

CHAPTER-2

PL/SQL

❖ Introduction

- Programming language/structured query language
- It provide facility of procedural and function
- It provides programming techniques like branching, looping, and condition check.
- It is fully structured programming language.
- It decreases the network traffic because entire block of code is passed to the DBA at one time for execution.
- Basic Structure of PL/SQL
 - PL/SQL stands for Procedural Language/SQL. PL/SQL extends SQL by adding constructs found in procedural languages, resulting in a structural language that is more powerful than SQL. The basic unit in PL/SQL is a *block*. All PL/SQL programs are made up of blocks, which can be nested within each other. Typically, each block performs a logical action in the program. A block has the following structure:
 - DECLARE
 - ◆ /* Declarative section: variables, types, and local subprograms. */
 - BEGIN
 - ◆ /* Executable section: procedural and SQL statements go here. */
 - ◆ /* This is the only section of the block that is required. */
 - EXCEPTION
 - ◆ /* Exception handling section: error handling statements go here. */
 - END;

❖ Advantages of PL/SQL

- PL/SQL is development tool that not only supports SQL data manipulation but also provide facilities of conditional checking, branching and looping.
- PL/SQL sends an entire block of statements to the Oracle engine at one time. This in turn reduces network traffic. The Oracle engine gets the SQL statements as a single block, and hence it processes this code much faster than if it got the code one sentence at a time. There is a definite improvement in the performance time of the Oracle engine. As an entire block of code is passed to the DBA at one time for execution, all changes made to the data in the table are done or undone, in one go.
- PL/SQL also permits dealing with errors as required, and facilitates displaying user-friendly messages, when errors are encountered.

- PL/SQL allows declaration and use of variables in blocks of code. These variables can be used to store intermediate results of a query for later processing, or calculate values and insert them into an Oracle table later. PL/SQL variables can be used anywhere, either in SQL statements or in PL/SQL blocks.
- Via PL/SQL, all sorts of calculations can be done quickly and efficiently without the use of the Oracle engine. This considerably improves transaction performance.
- Applications written in PL/SQL are portable to any computer hardware and operating system, where Oracle is operational. Hence, PL/SQL code blocks written for a DOS version of Oracle will run on its UNIX version, without any modifications at all.
- Support for SQL
- Support for object-oriented programming
- Better performance
- Higher productivity
- Full portability
- Tight integration with Oracle
- Tight security

❖ Data types

- Rowid Datatypes

Data Type Syntax	Oracle 10g	Explanation (if applicable)
Rowid	<p>The format of the rowid is: BBBB.BB.RRRR.FFFFF</p> <p>Where BBBB.BB is the block in the database file; RRRR is the row in the block; FFFFF is the database file.</p>	Fixed-length binary data. Every record in the database has a physical address or rowid .
Urowid(size)		<p>Universal rowid.</p> <p>Where size is optional.</p>

➤ Character Datatypes

Data Type Syntax	Oracle 10g	Explanation (if applicable)
char(size)	Maximum size of 2000 bytes.	Where size is the number of characters to store. Fixed-length strings. Space padded.
nchar(size)	Maximum size of 2000 bytes.	Where size is the number of characters to store. Fixed-length NLS string Space padded.
nvarchar2(size)	Maximum size of 4000 bytes.	Where size is the number of characters to store. Variable-length NLS string.
Varchar2(size)	Maximum size of 4000 bytes.	Where size is the number of characters to store. Variable-length string.
Long	Maximum size of 2GB.	Variable-length strings. (backward compatible)
Raw	Maximum size of 2000 bytes.	Variable-length binary strings
long raw	Maximum size of 2GB.	Variable-length binary strings. (backward compatible)

➤ Numeric Data types

Data Type Syntax	Oracle 10g	Explanation (if applicable)
number(p,s)	Precision can range from 1 to 38. Scale can range from -84 to 127.	Where p is the precision and s is the scale. For example, number(7,2) is a number that has 5 digits before the decimal and 2 digits after the decimal.
numeric(p,s)	Precision can range from 1 to 38.	Where p is the precision and s is the scale. For example, numeric(7,2) is a number that has 5 digits before the decimal and 2 digits after the decimal.
dec(p,s)	Precision can range from 1 to 38.	Where p is the precision and s is the scale. For example, dec(3,1) is a number that has 2 digits before the decimal and 1 digit after the decimal.
decimal(p,s)	Precision can range from 1 to 38.	Where p is the precision and s is the scale. For example, decimal(3,1) is a number that has 2

		digits before the decimal and 1 digit after the decimal.
--	--	--

It also contains integer ,float ,int data types to store numeric values

➤ Date/Time Datatypes

Data Type Syntax	Oracle 10g	Explanation (if applicable)
Date	A date between Jan 1, 4712 BC and Dec 31, 9999 AD.	
timestamp (fractional seconds precision)	Fractional second's precision must be a number between 0 and 9. (default is 6)	Includes year, month, day, hour, minute, and seconds. For example: timestamp(6)

❖ Control structures

➤ The flow of control statements can be classified into the following categories:

- Conditional Control
- Iterative Control
- Sequential Control

➤ Conditional Control

- PL/SQL allows the use of an **IF** statement to control the execution of a block of code.
- In PL/SQL, the **IF -THEN - ELSIF - ELSE - END IF** construct in code blocks allow specifying certain conditions under which a specific block of code should be executed.

- **Syntax:**

IF < Condition > THEN

< Action >

ELSIF <Condition> THEN

< Action >

ELSE

< Action >

END IF;

▪ *Example Using the IF-THEN-ELSEIF Statement*

```

DECLARE
    a Number := 30;
    b Number;
BEGIN
    IF a > 40 THEN
        b := a - 40;
        DBMS_OUTPUT.PUT_LINE('b=' || b);
    elsif a = 30 then
        b := a + 40;
        DBMS_OUTPUT.PUT_LINE('b=' || b);
    ELSE
        b := 0;
        DBMS_OUTPUT.PUT_LINE('b=' || b);
    END IF;
END;
/

```

➤ **Iterative Control**

- Iterative control indicates the ability to repeat or skip sections of a code block.
- A **loop** marks a sequence of statements that has to be repeated. The keyword **loop** has to be placed before the first statement in the sequence of statements to be repeated, while the keyword **end loop** is placed immediately after the last statement in the sequence.
- Once a loop begins to execute, it will **go on forever**. Hence a conditional statement that controls the number of times a loop is executed **always accompanies** loops.
- PL/SQL supports the following structures for iterative control:

▪ **Simple Loop**

- In simple loop, the key word **loop** should be placed before the first statement in the sequence and the keyword **end loop** should be written at the end of the sequence to end the loop.

▪ **Syntax:**

Loop

< Sequence of statements >

End loop;

- **Example:** Create a simple loop such that a message is displayed when a loop exceeds a particular value.

```

DECLARE
    i number := 0;
BEGIN
    LOOP
        i := i + 2;
        EXIT WHEN i < 10;
    END LOOP;
    dbms_output.put_line ('Loop exited as the value of i has reached' || to_char (i));
END;
```

- **Output:**

Loop exited as the value of i has reached 12
 PL/SQL procedure successfully completed.

- **The WHILE loop**

- **Syntax:**

```

WHILE<Condition>
LOOP
    <Action>
END LOOP;
```

- **Example 3:** Write a PL/SQL code block to calculate the area of a circle for a value of radius varying from 3 to 7. Store the radius and the corresponding values of calculated area in an empty table named Areas, consisting of two columns **Radius** and **Area**.

- **Table Name:** Areas

RADIUS	AREA
--------	------

- **CREATE TABLE AREAS (RADIUS NUMBER(5), AREA NUMBER(14,2));**

```

DECLARE
    pi constant number(4,2) := 3.14 ;
    radius number(5);
    area number( 14,2);
BEGIN
    radius := 3;
    WHILE RADIUS <= 7
    LOOP
```

```

        area := pi * power(radius,2);
        INSERT INTO areas VALUES (radius, area);
        radius := radius + 1;
    END LOOP;
END;
```

- After the loop is completed the table will now hold the following:

Table Name: **Areas**

RADIUS	AREA
3	28.26
4	50.24
5	78.5
6	113.04
7	153.86

- **The FOR Loop**

- Syntax:

```

FOR variable IN [REVERSE] start..end
LOOP
    <Action>
END LOOP;
```

- **Note**

- **The variable** in the For Loop need not be declared. Also the increment value cannot be specified. The For Loop variable is **always incremented by 1**.

- **Example:** Write a PL/SQL block of code for inverting a number 5639 to 9365.

DECLARE

```

    given_number varchar (5) := '5639';
    str_length number(2);
    inverted_number varchar(5);
```

BEGIN

```

    Str_length := length(given_number);
```

```

    FOR cntr IN REVERSE 1..str_length
```

```

    /* Variables used as counter in the for loop need not be declared i.e. cntr declaration is not
    required */
```

```

    LOOP
```

```

        inverted_number := inverted_number || substr(given_number, cntr, 1);
```

```

    END LOOP;
```

```

    dbms_output.put_line (The Given number is ' || given_number);
```

```

    dbms_output.put_line (The Inverted number is ' || inverted_number);
END;

```

▪ **Output:**

The Given number is 5639

The Inverted number is 9365

➤ **Sequential Control**

▪ **The GOTO Statement**

- The **GOTO** statement changes the **flow of control** within a PL/SQL block.
- This statement allows execution of a section of code, which is not in the normal flow of control.
- The entry point into such a block of code is marked using the tags «**userdefined name**».
- **The GOTO** statement can then make use of this user-defined name to jump into that block of code for execution.

▪ **Syntax:**

- **GOTO** <codeblock name>;

▪ **Example : Using a Simple GOTO Statement**

DECLARE

p VARCHAR2(30);

n number := 37; -- test any integer > 2 for prime

BEGIN

FOR j in 2..ROUND(SQRT(n)) LOOP

IF n MOD j = 0 THEN -- test for prime

p := ' is not a prime number'; -- not a prime number

GOTO print_now;

END IF;

END LOOP;

p := ' is a prime number';

<<print_now>>

DBMS_OUTPUT.PUT_LINE (TO_CHAR(n) || p);

END;

/

❖ **Concepts of error handling**

➤ **Exceptions**

- An Exception is an error situation, which arises during program execution. When an error occurs exception is raised, normal execution is stopped and control transfers to exception-handling part. Exception handlers are routines written to handle the exception. The exceptions can be internally defined (system-defined or pre-defined) or User-defined exception.

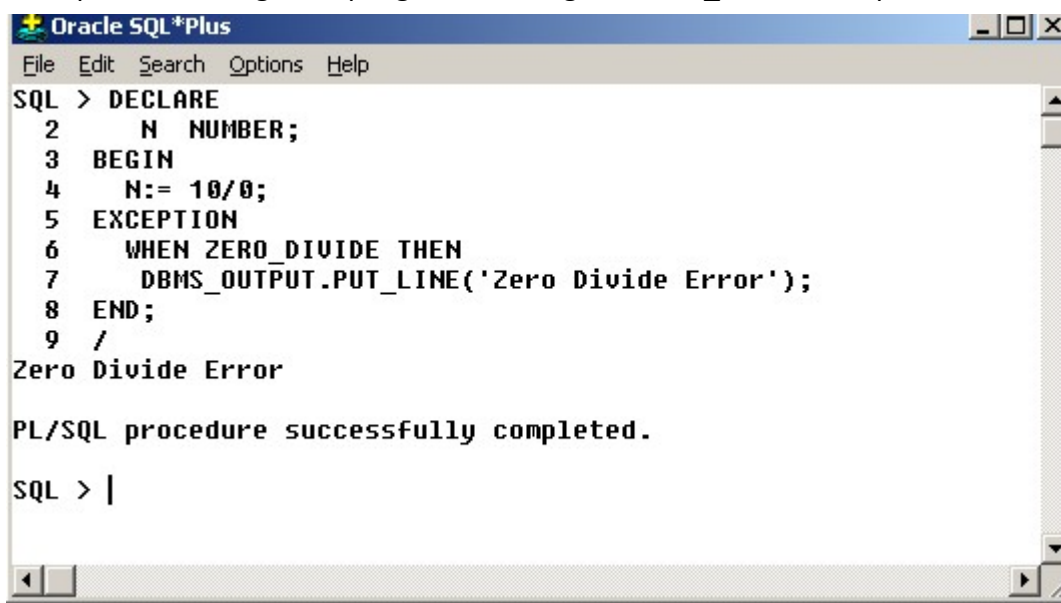
EXCEPTION

WHEN <ExceptionName> THEN

<User Defined Action To Be Carried Out>

- **Predefined exception** is raised automatically whenever there is a violation of Oracle coding rules. Predefined exceptions are those like ZERO_DIVIDE, which is raised automatically when we try to divide a number by zero. Other built-in exceptions are given below. You can handle unexpected Oracle errors using OTHERS handler. It can handle all raised exceptions that are not handled by any other handler. It must always be written as the last handler in exception block.
 - CURSOR_ALREADY_OPEN – Raised when we try to open an already open cursor.
 - DUP_VAL_ON_INDEX – When you try to insert a duplicate value into a unique column
 - INVALID_CURSOR – It occurs when we try accessing an invalid cursor
 - INVALID_NUMBER – On usage of something other than number in place of number value.
 - LOGIN_DENIED – At the time when user login is denied
 - TOO_MANY_ROWS – When a select query returns more than one row and the destination variable can take only single value.
 - VALUE_ERROR – When an arithmetic, value conversion, truncation, or constraint error occurs.
- Predefined exception handlers are declared globally in package STANDARD. Hence we need not have to define them rather just use them.
- The biggest advantage of exception handling is it improves readability and reliability of the code. Errors from many statements of code can be handles with a single handler. Instead of checking for an error at every point we can just add an exception handler and if any exception is raised it is handled by that.
- For checking errors at a specific spot it is always better to have those statements in a separate begin – end block

- Examples: Following example gives the usage of ZERO_DIVIDE exception



```

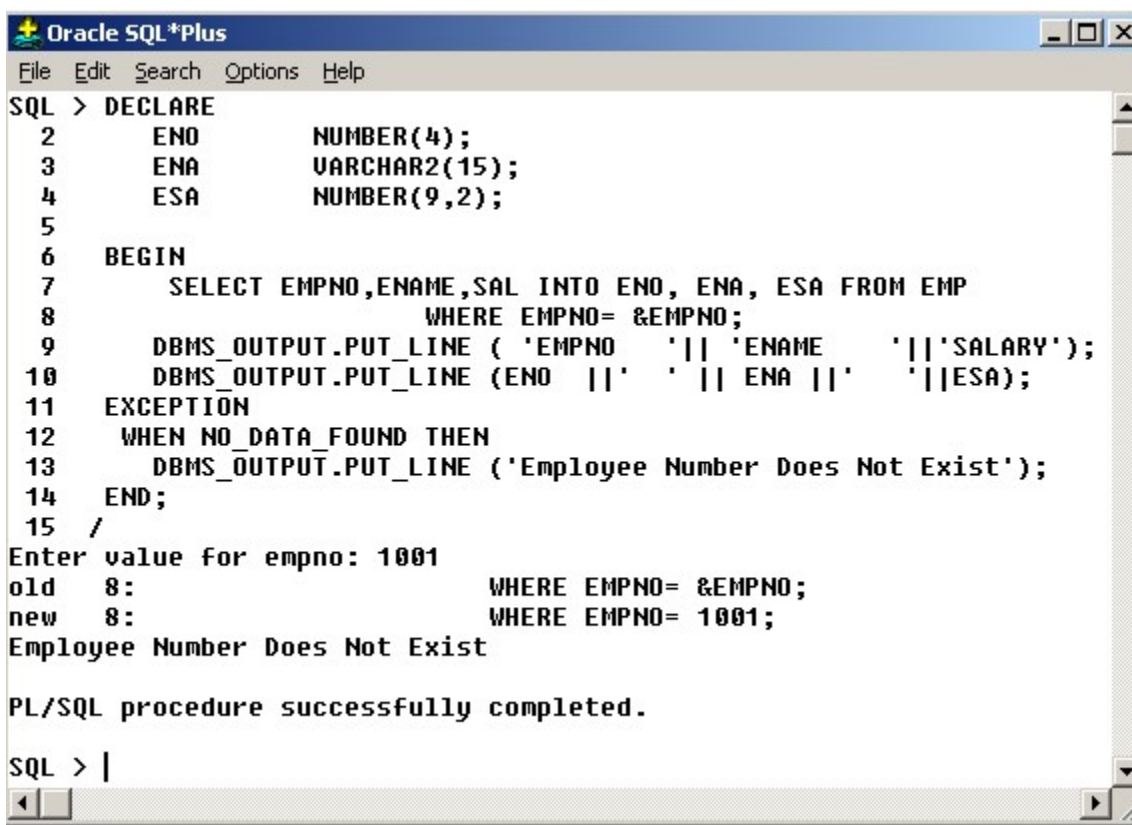
Oracle SQL*Plus
File Edit Search Options Help
SQL > DECLARE
  2   N   NUMBER;
  3   BEGIN
  4     N:= 10/0;
  5   EXCEPTION
  6     WHEN ZERO_DIVIDE THEN
  7       DBMS_OUTPUT.PUT_LINE('Zero Divide Error');
  8   END;
  9   /
Zero Divide Error

PL/SQL procedure successfully completed.

SQL > |

```

- Example 2: I have explained the usage of NO_DATA_FOUND exception in the following



```

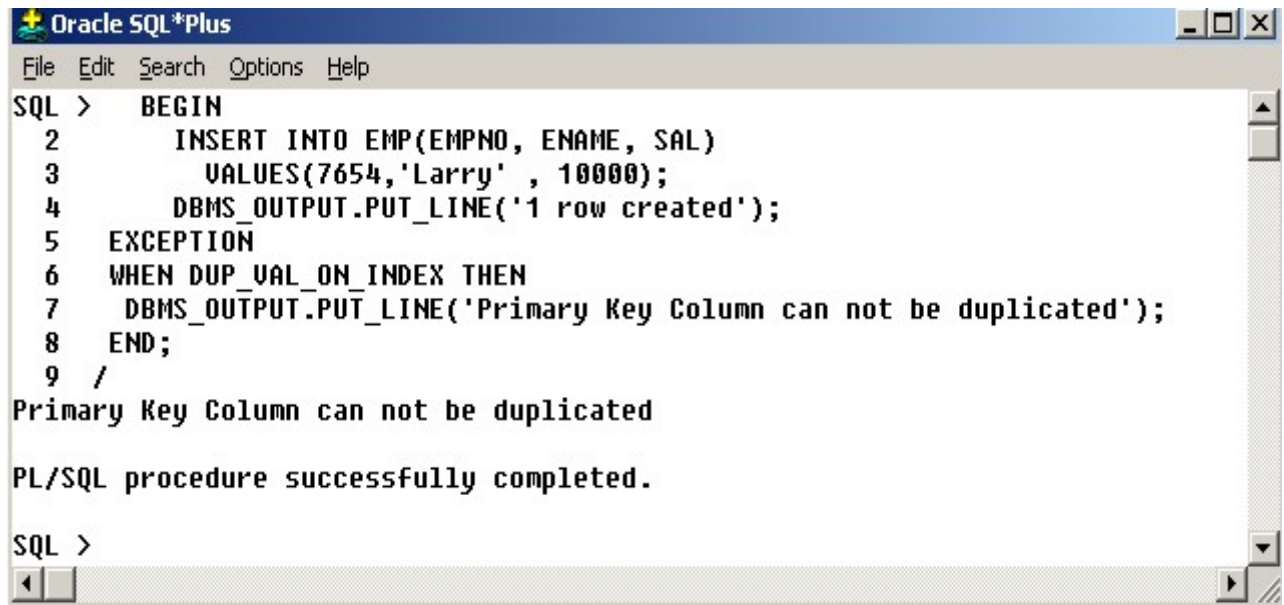
Oracle SQL*Plus
File Edit Search Options Help
SQL > DECLARE
  2   ENO      NUMBER(4);
  3   ENA      VARCHAR2(15);
  4   ESA      NUMBER(9,2);
  5
  6   BEGIN
  7     SELECT EMPNO,ENAME,SAL INTO ENO, ENA, ESA FROM EMP
  8           WHERE EMPNO= &EMPNO;
  9     DBMS_OUTPUT.PUT_LINE ( 'EMPNO   || 'ENAME   || 'SALARY');
 10     DBMS_OUTPUT.PUT_LINE (ENO || '   ' || ENA || '   ' || ESA);
 11   EXCEPTION
 12     WHEN NO_DATA_FOUND THEN
 13       DBMS_OUTPUT.PUT_LINE ('Employee Number Does Not Exist');
 14   END;
 15   /
Enter value for empno: 1001
old   8:                WHERE EMPNO= &EMPNO;
new   8:                WHERE EMPNO= 1001;
Employee Number Does Not Exist

PL/SQL procedure successfully completed.

SQL > |

```

- The **DUP_VAL_ON_INDEX** is raised when a SQL statement tries to create a duplicate value in a column on which primary key or unique constraints are defined.
- Example to demonstrate the exception DUP_VAL_ON_INDEX.



```

Oracle SQL*Plus
File Edit Search Options Help
SQL > BEGIN
2     INSERT INTO EMP(EMPNO, ENAME, SAL)
3     VALUES(7654,'Larry' , 10000);
4     DBMS_OUTPUT.PUT_LINE('1 row created');
5 EXCEPTION
6 WHEN DUP_VAL_ON_INDEX THEN
7     DBMS_OUTPUT.PUT_LINE('Primary Key Column can not be duplicated');
8 END;
9 /
Primary Key Column can not be duplicated

PL/SQL procedure successfully completed.

SQL >

```

➤ User-defined Exceptions

- The technique that is used is to bind a numbered exception handler to a name using **Pragma Exception_init ()**. This binding of a numbered exception handler, to a name (i.e. a **String**), is done in the Declare section of a PL/SQL block.
- The **Pragma** action word is a call to a pre-compiler, which immediately binds the numbered exception handler to a name when encountered.
- The function **Exception_init()** takes two parameters the first is the user defined exception name the second is the Oracle engine's exception number. These lines will be included in the **Declare** section of the PL/SQL block.
- *The user defined exception name must be the statement that immediately precedes the Pragma **Exception_init()** statement.*

- **Syntax:**

DECLARE

< ExceptionName > EXCEPTION ;

PRAGMA EXCEPTION_INIT (< ExceptionName >, <ErrorCodeNo>);

BEGIN

- Using this technique it is possible to bind appropriate numbered exception handlers to names and use these names in the Exception section of a PL/SQL block. When this is done the default exception handling code of the exception handler is overridden and the user-defined exception handling code is executed

- **Syntax:**

```

DECLARE
    <ExceptionName> EXCEPTION;
    PRAGMA EXCEPTION_INIT (<ExceptionName>.<ErrorCodeNo>);
BEGIN
    . . . .
EXCEPTION
    WHEN <ExceptionName> THEN
        <Action>
END;
```

- **Example:**

- SQL> -- create demo table
- SQL> create table Employee8(
 ID VARCHAR2(4 BYTE) notnull ,
 First_Name VARCHAR2(10 BYTE),
 Last_Name VARCHAR2(10 BYTE),
 Start_Date DATE,
 End_Date DATE,
 Salary Number(8,2),
 City VARCHAR2(10 BYTE),
 Description VARCHAR2(15 BYTE)
)
 /
- SQL> DECLARE
 e_MissingNull EXCEPTION;
 PRAGMA EXCEPTION_INIT(e_MissingNull, -1400);
 BEGIN
 INSERT INTO Employee8 (id) VALUES (NULL);
 EXCEPTION
 WHEN e_MissingNull then
 DBMS_OUTPUT.put_line('ORA-1400 occurred');
 END;
 /
 ORA-1400 occurred

PL/SQL procedure successfully completed.

➤ **User Defined Exception Handling (For Business Rule Validations)**

- To trap business rules being violated the technique of **raising** user-defined exceptions and then **handling** them, is used.
- User-defined error conditions must be declared in the declarative part of any PL/SQL block. In the executable part, a check for the condition that needs special attention is made. If that condition exists, the call to the user-defined exception is made using a RAISE statement. The exception once raised is then handled in the Exception handling section of the PL/SQL code block.
- **Syntax:**

DECLARE

<ExceptionName> Exception

BEGIN

<SQL Sentence >;

IF < Condition > THEN

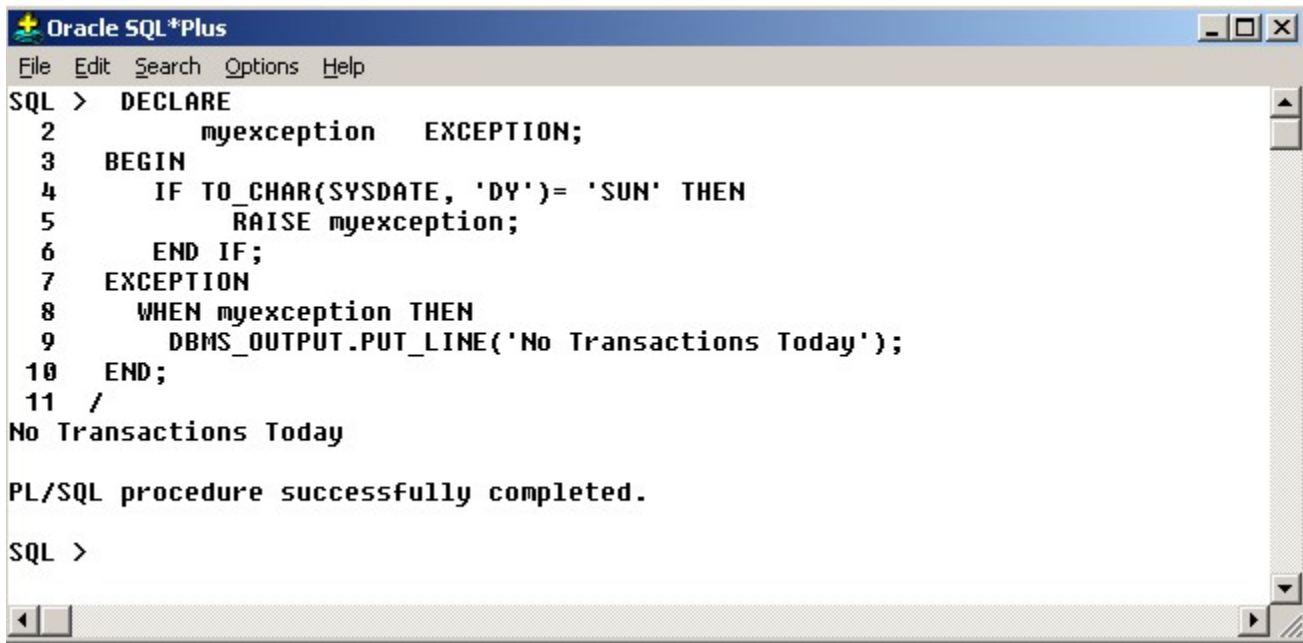
RAISE <ExceptionName>;

END IF;

EXCEPTION

WHEN <ExceptionName> THEN {User Defined Action To Be Taken};

- **END;**
- The following example explains the usage of User-defined Exception



```

Oracle SQL*Plus
File Edit Search Options Help
SQL > DECLARE
2      myexception  EXCEPTION;
3  BEGIN
4      IF TO_CHAR(SYSDATE, 'DY')= 'SUN' THEN
5          RAISE myexception;
6      END IF;
7  EXCEPTION
8      WHEN myexception THEN
9          DBMS_OUTPUT.PUT_LINE('No Transactions Today');
10 END;
11 /
No Transactions Today

PL/SQL procedure successfully completed.

SQL >

```

➤ **Points To Ponder:**

- An Exception cannot be declared twice in the same block.
- Exceptions declared in a block are considered as local to that block and global to its sub-blocks.
- An enclosing block cannot access Exceptions declared in its sub-block. Where as it possible for a sub-block to refer its enclosing Exceptions

❖ **Exception and cursor management**

➤ **What are Cursors?**

- 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:***
 - These are created by default when DML statements like, INSERT, UPDATE, and DELETE statements are executed. They are also created when a SELECT statement that returns just one row is executed.
- ***Explicit cursors:***
 - They must be created when you are executing a SELECT statement that returns more than one row. Even though the cursor stores multiple records, only one record can be processed at a time, which is called as current row. When you fetch a row the current row position moves to next row.
- Both implicit and explicit cursors have the same functionality, but they differ in the way they are accessed.

Table 6.1: Cursor Attributes

Name	Description
%FOUND	Returns TRUE if record was fetched successfully, FALSE otherwise.
%NOTFOUND	Returns TRUE if record was not fetched successfully, FALSE otherwise.
%ROWCOUNT	Returns number of records fetched from cursor at that point in time.
%ISOPEN	Returns TRUE if cursor is open, FALSE otherwise.

➤ **Implicit Cursors:**

- When you execute DML statements like DELETE, INSERT, UPDATE and SELECT statements, implicit statements are created to process these statements.
- Oracle provides few attributes called as implicit cursor attributes to check the status of DML operations. The cursor attributes available are %FOUND, %NOTFOUND, %ROWCOUNT, and %ISOPEN.
- For example, When you execute INSERT, UPDATE, or DELETE statements the cursor attributes tell us whether any rows are affected and how many have been affected. When a SELECT... INTO statement is executed in a PL/SQL Block, implicit cursor attributes can be used to find out whether any row has been returned by the SELECT statement. PL/SQL returns an error when no data is selected.
- The status of the cursor for each of these attributes are defined in the below table.
- For Example: Consider the PL/SQL Stock that uses implicit cursor attributes as shown below:

```

• DECLARE
    Eid number(3);
BEGIN
    UPDATE emp set eid=&eid where salary=&salary;
    eid:=sql%rowcount;
    IF SQL%found then
        dbms_output.put_line('success');
    ELSE
        dbms_output.put_line ( ' not' );
    END IF;
    dbms_output.put_line( 'rowcount' | |eid);
END;

```

➤ **Explicit Cursors**

- An explicit cursor is defined in the declaration section of the PL/SQL Block. It is created on a SELECT Statement which returns more than one row. We can provide a suitable name for the cursor.
- **The General Syntax for creating a cursor is as given below:**
 - **CURSOR cursor_name IS select_statement;**
 - ◆ *cursor_name* -A suitable name for the cursor.
 - ◆ *Select_statement* - A select query which returns multiple rows.

▪ How to use Explicit Cursor?

- There are four steps in using an Explicit Cursor.
 - ◆ DECLARE the cursor in the declaration section.
 - ◆ OPEN the cursor in the Execution Section.
 - ◆ FETCH the data from cursor into PL/SQL variables or records in the Execution Section.
 - ◆ CLOSE the cursor in the Execution Section before you end the PL/SQL Block.
- Declaring a Cursor in the Declaration Section:


```
DECLARE
  CURSOR emp_cur IS
  SELECT * FROM emp_tbl WHERE salary > 5000;
```
- In the above example we are creating a cursor 'emp_cur' on a query which returns the records of all the employees.
- employees with salary greater than 5000. Here 'emp_tbr' in the table which contains records of all the employees.
- Accessing the records in the cursor:
 - ◆ Once the cursor is created in the declaration section we can access the cursor in the execution section of the PL/SQL program,

▪ How to access an Explicit Cursor?

- These are the three steps in accessing the cursor.
 - ◆ Open the cursor.
 - ◆ Fetch the records in the cursor one at a time.
 - ◆ Close the cursor.
- General Syntax to open a cursor is:


```
OPEN cursor_name;
```
- General Syntax to fetch records from a cursor is:


```
FETCH cursor_name INTO record_name;
OR
FETCH cursor_name INTO variable_list;
```
- General Syntax to close a cursor is:


```
CLOSE cursor__name;
```

- When a cursor is opened, the first row becomes the current row. When the data is fetched it is copied to the record or variables and the logical pointer moves to the next row and it becomes the current row. On every fetch statement, the pointer moves to the next row. If you want to fetch after the last row, the program will throw an error. When there is more than one row in a cursor we can use loops along with explicit cursor attributes to fetch all the records.
- Points to remember while fetching a row:
 - ◆ We can fetch the rows in a cursor to a PL/SQL Record or a list of variables created in the PL/SQL Block.
 - ◆ If you are fetching a cursor to a PL/SQL Record, the record should have the same structure as the cursor.
 - ◆ If you are fetching a cursor to a list of variables, the variables should be listed in the same order in the fetch statement as the columns are present in the cursor.

▪ **General Form of using an explicit cursor is:**

```
DECLARE
    variables;
    records;
    create a cursor;
BEGIN
    OPEN cursor;
    FETCH cursor;
        process the records;
    CLOSE cursor;
END;
```

▪ Lets Look at the example below

• **Example 1:**

```
DECLARE
    CURSOR er IS select eid,name from emp order by name ;
    id emp.eid%type;
    ename emp.name%type;
BEGIN
    OPEN er;
```

```

    Loop
        FETCH er into id,ename;
        Exit when er%notfound;
        dbms_output.put_line (id || ename);
    end loop;
    close er;
END;

```

❖ Sub packages and packages

➤ PL/SQL Functions

➤ What is a Function in PL/SQL?

- A function is a named PL/SQL Block which is similar to a procedure. The major difference between a procedure and a function is, a function must always return a value, but a procedure may or may not return a value.
- The General Syntax to create a function is:

```

CREATE [OR REPLACE] FUNCTION function_name [parameters]
    RETURN return_datatype;
    {IS, AS}
    Declaration_section <variable,constant> ;
BEGIN
    Execution_section
    Return return_variable;
EXCEPTION
    exception section
    Return return_variable;
END;

```

- **Return Type:** The header section defines the return type of the function. The return datatype can be any of the oracle datatype like varchar, number etc.
- The execution and exception section both should return a value which is of the datatype defined in the header section.

- For example, let's create a function called 'emp_details_func'

```
CREATE OR REPLACE FUNCTION emp_details_func
RETURN VARCHAR2
IS
emp_name VARCHAR(20);
BEGIN
    SELECT name INTO emp_name
    FROM emp WHERE eid = '13';
    RETURN emp_name;
END;
```

- In the example we are retrieving the 'name' of employee with eid 13 to variable 'emp_name'.
- The return type of the function is VARCHAR2 which is declared in line no 2.
- The function returns the 'emp_name' which is of type VARCHAR2 as the return value in line no 9.

➤ How to execute a PL/SQL Function?

- A function can be executed in the following ways.
- Since a function returns a value we can assign it to a variable.


```
emp_name := emp_details_func;
```

 - If 'emp_name' is of datatype varchar2 we can store the name of the employee by assigning the return type of the function to it.
- As a part of a SELECT statement


```
SELECT emp_details_func FROM dual;
```
- In a PL/SQL Statements like,


```
dbms_output.put_line(emp_details_func);
```
- This line displays the value returned by the function

➤ What is a Stored Procedure?

- A **stored procedure** or in simple a **proc** is a named PL/SQL block which performs one or more specific task. This is similar to a procedure in other programming languages.
- A procedure has a header and a body.
- The header consists of the name of the procedure and the parameters or variables passed to the procedure.
- The body consists of declaration section, execution section and exception section similar to a general PL/SQL Block. A procedure is similar to an anonymous PL/SQL Block but it is named for repeated usage.
- We can pass parameters to procedures in three ways.
 - **IN type parameter:** These types of parameters are used to send values to stored procedures.
 - **OUT type parameter:** These types of parameters are used to get values from stored procedures. This is similar to a return type in functions.
 - **IN OUT parameter:** These types of parameters are used to send values and get values from stored procedures.
- **NOTE:** If a parameter is not explicitly defined a parameter type, then by default it is an IN type parameter.
- A procedure may or may not return any value.
- **General Syntax to create a procedure is:**

```
CREATE [OR REPLACE] PROCEDURE procedure_name (<Argument> {IN, OUT, IN OUT}
<Datatype>,...)
```

```
IS
```

```
    Declaration section<variable, constant> ;
```

```
BEGIN
```

```
    Execution section
```

```
EXCEPTION
```

```
    Exception section
```

```
END;
```

- **IS** - marks the beginning of the body of the procedure and is similar to DECLARE in anonymous PL/SQL Blocks. The code between IS and BEGIN forms the Declaration section.
- The syntax within the brackets [] indicate they are optional. By using CREATE OR REPLACE together the procedure is created if no other procedure with the same name exists or the existing procedure is replaced with the current code.
- The following illustrates the use of an OUT parameter:

```

CREATE TABLE T1 (a INTEGER,b INTEGER) ;

CREATE PROCEDURE p1 (a NUMBER, b OUT NUMBER)
AS
BEGIN
    b := 4;
    INSERT INTO T1 VALUES (a, b);
END;
/

DECLARE
    v NUMBER;
BEGIN
    P1(10, v);
END;
/

```

- How to execute a Stored Procedure?
 - There are two ways to execute a procedure.
 - ◆ From the SQL prompt.


```
EXECUTE [or EXEC] procedure_name;
```
 - ◆ Within another procedure – simply use the procedure name.
 - procedure_name;

➤ **Procedures VS Functions**

- Here are a few more differences between a procedure and a function:
 - A function **MUST** return a value
 - A procedure cannot return a value
 - Procedures and functions can both return data in OUT and IN OUT parameters
 - The return statement in a function returns control to the calling program and returns the results of the function
 - The return statement of a procedure returns control to the calling program and cannot return a value
 - Functions can be called from SQL, procedure cannot
 - Functions are considered expressions, procedure are not

➤ **Package**

▪ ***Components of an oracle package***

- A package has usually two components, a **specification** and a **body**.
- A package's **specification** declares the types (variables of the **record** type), memory variables, constants, exceptions, cursors, and subprograms that are available for use.
- A package's body fully defines cursors, functions, and procedures and thus implements the specification

▪ **Package specification**

- The package specification contains:
 - ◆ Name of the package
 - ◆ Names of the data types of any arguments
 - ◆ This declaration is local to the database and global to the package
- This means that procedures, functions, variables, constants, cursors and exceptions and other objects, declared in a package are accessible from anywhere in the package.
- Therefore, all the information a package needs, to execute a stored subprogram, is contained in the package specifications itself.
- **The simplified syntax for the create package statement is as follows:**

Create [or replace] package package_name

{is | as}

Package_specification

End package_name;

▪ **The package body**

- The body of a package contains the definition of public objects that are declared in the specification.
- The body can also contain other object declarations that are private to the package.
- The objects declared privately in the package body are not accessible to other objects outside the package.
- Unlike package specification, the **package body** can contain **subprogram bodies**.
- After the package is written, debugged, compiled and stored in the database applications can reference the package's types, call its subprograms, use its cursors, or raise its exceptions.
- **The simplified syntax for the create package body statement is as follows:**

Create [or replace] package body package_name

{is | as}

```
Package_body  
End package_name;
```

➤ Example package specification and package body

- **Sql> create or replace package pkg1**
2 as
3 function area (rad number) return number;
4 procedure print (str1 varchar2 := 'hello',
5 str2 varchar2 := 'world',
6 end varchar2 := '!') ;
7 end;
8 /
- Package created.

- **Sql> create or replace package body pkg1**
2 as
3 function area (rad number) return number
4 is
5 pi number := 3.14;
6 begin
7 return pi * (rad ** 2);
8 end;
9
10 procedure print(str1 varchar2 := 'hello',
11 str2 varchar2 := 'world',
12 end varchar2 := '!')
13 is
14 begin
15 dbms_output.put_line(str1 || ',' || str2 || end);
16 end;
17 end;
18 /
- Package body created.

➤ **Advantages package:**

- Packages enable the organization of commercial applications into efficient modules. Each package is easily understood and the interfaces between packages are simple, clear and well defined
- Packages allow granting of privileges efficiently
- A package's public variables and cursors persist for the duration of the session. Therefore all cursors and procedures that execute in this environment can share them
- Packages enable the overloading of procedures and functions when required
- Packages improve performance by loading multiple objects into memory at once. Therefore, subsequent calls to related subprograms in the package require no i/o
- Packages promote code reuse through the use of libraries that contain stored procedures and functions, thereby reducing redundant coding