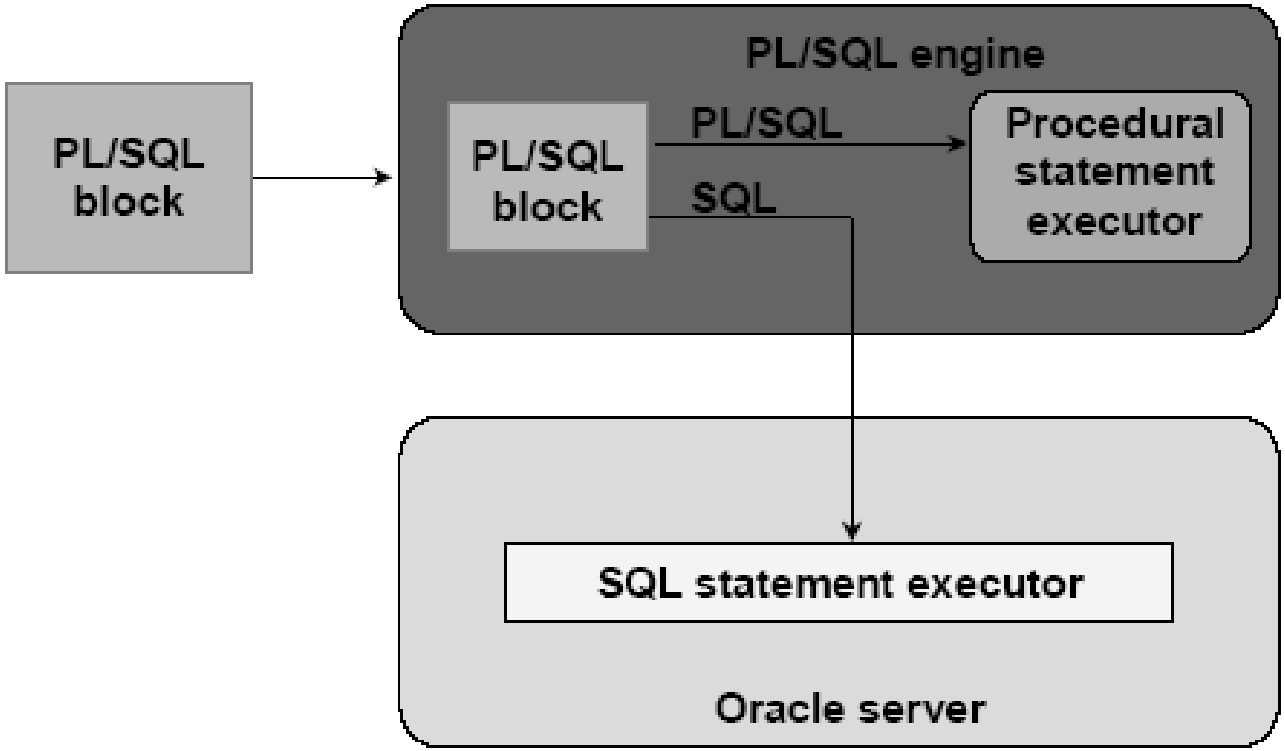
PL/SQL

- Introduction
- Structure of a block
- Variables and types
- Accessing the database
- Control flow
- Cursors
- Exceptions
- Procedures and functions

Introduction

- 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.

PL/SQL Environment



Benefits

- Integrate of database technology and procedural programming capabilities
- Provide improved performance of an application
- Modularize program development
- Portable among host environments supporting Oracle server and PL/SQL
- Handle errors

Structure of a block

```
DECLARE
  /* Declarative section: variables, types, cursors,
    user-defined exceptions */
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; /* mandatory */
```

Block (1)

- The only SQL statements allowed in a PL/SQL program are SELECT, INSERT, UPDATE, DELETE and several other data manipulation statements plus some transaction control.
- Data definition statements like CREATE, DROP, or ALTER are not allowed.
- The executable section also contains constructs such as assignments, branches, loops, procedure calls, and triggers

Block (2)

- PL/SQL is not case sensitive. C style comments (/* ... */) may be used.

- To execute a PL/SQL program, we must follow the program text itself by: a line with a single dot ("."), and then a line with “run”; or a line with “/”

- We can invoke a PL/SQL program either by typing it in sqlplus or by putting the code in a file and invoking the file

Variables and Types (1)

- Information is transmitted between a PL/SQL program and the database through variables.
- Every variable has a specific type associated with it, that can be:
  - One of the types used by SQL for database columns
  - A generic type used in PL/SQL such as NUMBER
  - Declared to be the same as the type of some database column

Variables and Types (2)

- The most commonly used generic type is NUMBER. Variables of type NUMBER can hold either an integer or a real number.
- The most commonly used character string type is VARCHAR(*n*), where *n* is the maximum length of the string in bytes. This length is required, and there is no default.

Example

```
DECLARE
    price  NUMBER;
   book_name VARCHAR(20);
```

Declaring Boolean variables

- Only the values TRUE, FALSE, and NULL can be assigned to a Boolean variable.
- The variables are compared by the logical operators AND, OR, and NOT.
- The variables always yield TRUE, FALSE, or NULL.
- Arithmetic, character, and date expressions can be used to return a Boolean value.

Example

```
DECLARE
   v_flag BOOLEAN := FALSE;
BEGIN
   v_flag := TRUE;
END;
```

The %TYPE attribute

- A PL/SQL variable can be used to manipulate data stored in a existing relation.
- The variable must have the same type as the relation column. If there is any type mismatch, variable assignments and comparisons may not work the way you expect.
- To be safe, instead of hard coding the type of a variable, you should use the %TYPE operator.

Ex:

```
DECLARE mybook books.name%TYPE;
```

gives PL/SQL variable mybook whatever type was declared for the name column in relation books.

The %ROWTYPE attribute

- A variable may also have a type that is a record with several fields.
- The simplest way to declare such a variable is to use %ROWTYPE on a relation name. **The result is a record type in which the fields have the same names and types as the attributes of the relation.**

Ex:

```
DECLARE    bookTuple Books%ROWTYPE;
```

makes variable bookTuple be a record with fields: name and author, assuming that the relation has the schema Books(name, author).

%ROWTYPE Example

```
DECLARE
   customer_rec   customers%rowtype;
BEGIN
   SELECT * into customer_rec
   FROM customers
   WHERE id = 5;
   dbms_output.put_line('Customer ID: ' || customer_rec.id);
   dbms_output.put_line('Customer Name: ' || customer_rec.name);
   dbms_output.put_line('Customer Address: ' || customer_rec.address);
   dbms_output.put_line('Customer Salary: ' || customer_rec.salary);
END;
```

Output:

Customer ID: 5
Customer Name: Hardik
Customer Address: Bhopal
Customer Salary: 9000

PL/SQL procedure completed.

Default values and assignments

- The initial value of any variable, regardless of its type, is NULL.
- We can assign values to variables, using the ":=" operator.
- The assignment can occur either immediately after the type of the variable is declared, or anywhere in the executable portion of the program.

Ex:

```
DECLARE
  a NUMBER := 3;
BEGIN
  a := a + 1;
END;
```

Bind variables

- Variable that you declare in a host environment (ex: SQL*Plus), used to pass run-time values (number or characters) into or out of PL/SQL programs
- The only kind that may be printed with a print command.
- Bind variables must be prefixed with a colon in PL/SQL statements

Steps to create a bind variable

1. We declare a bind variable as follows:
   VARIABLE <name> <type>
   where the type can be only one of three things: NUMBER, CHAR, or CHAR(*n*).
2. We may then assign to the variable in a following PL/SQL statement, but we must prefix it with a colon.
3. Finally, we can execute a statement :
   PRINT :<name>;
   outside the PL/SQL statement

Example

```
VARIABLE x NUMBER
BEGIN
    :x := 1;
END;
.
run;
PRINT :x;
```

DBMS_OUTPUT.PUT_LINE package

- An Oracle-supplied packaged procedure
- An alternative for displaying data from a PL/SQL block
- Must be enabled in SQL*Plus with: **SET SERVEROUTPUT ON**

```
SET SERVEROUTPUT ON
DEFINE p_annual_sal = 60000
```

```
DECLARE
  v_sal NUMBER(9,2) := &p_annual_sal;
BEGIN
  v_sal := v_sal/12;
  DBMS_OUTPUT.PUT_LINE ('The monthly salary is ' ||
                        TO_CHAR(v_sal));
END;
/
```

Simple programs accessing the database

- The simplest form of program has some declarations followed by an executable section consisting of one or more of the SQL statements with which we are familiar.
- After the SELECT clause, we must have an INTO clause listing variables, one for each attribute in the SELECT clause, into which the components of the retrieved tuple must be placed.
- The SELECT statement in PL/SQL only works if the result of the query contains a single tuple. If the query returns more than one tuple, you need to use a cursor

Example (SQL)

```
CREATE TABLE T1(
      e INTEGER,
      f INTEGER );

DELETE FROM T1;

INSERT INTO T1 VALUES(1, 3);
INSERT INTO T1 VALUES(2, 4);
```

Example (PL/SQL)

```
DECLARE
     a NUMBER;
     b NUMBER;
BEGIN
   SELECT e,f INTO a,b FROM T1 WHERE e>1;
   INSERT INTO T1 VALUES(b,a);
END;
  .
run;
```

Control flow

- PL/SQL allows you to branch and create loops in a fairly familiar way.
- An IF statement looks like:

```
IF <condition> THEN <statement_list>
ELSE <statement_list>
END IF;
```

- The ELSE part is optional. If you want a multiway branch, use:

```
IF <condition_1> THEN ...
ELSIF <condition_2> THEN ... ... ...
ELSIF <condition_n> THEN ...
ELSE ...
END IF;
```

Example

```
DECLARE
 a NUMBER;
 b NUMBER;
BEGIN
     SELECT e,f INTO a,b FROM T1 WHERE e>1;
     IF b=1 THEN
          INSERT INTO T1 VALUES (b,a);
     ELSE
          INSERT INTO T1 VALUES (b+10,a+10);
     END IF;
END;
.
run;
```

Write a program accept the value of A,B&C display which is greater

```
DECLARE
A NUMBER:=&A;
B NUMBER:=&B;
C NUMBER:=&C;
BEGIN
IF (A>B AND A>C) THEN
DBMS_OUTPUT.PUT_LINE('A IS GREATER '||''||A);

ELSIF B>C THEN
DBMS_OUTPUT.PUT_LINE('B IS GREATE '||''||B);

ELSE
DBMS_OUTPUT.PUT_LINE('C IS GREATER '||''||C);

END IF;
END;
/
```

Loops

- Loops are created with the following:

```
LOOP
  <loop_body> /* A list of
   statements. */
 END LOOP;
```

- At least one of the statements in <loop_body> should be an EXIT statement of the form

```
EXIT WHEN <condition>;
```

- The loop breaks if <condition> is true.

Example

```
DECLARE
   i NUMBER := 1;
BEGIN
    LOOP
        INSERT INTO T1 VALUES (i,i);
              i := i+1;
        EXIT WHEN i>100;
    END LOOP;
END;
  .
run;
```

Write a Pl/SQL BlockSum of odd number between 1 to 100 (GTU old papers)

```
Declare
num number:=1;
total  number:=0;
begin
loop
total:=total+num;
num := num+2;
exit when num>=100;
end loop;
dbms_output.put_line (total);
end;
```

Output: 2500

Other useful loop-forming statements

- A WHILE loop can be formed with:

```
WHILE <condition> LOOP
   <loop_body>
END LOOP;
```

- A simple FOR loop can be formed with:

```
FOR <var> IN <start>..<finish> LOOP
  <loop_body>
END LOOP;
```

- Here, <var> can be any variable; it is local to the for-loop and need not be declared. Also, <start> and <finish> are constants.

Write a PL/SQL code for multiplication table (while loop)

```
DECLARE
        A NUMBER:=&A;
        B NUMBER:=1;
        C NUMBER;
BEGIN
        WHILE B <=10
        LOOP
        C:=A*B;
        DBMS_OUTPUT.PUT_LINE(A||'*'||B||'='||C);
        B:=B+1;
        END LOOP;
END;
/
```

Write a PL/SQL code block to find factorial of a number. (for loop)

```
declare
n number;
i number;
f number:=1;
begin
n:=&n;
for i in 1..n
loop
f:=f*i;
end loop;
dbms_output.put_line(n||'! = '||f);
end;
```

Cursors

- A cursor is a variable that runs through the tuples of some relation.
  - Can be a stored table or the answer to some query.
- By fetching into the cursor each tuple of the relation, we can write a program to read and process the value of each such tuple.
- If the relation is stored, we can also update or delete the tuple at the current cursor position.

Cursors (contd.)

- A **cursor** is a pointer to a memory area, known as the context area, for processing an SQL statement, which contains all the information needed for processing the statement; for example, the number of rows processed, etc.
- PL/SQL controls the context area through a cursor.
- A cursor holds the rows (one or more) returned by a SQL statement.
- The set of rows the cursor holds is referred to as the **active set**.
- The rows returned by the SQL statement, one at a time.
- There are two types of cursors:
  - Implicit cursors
  - Explicit cursors

Implicit Cursors

- Implicit cursors are automatically created by Oracle whenever an SQL statement is executed, when there is no explicit cursor for the statement.
- Programmers cannot control the implicit cursors and the information in it.
- Whenever a DML statement (INSERT, UPDATE and DELETE) is issued, an implicit cursor is associated with this statement.
- For INSERT operations, the cursor holds the data that needs to be inserted.
- For UPDATE and DELETE operations, the cursor identifies the rows that would be affected.

Cursor Attributes

Attribute	Description
%FOUND	Returns TRUE if an INSERT, UPDATE, or DELETE statement affected one or more rows or a SELECT INTO statement returned one or more rows. Otherwise, it returns FALSE.
%NOTFOUND	The logical opposite of %FOUND. It returns TRUE if an INSERT, UPDATE, or DELETE statement affected no rows, or a SELECT INTO statement returned no rows. Otherwise, it returns FALSE.
%ISOPEN	Always returns FALSE for implicit cursors, because Oracle closes the SQL cursor automatically after executing its associated SQL statement.
%ROWCOUNT	Returns the number of rows affected by an INSERT, UPDATE, or DELETE statement, or returned by a SELECT INTO statement.

Cursor Example

- The following program will update the table and increase the salary of each customer by500 and use the **SQL%ROWCOUNT** attribute to determine the number of rows affected:

```
DECLARE
    total_rows number(2);
BEGIN
    UPDATE customers
    SET salary = salary + 500;
    IF sql%notfound THEN
        dbms_output.put_line('no customers selected');
    ELSIF sql%found THEN
        total_rows := sql%rowcount;
      dbms_output.put_line( total_rows || ' customers selected ');
   END IF;
END;
/
```

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

```
6 customers selected


PL/SQL procedure successfully completed.
```

Explicit Cursors

Explicit cursors are programmer-defined cursors for gaining more control over the **context area**. An explicit cursor should be defined in the declaration section of the PL/SQL Block. It is created on a SELECT Statement which returns more than one row.

The syntax for creating an explicit cursor is:

```
CURSOR cursor_name IS select_statement;
```

Working with an explicit cursor includes the following steps:

- Declaring the cursor for initializing the memory
- Opening the cursor for allocating the memory
- Fetching the cursor for retrieving the data
- Closing the cursor to release the allocated memory

Cursor Declaration and usage

Declaring the Cursor

Declaring the cursor defines the cursor with a name and the associated SELECT statement. For example:

```
CURSOR c_customers IS
   SELECT id, name, address FROM customers;
```

Opening the Cursor

Opening the cursor allocates the memory for the cursor and makes it ready for fetching the rows returned by the SQL statement into it. For example, we will open the above-defined cursor as follows:

```
OPEN c_customers;
```

Fetching and Closing Cursor

Fetching the Cursor

Fetching the cursor involves accessing one row at a time. For example, we will fetch rows from the above-opened cursor as follows:

```
FETCH c_customers INTO c_id, c_name, c_addr;
```

Closing the Cursor

Closing the cursor means releasing the allocated memory. For example, we will close the above-opened cursor as follows:

```
CLOSE c_customers;
```

Explicit Cursor Example

Following is a complete example to illustrate the concepts of explicit cursors:

```
DECLARE
   c_id customers.id%type;
   c_name customerS.No.ame%type;
   c_addr customers.address%type;
   CURSOR c_customers is
      SELECT id, name, address FROM customers;
BEGIN
   OPEN c_customers;
   LOOP
      FETCH c_customers into c_id, c_name, c_addr;
      EXIT WHEN c_customers%notfound;
      dbms_output.put_line(c_id || ' ' || c_name || ' ' || c_addr);
   END LOOP;
   CLOSE c_customers;
END;
/
```

Output of Explicit Cursor

```
1 Ramesh Ahmedabad

2 Khilan Delhi

3 kaushik Kota

4 Chaitali Mumbai

5 Hardik Bhopal

6 Komal MP


PL/SQL procedure successfully completed.
```

Procedures vs Functions

- 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.

Procedures

- Behave very much like procedures in other programming language
- A procedure is introduced by the keywords CREATE PROCEDURE followed by the procedure name and its parameters.
- CREATE may be followed by OR REPLACE.
  - If the procedure is already created, we will not get an error
  - If the previous definition is a different procedure with the same name, the old procedure will be lost.

Stored Procedures

- The first line is called the Procedure Specification
- The remainder is the Procedure Body
- A procedure is compiled and loaded in the database as an object
- Procedures can have parameters passed to them

Parts of a PL/SQL Subprogram

S.No	Parts & Description
1	**Declarative Part** It is an optional part. However, the declarative part for a subprogram does not start with the DECLARE keyword. It contains declarations of types, cursors, constants, variables, exceptions, and nested subprograms. These items are local to the subprogram and cease to exist when the subprogram completes execution.
2	**Executable Part** This is a mandatory part and contains statements that perform the designated action.
3	**Exception-handling** This is again an optional part. It contains the code that handles run-time errors.

Creating a Procedure

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

Where,

- *procedure-name* specifies the name of the procedure.
- [OR REPLACE] option allows the modification of an existing procedure.
- The optional parameter list contains name, mode and types of the parameters. **IN** represents the value that will be passed from outside and OUT represents the parameter that 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.

Parameters in Procedure

- In PL/SQL, we can pass parameters to procedures and functions in three ways.

1) IN type parameter: These types of parameters are used to send values to stored procedures.
2) OUT type parameter: These types of parameters are used to get values from stored procedures. This is similar to a return type in functions.
3) 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.

Stored Procedures

- Run a procedure with the PL/SQL EXECUTE command
- Parameters are enclosed in parentheses

IN & OUT Mode Example

```
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  // Calling procedure
   a:= 23;
   b:= 45;
   findMin(a, b, c);
   dbms_output.put_line(' Minimum of (23, 45) : ' || c);
END;
/
```

Output:

Minimum of (23, 45) : 23

PL/SQL procedure successfully completed.

Write a Pl/SQl code to compute the square of value of a passed value.

```
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;
/
```

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

Square of (23): 529

PL/SQL procedure successfully completed.

Methods for Passing Parameters

- Positional Notation

  In positional notation, you can call the procedure as −

  - findMin(a, b, c, d);
  - In positional notation, the first actual parameter is substituted for the first formal parameter; the second actual parameter is substituted for the second formal parameter, and so on. So, **a** is substituted for **x, b** is substituted for **y, c** is substituted for **z** and **d** is substituted for **m**.

- Named Notation
  - findMin(x => a, y => b, z => c, m => d);
- Mixed Notation
  - In mixed notation, you can mix both notations in procedure call; however, the positional notation should precede the named notation.

  - The following call is legal −
    - findMin(a, b, c, m => d);
  - However, this is not legal:
    - findMin(x => a, b, c, d);

Functions

CREATE [OR REPLACE] FUNCTION function_name

[(parameter_name [IN | OUT | IN OUT] type [, ...])]

RETURN return_datatype

{IS | AS}

BEGIN

  < function_body >

END [function_name];

Functions parameters

- *function-name* specifies the name of the function.
- [OR REPLACE] option allows the modification of an existing function.
- The optional parameter list contains name, mode and types of the parameters. IN represents the value that will be passed from outside and OUT represents the parameter that will be used to return a value outside of the procedure.
- The function must contain a **return** statement.
- The *RETURN* clause specifies the data type you are going to return from the function.
- *function-body* contains the executable part.
- The AS keyword is used instead of the IS keyword for creating a standalone function.

Function example

The following example demonstrates Declaring, Defining, and Invoking a Simple PL/SQL Function that computes and returns the maximum of two values.

```
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;
/
```

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

```
Maximum of (23,45): 45


PL/SQL procedure successfully completed.
```

Triggers

- Associated with a particular table
- Automatically executed when a particular event occurs
  - Insert
  - Update
  - Delete
  - Others

Trigger Benefits

- 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

Triggers vs. Procedures

- Procedures are explicitly executed by a user or application
- Triggers are implicitly executed (fired) when the triggering event occurs
- Triggers should not be used as a lazy way to invoke a procedure as they are fired every time the event occurs

Triggers

```
CREATE TRIGGER TriggerName
BEFORE [AFTER] event[s] ON TableName
[FOR EACH ROW]
DECLARE
   Declaration of any local variables
BEGIN
   Statements in Executable section
EXCEPTION
   Statements in optional Exception section
END;
/
```

Triggers

- The trigger specification names the trigger and indicates when it will fire
- The trigger body contains the PL/SQL code to accomplish whatever task(s) need to be performed

Triggers Timing

- A triggers timing has to be specified first
  - Before (most common)
    - Trigger should be fired before the operation
      - i.e. before an insert
  - After
    - Trigger should be fired after the operation
      - i.e. after a delete is performed

Trigger Events

- Three types of events are available
  - DML events
  - DDL events
  - Database events

DML Events

- Changes to data in a table
  - Insert
  - Update
  - Delete
- Can specify one or more events in the specification
  - i.e. INSERT OR UPDATE OR DELETE
- Can specify one or more columns to be associated with a type of event
  - i.e. BEFORE UPDATE OF SID OR SNAME

DDL Events

- Changes to the definition of objects
  - Tables
  - Indexes
  - Procedures
  - Functions
  - Others
    - Include CREATE, ALTER and DROP statements on these objects

Database Events

- Server Errors
- Users Log On or Off
- Database Started or Stopped

Trigger Level

- Two levels for Triggers
  - Row-level trigger
    - Requires FOR EACH ROW clause
      - If operation affects multiple rows, trigger fires once for each row affected
  - Statement-level trigger
  - DML triggers should be row-level
  - DDL and Database triggers should not be row-level

Trigger Example

Example

To start with, we will be using the CUSTOMERS table we had created and used in the previous chapters –

```
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 |
+----+----------+-----+-----------+----------+
```

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 the SQL prompt, it produces the following result −

```
Trigger created.
```

Triggering a Trigger

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

When a record is created in the CUSTOMERS table, the 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, old salary is not available and the above result comes as null. Let us now perform one more DML operation on the CUSTOMERS table. The UPDATE statement will update an existing record in the table –

```
UPDATE customers
SET salary = salary + 500
WHERE id = 2;
```

When a record is updated in the CUSTOMERS table, the 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
```

Exceptions

- An exception is an identifier in PL/SQL that is raised during execution.
- How is it raised?
  - An Oracle error occurs.
  - You raise it explicitly.
- How do you handle it?
  - Trap it with a handler.
  - Propagate it to the calling environment.