| 2 | Khilan | 25 | Delhi | 1500.00 |
| 3 | kaushik | 23 | Kota | 2000.00 |
| 4 | Chaitali | 25 | Mumbai | 6500.00 |
| 5 | Hardik | 27 | Bhopal | 8500.00 |
| 6 | Komal | 22 | MP | 4500.00 |
+----+----------+-----+----------+----------+
```

```
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;
/
```

Example 2

```
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;
/
```

PL/SQL Recursive Functions

- When a subprogram calls itself, it is referred to as a recursive call and the process is known as recursion
  - Factorial of a number n is defined as:

    n! = n*(n-1)!

    = n*(n-1)*(n-2)!

    ...

    = n*(n-1)*(n-2)*(n-3)... 1

Example - factorial

```
DECLARE
      num number;
      factorial number;
FUNCTION fact(x number)
RETURN number IS
f number;
BEGIN
            IF x=0 THEN
                  f := 1;
            ELSE
                  f := x * fact(x-1);
            END IF;
      RETURN f;
END;

BEGIN
      num:= 6;
      factorial := fact(num);
      dbms_output.put_line(' Factorial '|| num || ' is ' || factorial);
END;
/
```

Exceptions

- There are two types of exceptions:
  - System-defined exceptions
  - User-defined exceptions

- Syntax for Exception Handling

```
DECLARE
<declarations section>
BEGIN
<executable command(s)>
EXCEPTION
<exception handling goes here >
WHEN exception1 THEN
exception1-handling-statements
WHEN exception2 THEN
exception2-handling-statements
WHEN exception3 THEN
exception3-handling-statements
.....
WHEN others THEN
exception3-handling-statements
END;
```

Example

```
DECLARE
    c_id customers.id%type := 8;
    c_name customers.name%type;
    c_addr customers.address%type;
BEGIN
    SELECT name, address INTO c_name, c_addr
    FROM customers
    WHERE id = c_id;

    DBMS_OUTPUT.PUT_LINE ('Name: '|| c_name);
    DBMS_OUTPUT.PUT_LINE ('Address: ' || c_addr);

EXCEPTION
    WHEN no_data_found THEN
    dbms_output.put_line('No such customer!');
    WHEN others THEN
    dbms_output.put_line('Error!');
END;
/
```

Raising Exceptions

```
DECLARE
    exception_name EXCEPTION;
BEGIN
    IF condition THEN
    RAISE exception_name;
    END IF;
EXCEPTION
    WHEN exception_name THEN
    statement;
END;
```

Triggers

- Triggers are stored programs, which are automatically executed or fired when some events occur.
  - 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

- 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

```
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;
```

Example

```
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;
/
```

Important

- OLD and NEW references are not available for table level triggers, rather you can use them for record level triggers.
- To query the table in the same trigger, use the AFTER keyword.

- Triggering a Trigger ??????

Transactions

- A database transaction is an atomic unit of work that may consist of one or more related SQL statements.
- Atomic because the database modifications brought about by the SQL statements that constitute a transaction can collectively be either committed, i.e., made permanent to the database or rolled back (undone) from the database.
  - Commit
  - Savepoint sv1
  - Rollback sv1

  - Set autocommit on/off

Advantages of PL/SQL

- ***Block Structures***:
  - PL SQL consists of blocks of code, which can be nested within each other. Each block forms a unit of a task or a logical module. PL/SQL Blocks can be stored in the database and reused.

- ***Procedural Language Capability***:
  - PL SQL consists of procedural language constructs such as conditional statements (if else statements) and loops like (FOR loops).

- ***Better Performance***:
  - PL SQL engine processes multiple SQL statements simultaneously as a single block, thereby reducing network traffic.

Thank You